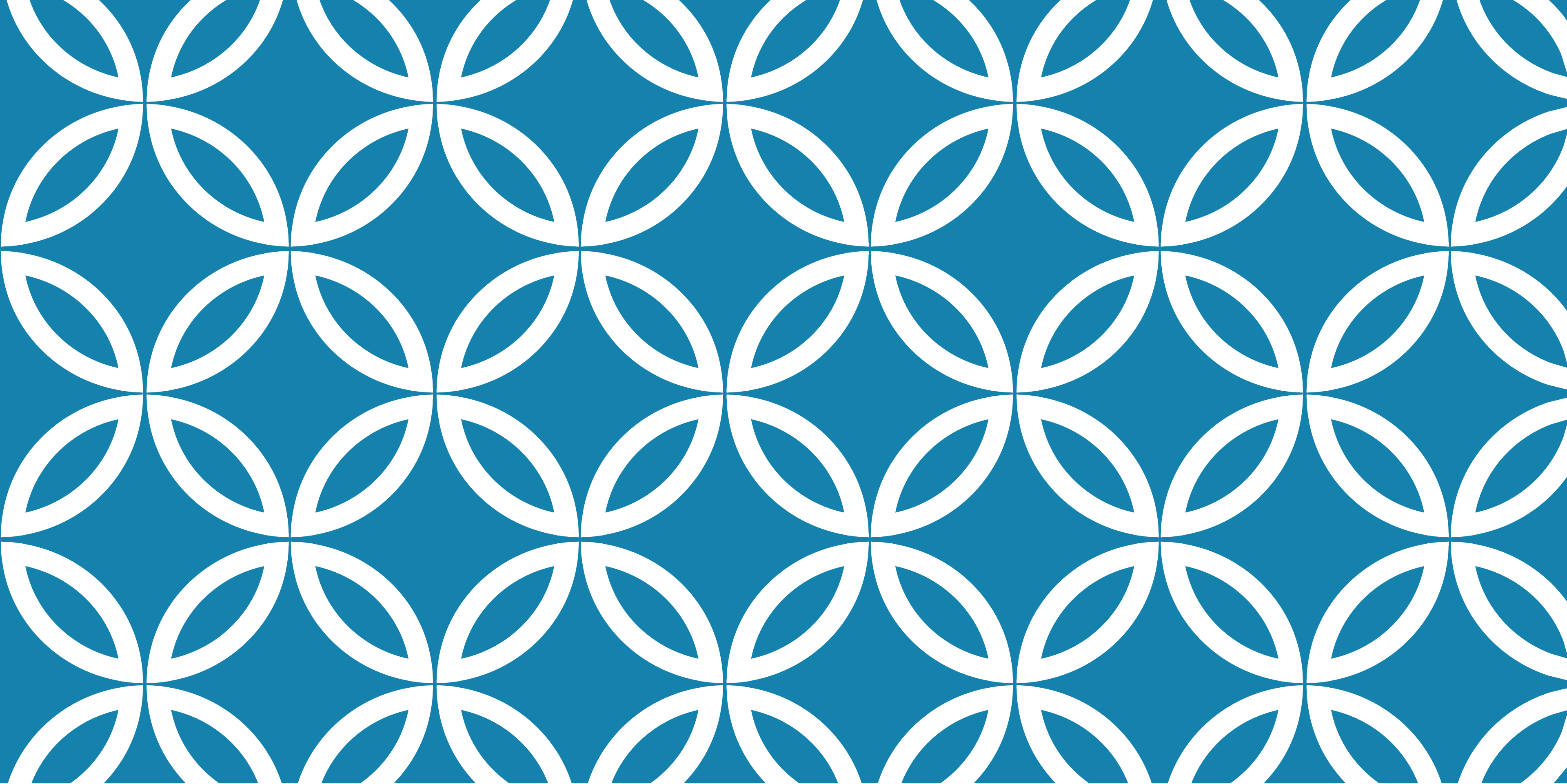


CT102: ALGORITHMS

Searching Linear Lists

SEARCHING

- Searching is a fundamental operation in computing – seen in many different areas and across many problems
- Most programming languages have built in searching functions that can be used at a later stage (but not in Algorithms!).

Searching and Sorting Functions

Function	Search or Sort
bsearch	Binary search
bsearch_s	A more secure version of **bsearch**
_lfind	Linear search for given value
_lfind_s	A more secure version of **_lfind**
_lsearch	Linear search for given value, which is added to array if not found
_lsearch_s	A more secure version of **_lsearch**
qsort	Quick sort
qsort_s	A more secure version of **qsort**

LINEAR SEARCH: OUTLINE

Problem: In a linear data structure, find the position of a given *item*, returning the position the item is found or else value -1 if item is not found.

Input: Array arrA with (distinct) values; Size of array (size); item to find (`item`)

Output: *one* integer value indicating not found (-1) or the position *item* was found at.

Algorithm idea: Start at index position 0, for each position until the end of array, keep checking if value at current position is the item required; once finished searching, output result

LINEAR SEARCH: EXAMPLES

Searching for "Enya" in array `names[]`

`names[5]`	Louis	Ben	Enya	Don	Ali

Searching for 100 in array `costs[]`:

`costs[5]`	1007	2002	2007	1000	2003

LINEAR SEARCH ALGORITHM (FRAGMENT OF C)

```
int i;
int position = -1;

for (i = 0; i < size; i++){
    if (arrA[i] == item) {
        position = i; //found
    }
}
```

WHAT DO WE NEED TO ADD TO THIS TO ACTUALLY RUN IT?

```
int i;
int position = -1;

for (i = 0; i < size; i++){
    if (arrA[i] == item) {
        position = i; //found
    }
}
```

- Need `#include <stdio.h>` and `main(){ … }`
- Need data! An array of integers, its size and an item to find
- … for now, we will hard code the array data but we should enter search item (using `scanf`)
- Need to output answer … use `printf()` statement(s)

FULL CODE WHEN WORKING WITH INTEGERS:

QUESTIONS

```
int i;
int position = -1;

for (i = 0; i < size; i++){
    if (arrA[i] == item) {
        position = i; //found
    }
}
```

Consider this code with a sample array (as given):

```
int arrA[6] = {12, 6, 4, 2, 13, 19};
```

And searching for `item = 6`

How does the algorithm progress?

QUESTIONS:
How fast/slow is it?

```
int i;
int position = -1;

for (i = 0; i < size; i++){
    if (arrA[i] == item) {
        position = i; //found
    }
}
```

WORKSHEET 2 QUESTION 1:

```
int i;
int position = -1;

for (i = 0; i < size; i++){
    if (arrA[i] == item) {
        position = i; //found
    }
}
```

What inefficiencies can you see in this version of a linear search assuming integer array **`arrA[]`** of size **`size`** and searching for item? (i.e. assume you are given an array and the item)

WRITING A BETTER VERSION?

ORDERED (SORTED) ARRAYS

An array is **ordered** if its values are in either ascending or descending order

In an **ascending array**, the value of each element is less than (or equal to if duplicates allowed) the value of the next element.

`names[]`	Aaron	Ali	Cait	Dara	Eli

In a **descending array**, the value of each element is greater than (or equal to if repeats allowed) to the value of the next element.

`years[]`	2022	2020	2017	2015	2012

WORKING WITH SORTED DATA …

Question was … Any efficiencies that can be made to the linear search if we can assume that the data is in sorted order with distinct (no repeating) values?

For example, array:

```
int arrA[6] = {2, 6, 14, 29, 32, 49};
```

and searching for `item = 9`

WORKSHEET 2 QUESTION 2 CODE:

But, if data in array is sorted, can have an even better approach, using a BINARY SEARCH …

BINARY SEARCH: OUTLINE

Problem: In a linear data structure with data in sorted order, with no duplicates, find the position of a given *item*, returning the position found or else value -1 if not found.

Input: Array arrA[] with data values is sorted order; Size of array (size); item to find (`item`)

Output: *one* integer value indicating not found (-1) or the position *item* was found at.

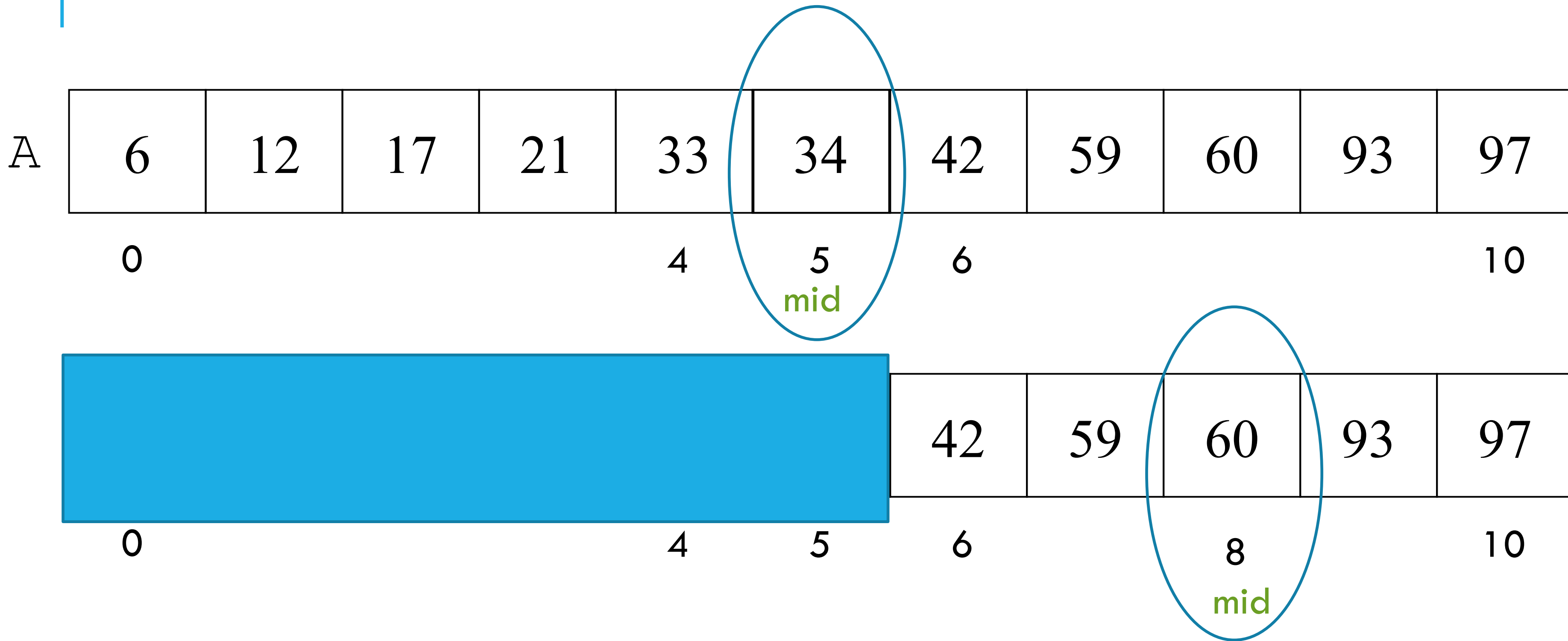
Assumptions: Without loss of generality, we will assume the input array contains integer values and that the values are sorted in *ascending* order.

BINARY SEARCH: IDEA

Until there is nothing left to search:

- Start, as close as possible to the middle of the array
- Check if value at middle position is the item required
  - If yes, stop and return position
  - If no, check whether the value required is less than or greater than the value at the middle position. As a result of this check, repeat the search with the lower (left) or upper (right) portion of the array.

EXAMPLE:
Search for item = 60 in array A of size 11



Note: This has taken two checks to find item in comparison to ??? if using the linear search code?

FINDING MIDDLE POSITION OF ARRAY:

Somewhere “near” the middle

Depending on size of array or sub-arrays may not always be “exact” middle

First mid in array of size `size`

```
int begSec, endSec, mid;
begSec = 0;
endSec = size - 1;
mid = int((begSec + endSec)/ 2);
```

RECALL: `int()`

`int(n)` returns the truncation of n

That is, the integer whose absolute value is no greater than that of `n`

WORKING WITH `mid` ...

Compare `item` with arr`A[mid]` ... middle value in array

Three possible situations:

- `item == A[mid]` so can stop
- `item > A[mid]` so continue searching **upper half** of array (from mid + 1 to size -1)
- `item < A[mid]` so continue searching **lower half** of array (from 0 to mid - 1)

EXAMPLE: SEARCH FOR ITEM = 60 IN ARRAY A OF SIZE 11

A	6	12	17	21	33	34	42	59	60	93	97
	0				4	5 mid	6				10

UPDATING `mid`...

```
if (item > arrA[mid]) {
      begSec = mid + 1;
}
else if (item < arrA[mid]) {
      endSec = mid - 1;
}
mid = int( (begSec + endSec) / 2);
```

STOPPING CONDITION

```
while (
    begSec <= endSec &&
    arrA[mid] != item)
{
```

That is, will stop when have checked all possible locations in the `[begSec, endSec]` range or have found item

LOOKING AT CODE AND SHARING CODE ..

```
// binary search of a sorted int array,
// given array, size and item to find;
// returning location
int binarySearch(int arrA[], int size, int item) {

    printf("*** Assuming Data is in Sorted Order ***");

    int begSec, endSec, mid, location;
    begSec = 0;
    endSec = size - 1;
    mid = int ((begSec + endSec) / 2);

    location = -1;

    while (begSec <= endSec && arrA[mid] != item) {
        if (item > arrA[mid]) {
            begSec = mid + 1;
        }
        else if (item < arrA[mid]) {
            endSec = mid - 1;
        }

        mid = int ((begSec + endSec) / 2);
    }
    if (arrA[mid] == item) {
        location = mid;
    }

    return (location);
}
```

WORKING WITH THE CODE

- What function declaration do we need for this?

```
40  int binarySearch(int arrA[], int size, int item) {
41
```

- What data needs to be passed to the function?

- How do we “call” function?

- And why did we write this as a function anyway?

HOW “LONG” DOES IT TAKE? (HOW FAST/SLOW?)

- How does it compare to linear search? [Will return to this]
- Can you write the linear search code as a function?

HOW TO CHECK ARRAY IS SORTED?

Binary Search assumes that the data in the array is in sorted order - algorithm will not work correctly unless this assumption holds

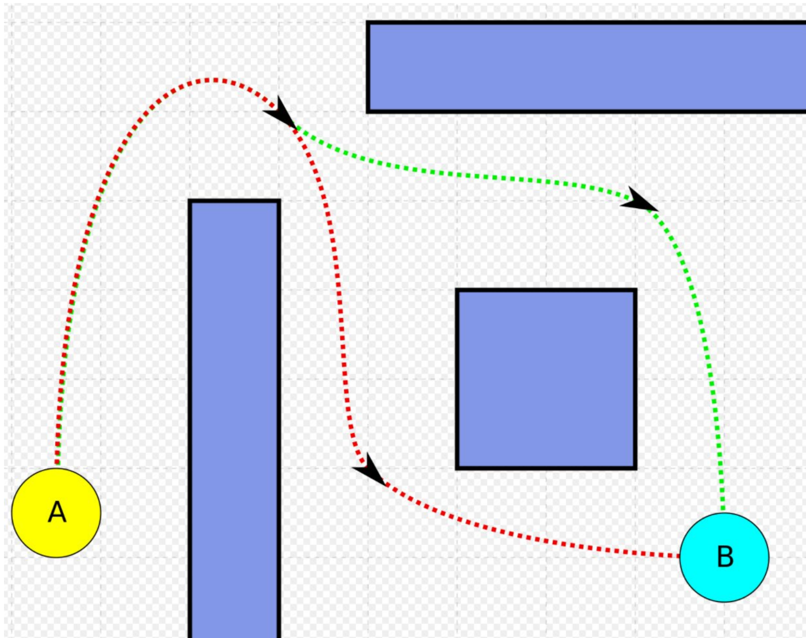
? How to check if an array of integers is in sorted ascending order (no duplicates)? See worksheet 2, question 4.

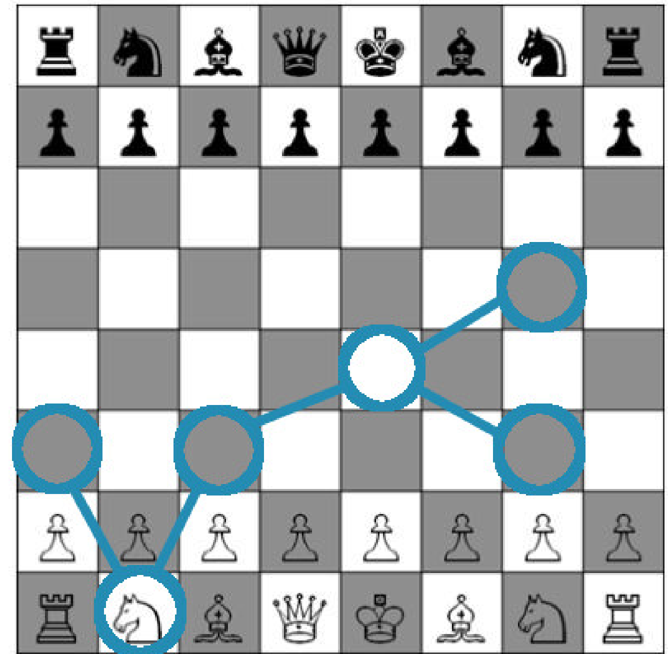
? If not sorted, how to sort [Later topic]

ADVANCED SEARCHING: PATHFINDING ALGORITHMS

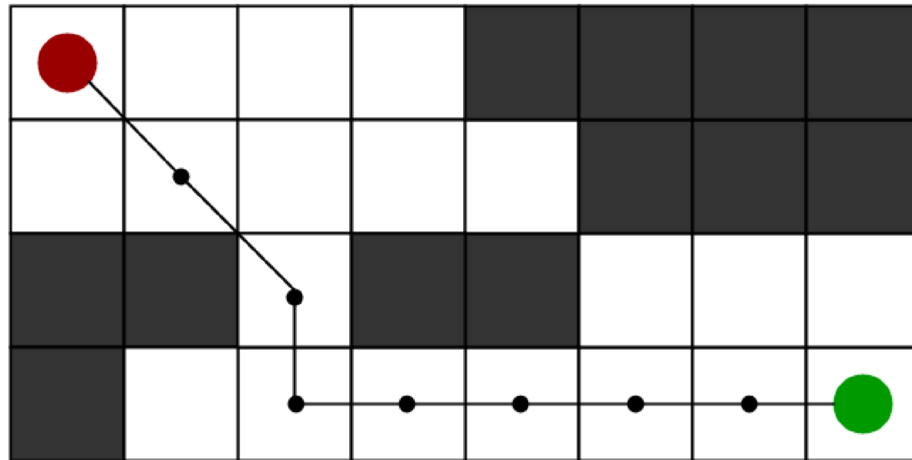
- A particularly important type of (non-linear) searching is called pathfinding which involves finding the shortest route between two paths.
- This relates to a topics already seen in Social Network Analysis – the finding the shortest path between two points in a large network.

EXAMPLES





GENERAL APPROACH



Need to keep track of:

- Nodes/Points/Spaces already visited
- Nodes (Neighbours) of current node where you can move to …. If there are a number of options here, must pick one point to move to, but potentially "explore" other points later on if the current choice does not result in success and delete nodes on the path that did not lead to success.

SUMMARY

Searching is a fundamental operation in computing

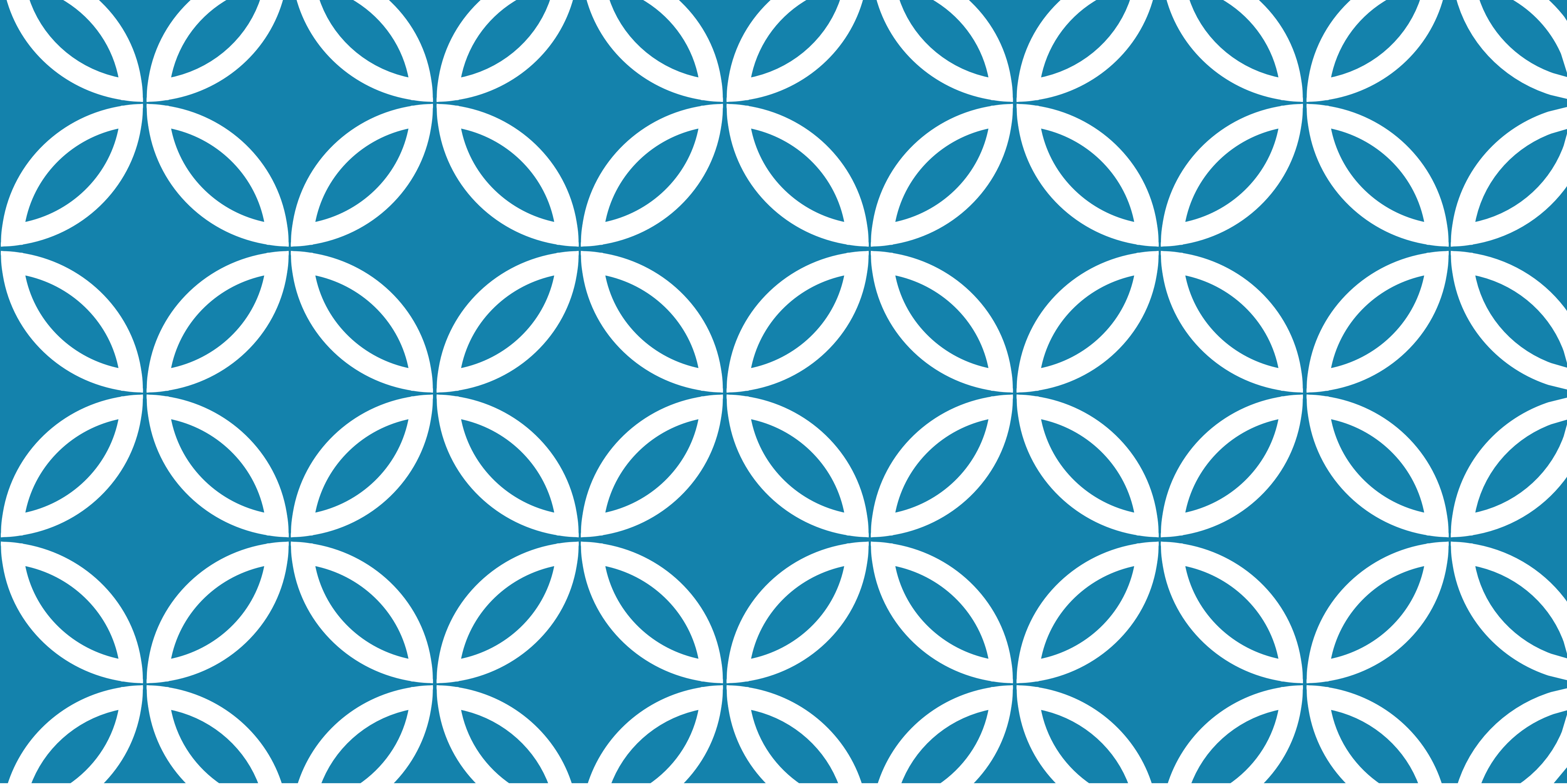
Very few applications are built that don't involve a search feature (think about this!)

The most general search that works on any type of data is a **linear search** – for an array of size N, at most N items will have to be checked

If data is in a sorted order then a much more efficient search can be used – **binary search** - for an array of size N, at most $\log_2(N)$ items will have to be checked.

TUTORIAL THURSDAY ....

If you are confused about any of this material or with any of the C code used please come to the tutorial on Thursday where we can go through the code in more detail



CT102: ALGORITHMS AND INFORMATION SYSTEMS

Topic:

Algorithms and Algorithm Analysis

FOCUS OF ALGORITHM ANALYSIS …

The analysis and comparison of algorithms with respect to resources used, that is:

- Space (memory)
- Time (to run)

Important Question for us: When two programs solve the same problem differently why is one better than the other?

FOR MEANINGFUL COMPARISON, MUST HAVE SOME *STANDARDISATION*

- both programs in the same language
- executable code produced by the same compiler
- the same hardware platform for each test
- identical data sets to test both programs
- data sets that test many cases (expected, unexpected, edge/corner cases)

3 APPROACHES USED FOR COMPARISON

1. Actual Time: Code and Run and track time
2. Estimate time: Count time "steps" in code (without running)
3. Estimate rate of growth of time used: Use Big-O notation (and others) for large input sizes

APPROACH 1 … CODE AND RUN

- Get actual values for time and space
- Often focus on *key operations* as well as overall time to run (i.e. focus on time functions that do the main work take, not Input/output work)

- *Generally in practice,* … want a (good) idea of the time and space efficiency of an algorithm **before** we fully code a solution

APPROACH 1 IN C .. using `time.h`

```
clock_t time = clock();
double timeTaken;
//do the work


// check time elapsed
timeTaken = clock() - time;
// convert to seconds
printf("\n time taken is %lf
seconds", timeTaken/CLOCKS_PER_SEC);
```

The C library function

`clock_t clock(void)`

returns the number of clock ticks elapsed since the program was launched.

To get the number of seconds used by the CPU, you will need to divide by CLOCKS_PER_SEC

ADDING THIS TO linearSearch

```
56
57
58  int linearSearch (int arrA[], int size, int item) {
59
60      clock_t time = clock();
61      double timeTaken;
62
63      int i;
64      int position = -1;
65
66      for(i = 0; i < size && position == -1; i++) {
67          if (arrA[i] == item) {
68              position = i;
69          }
70      }
71
72      timeTaken = clock() - time; // check time elapsed after doing work
73      printf("\n time taken is %lf seconds", timeTaken/CLOCKS_PER_SEC);
74
75      return (position);
76  }
77
```

APPROACH 2 … COUNT "TIME STEPS"

The idea is to estimate the amount of work there is to do by summing up "time steps" in each statement.

The result is a function, *f*, which represents these time steps and is (usually) dependent on the data size input which is represented as some constant (e.g,. N) and then function is represented as *f(N)*

APPROACH 2 … COUNT “TIME STEPS” ctd.

- Each simple statement = 1 time step

Examples: declarations, initialisations, calculations, if conditions, function call, etc.

- `int i = 0;`
- `if (position == -1) {`
- `while (begSec <= endSec) {`

- Each memory access = 1 time step

Approach 2 ctd., ... Count "time steps"

- Loops, function execution, use of built in libraries are not simple statements and are usually dependent on the input size
- Loops:
  - In a simple case, any statement in a loop is "multiplied" by the number of iterations of a loop
  - The condition can be taken as 1 timestep (checked multiple times)
  - In for loops, where there are 2 actions per iteration (minimum) – can choose to count each action OR count everything in loop guard as 1 timestep (usual approach).
- Do we always know when the loop will stop?

TYPES OF ANALYSIS

For approach 2 and 3 there are 3 types of analysis that can be performed and that we are interested in:

Worst case: The function defined by the maximum number of steps taken in any instance of size n.

Best case: The function defined by the minimum number of steps taken in any instance of size n.

Average case: The function defined by the average number of steps over all instances of size n. (Assumes that the input is random)

WHICH TO USE?

- Mostly interested in average and worst case situations – best case situation is often not useful for analysis
- For time step analysis often focus on worst case analysis as it is important to know the upper limit on how poorly an algorithm can perform
- Our average case may be better than this in many situations or at least, can be no worse than this

Using Worst Case Analysis, how to Calculate the Time Steps and the Function which Represents the Worst Case Situation?

Generally ignore Input/Output statements (should be standard across solutions)

For each line of code/statement:

- List the time step count (cost)
- List the maximum number of times it is done (numTimes)
- Multiply cost by numTimes for each step
- Add up all the steps to get the function – most likely dependent on **N** the input size

WHAT IS N?

- N is the number of elements in the input
- Can be:
  - Size of an array (or list or tree or graph)
  - Number of words in a file
  - Number of elements to sort
  - Number of transactions to check for fraud patterns
  - Number of movies to display or recommend
  - Number of tweets to analyse to check if they are political ads.
  - etc.

COUNTING TIME STEPS

Can annotate code or create a table like the following:

Line	Cost	numTimes	cost*numTimes	Total

COUNTING TIME STEPS FOR LINEAR SEARCH

```
57
58  int linearSearch (int arrA[], int size, int item) {
59
60      int i;
61      int position = -1;
62
63      for(i = 0; i < size && position == -1; i++) {
64          if (arrA[i] == item) {
65              position = i;
66          }
67      }
68      return (position);
69  }
70
```

TIME STEP ANALYSIS: Linear Search

```
57
58  int linearSearch (int arrA[], int size, int item) {
59
60      int i;
61      int position = -1;
62
63      for(i = 0; i < size && position == -1; i++) {
64          if (arrA[i] == item) {
65              position = i;
66          }
67      }
68      return (position);
69  }
70
```

Let N = array size (number of items to search)

Line	Cost	(max) numTimes	cost*numTimes	Total
60	1	1	1	
61	1	1	1	
63	1	N+1	N+1	
64	1	N	N	
65	1	1	1	
68	1	1	1	
				2N + 5

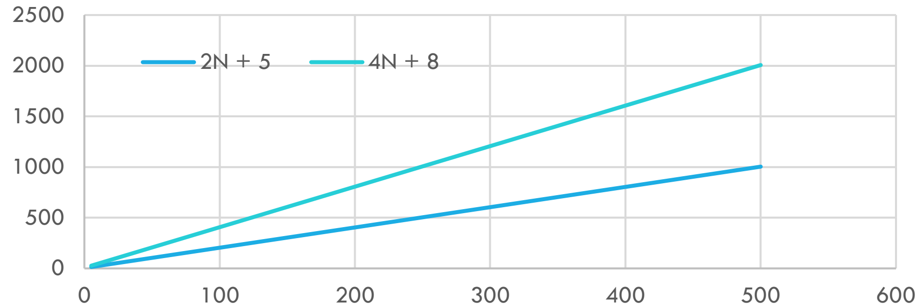
Alternative: counting individual statements in for loop guard as 3

Let N = array size (number of items to search)

```
57
58  int linearSearch (int arrA[], int size, int item) {
59
60      int i;
61      int position = -1;
62
63      for(i = 0; i < size && position == -1; i++) {
64          if (arrA[i] == item) {
65              position = i;
66          }
67      }
68      return (position);
69  }
70
```

Line	Cost	(max) numTimes	cost*numTimes	Total
60	1	1	1	
61	1	1	1	
63 (i = 0)	1	1	1	
63	3	N+1	3N+3	
65	1	N	N	
68	1	1	1	
60	1	1	1	
				4N + 8

COMPARING: 2N + 5 4N + 8



Note that although 4N + 8 timesteps will always take more time than 2N + 5, they are both “linear” or “grow in the same way”.

Generally some simplifications must be made when counting lines of code even if in reality we know that the following two lines of code will take different amounts of time:

```
int i;
for(i = 0; i < size && position == -1; ++i)
```

As many lines of code will not be equivalent it is usual to use a cost of 1, both for lines that might seem more simple, or more complex, than “normal”.

You Try ... TIME STEP ANALYSIS: Checking if an array is sorted

- Given an integer array, `arrA[]`, and its size (`size`) write an algorithm (code) in C to check if the array is sorted, returning 1 (or Boolean True) if the array is sorted and 0 (or Boolean False) if the array is not sorted. You may assume that we wish to search for sorted ***ascending*** order.
- Using the function written, perform a time step analysis to get a function representing the number of time steps needed (as a function of the size of the array):

Which fragment of code correctly identifies if an array has values in ascending sorted order?
A, B, C or D

COMPLETE THE TIME STEP ANALYSIS:

Let N = size

TIME STEP ANALYSIS BINARY SEARCH

```
51
52  int binarySearch(int arrA[], int size, int item)
53  {
54      printf("*** Data is in sorted order ***");
55
56      int begSec, endSec, mid, location;
57      begSec = 0;
58      endSec = size-1;
59      mid = int((begSec+endSec)/2);
60
61      location = -1;
62
63      while (begSec <= endSec && arrA[mid] != item){
64          if(item > arrA[mid]){
65              begSec = mid+1;
66          }
67          else if (item < arrA[mid]){
68              endSec = mid-1;
69          }
70
71          mid = int((begSec+endSec)/2);
72
73      }
74      if (arrA[mid]==item){
75          location = mid;
76      }
77      return(location);
78  }
```

TIME STEP ANALYSIS: Binary Search

Let N = size of array

Ignore `printf` at line 54 as it is not part of solution

```
51
52  int binarySearch(int arrA[], int size, int item)
53  {
54      printf("*** Data is in sorted order ***");
55
56      int begSec, endSec, mid, location;
57      begSec = 0;
58      endSec = size-1;
59      mid = int((begSec+endSec)/2);
60
61      location = -1;
62
63      while (begSec <= endSec && arrA[mid] != item){
64          if(item > arrA[mid]){
65              begSec = mid+1;
66          }
67          else if (item < arrA[mid]){
68              endSec = mid-1;
69          }
70
71          mid = int((begSec+endSec)/2);
72
73      }
74      if (arrA[mid]==item){
75          location = mid;
76      }
77      return(location);
```

Line	Cost	numTimes	Cost*Num Times	Total
56-59 & 61	1 for each (5 in total)	1 for each	5	
63	1	?		
64&65 **or** 64, 67&68	(pick larger) 3	**?**		
71	1	?		
74, 75, 77 **or** 74 and 77	(pick larger) 3	1	3	

How many times?

LOOKING AT LOOP GUARD:

while (begSec <= endSec && arrA[mid] != item)

Worst case analysis when item is not in array means arrA[mid] is never equal to item or item is found at the very last check

Assume N items in the array:

1st check: check for item and approx. ½ of items left to check ($\frac{N}{2}$ items)

2nd check: check for item and approx. ½ of items left to check (½ of $\frac{N}{2} = \frac{N}{4} = \frac{N}{2^2}$)

3nd check: check for item and approx. ½ of items left to check (½ of $\frac{N}{4} = \frac{N}{8} = \frac{N}{2^3}$)

4th check: check for item and approx. ½ of items left to check (½ of $\frac{N}{8} = \frac{N}{16} = \frac{N}{2^4}$)

etc.

kth check: check for item and approx. $\frac{N}{2^k}$ items left to check

Note: At each stage *reducing* by a power of 2 (not linear)

SOLVING .... $\frac{N}{2^k} = 1$

How many checks will be needed to have one value left to check?

$\frac{N}{2^k} = 1$

$N = 2^k$

- Multiplying both sides by $\log_2$ to get:

$\log_2 N = \log_2 2^k$

$\log_2 N = k$

Therefore, in the worst case, need $\log_2 N$ checks to find item or to know that it isn't there

WILL "STEPS OF POWER OF 2" ALWAYS GIVE US LOG_2 BEHAVIOUR?

Consider this loop guard:

```
for (i = 1; i < n; i = i * 2) {
```

for (i = 1; i < n; i = i * 2) {		
n	**Values of i:**	**Number of iterations**
10	1, 2, 4, 8,	4
20	1, 2, 4, 8, 16	5
30	1, 2, 4, 8, 16	5
40	1, 2, 4, 8, 16, 32	6
50	1, 2, 4, 8, 16, 32	6
60	1, 2, 4, 8, 16, 32	6
70	1, 2, 4, 8, 16, 32, 64	7
etc.		

Consider this loop guard:

```
for (i = n; i > 0; i = int (i / 2) ) {
```

for (i = n; i > 0; i = int (i / 2)) {			
n	**Values of i:**	**Number of iterations**	**$\log_2(n)$**
10	10, 5, 2, 1	4	3.32
20	20, 10, 5, 2, 1	5	4.32
30	30, 15, 7, 3, 1	5	4.91
40	40, 20, 10, 5, 2, 1	6	5.32
50	50, 25, 14, 7, 3, 1	6	5.64
60	60, 30, 15, 7, 3, 1	6	5.91
70	70, 35, 17, 8, 4, 2, 1	7	6.13
80			
....			
10000		approx $\log_2$ (n)	13.28

BACK TO: Binary Search time step analysis:

Let N = size of array

Ignore `printf` at line 54

Ignore ±1 iterations in loop

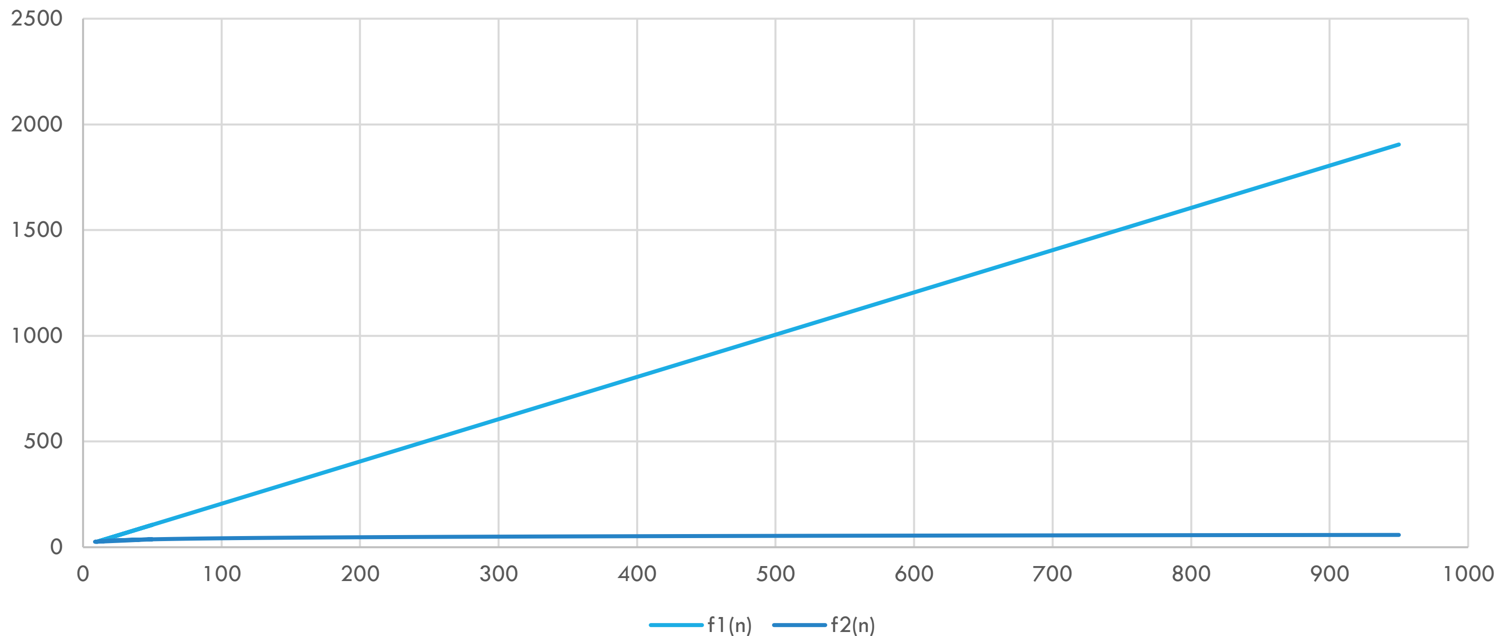
```
51
52  int binarySearch(int arrA[], int size, int item)
53  {
54      printf("*** Data is in sorted order ***");
55
56      int begSec, endSec, mid, location;
57      begSec = 0;
58      endSec = size-1;
59      mid = int((begSec+endSec)/2);
60
61      location = -1;
62
63      while (begSec <= endSec && arrA[mid] != item){
64          if(item > arrA[mid]){
65              begSec = mid+1;
66          }
67          else if (item < arrA[mid]){
68              endSec = mid-1;
69          }
70
71          mid = int((begSec+endSec)/2);
72
73      }
74      if (arrA[mid]==item){
75          location = mid;
76      }
77      return(location);
```

Line	Cost	numTimes	Cost*Num Times	Total
56-59 & 61	5	1	5	
63	1	log_2(N)	log_2(N)	
64&65 **or** 64, 67&68	(pick larger) 3	log_2(N)	$3log_2$(N)	
71	1	log_2(N)	log_2(N)	
74, 75, 77 **or** 74 and 77	(pick larger) 3	1	3	
				$5log_2$(N) + 8

HOW DOES THIS COMPARE TO LINEAR SEARCH?

Linear Search: f(N) = 2N + 5

Binary Search: f(N) = $5log_2$(N) + 8



APPROACH 2 … COUNT "TIME STEPS" SUMMARY

- *Generally,*
  - Do not count I/O steps
  - Gives a good approximation of the actual run time
  - But, requires effort in counting
  - Often we want a more generic way to compare algorithms … especially across different programming languages … without getting distracted with coefficients, additive and multiplicative constants of n (the input size)

APPROACH 3: RATE OF GROWTH

Given that n will have different values for each run it is usually the *rate of growth* or increase of f(n) that we want to analyse

For example, if timestep analysis gives us: $n^3 + 2n + 1234$ it is only as n gets larger that f(n) starts to get very large

n	**n^3**	**2n**	**1234**	**fn = n^3+2n+1234**
10	1000	20	1234	2254
100	1000000	200	1234	1001434
1000	1E+09	2000	1234	1000003234
10000	1E+12	20000	1234	1E+12
1000000	1E+18	2000000	1234	1E+18

SUMMARY

Algorithm analysis is a fundamental aspect of Algorithms

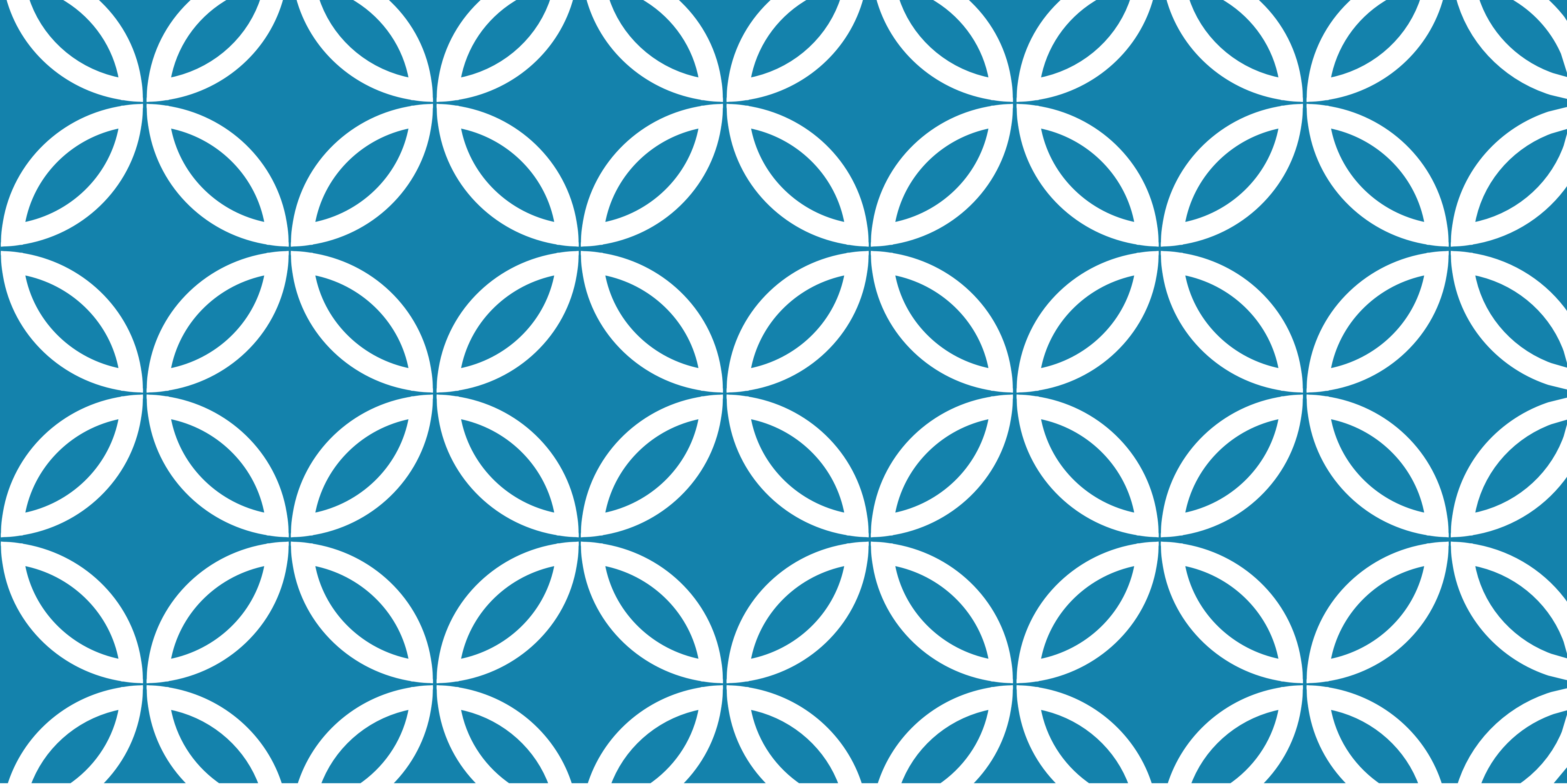
Important to know and understand the different approaches for Algorithm Analysis and how to analyse algorithms using a number of approaches

We will consider 3 approaches:

Actual run time

Time step analysis (generally worst case)

Big O Analysis (rate of growth) – next lecture



CT102: ALGORITHMS AND INFORMATION SYSTEMS

Topic:

Algorithms and Algorithm Analysis

RECALL:
FOCUS OF ALGORITHM ANALYSIS ...

The analysis and comparison of algorithms with respect to resources used, that is:

- Space (memory)
- Time (to run)

Important Question for us: When two programs solve the same problem differently why is one better than the other?

RECALL:
3 APPROACHES USED FOR COMPARISON

1. Actual Time: Code and Run and track time
2. Estimate time: Count time “steps” in code (without running)
3. Estimate rate of growth of time used: Use Big-O notation (and others) for large input sizes

COMPLETE THE TIME STEP ANALYSIS:

Let N = size

```
10
11  bool isSorted(int arrA[], int size)
12  {
13      int i;
14      bool sorted = true;
15
16      for(i = 0; i < size-1 && sorted; i++){
17          if(arrA[i] > arrA[i+1]){
18              sorted = false;
19          }
20      }
21      return(sorted);
22  }
23
24
```

Line	Cost	(max) numTimes	cost*numTimes	Total
13	1	1	1	
14	1	1	1	
16	1	N	N	
17	1	N-1	N-1	
18	1	1	1	
21	1	1	1	
				2N+3

TIME STEP ANALYSIS BINARY SEARCH

```
51
52 int binarySearch(int arrA[], int size, int item)
53 {
54     printf("*** Data is in sorted order ***");
55
56     int begSec, endSec, mid, location;
57     begSec = 0;
58     endSec = size-1;
59     mid = int((begSec+endSec)/2);
60
61     location = -1;
62
63     while (begSec <= endSec && arrA[mid] != item){
64         if(item > arrA[mid]){
65             begSec = mid+1;
66         }
67         else if (item < arrA[mid]){
68             endSec = mid-1;
69         }
70
71         mid = int((begSec+endSec)/2);
72
73     }
74     if (arrA[mid]==item){
75         location = mid;
76     }
77     return(location);
78 }
```

TIME STEP ANALYSIS: Binary Search

Let N = size of array

Ignore `printf` at line 54

as it is not part of solution

```c
52 int binarySearch(int arrA[], int size, int item)
53 {
54     printf("*** Data is in sorted order ***");
55 
56     int begSec, endSec, mid, location;
57     begSec = 0;
58     endSec = size-1;
59     mid = int((begSec+endSec)/2);
60 
61     location = -1;
62 
63     while (begSec <= endSec && arrA[mid] != item){
64         if(item > arrA[mid]){
65             begSec = mid+1;
66         }
67         else if (item < arrA[mid]){
68             endSec = mid-1;
69         }
70 
71         mid = int((begSec+endSec)/2);
72 
73     }
74     if (arrA[mid]==item){
75         location = mid;
76     }
77     return(location);
78 }
```

Line	Cost	numTimes	Cost*Num Times	Total
56-59 & 61	1 for each (5 in total)	1 for each	5	
63	1	?		
64&65 **or** 64, 67&68	(pick larger) 3	?		
71	1	?		
74, 75, 77 **or** 74 and 77	(pick larger) 3	1	3	

How many times?

LOOKING AT LOOP GUARD:

while (begSec <= endSec && arrA[mid] != item)

Worst case analysis when item is not in array means arrA[mid] is never equal to item or item is found at the very last check

Assume N items in the array:

1st check: check for item and approx. ½ of items left to check ($\frac{N}{2}$ items)

2nd check: check for item and approx. ½ of items left to check (½ of $\frac{N}{2} = \frac{N}{4} = \frac{N}{2^2}$)

3nd check: check for item and approx. ½ of items left to check (½ of $\frac{N}{4} = \frac{N}{8} = \frac{N}{2^3}$)

4th check: check for item and approx. ½ of items left to check (½ of $\frac{N}{8} = \frac{N}{16} = \frac{N}{2^4}$)

etc.

kth check: check for item and approx. $\frac{N}{2^k}$ items left to check

Note: At each stage *reducing* by a power of 2 (not linear)

SOLVING .... $\frac{N}{2^k} = 1$

How many checks will be needed to have one value left to check?

$\frac{N}{2^k} = 1$

$N = 2^k$

- Multiplying both sides by $\log_2$ to get:

$\log_2 N = \log_2 2^k$

$\log_2 N = k$

Therefore, in the worst case, need $\log_2 N$ checks to find item or to know that it isn't there

WILL "STEPS OF POWER OF 2" ALWAYS GIVE US LOG_2 BEHAVIOUR?

Consider this loop guard:

```
for (i = 1; i < n; i = i * 2) {
```

for (i = 1; i < n; i = i * 2) {		
n	**Values of i:**	**Number of iterations**
10	1, 2, 4, 8,	4
20	1, 2, 4, 8, 16	5
30	1, 2, 4, 8, 16	5
40	1, 2, 4, 8, 16, 32	6
50	1, 2, 4, 8, 16, 32	6
60	1, 2, 4, 8, 16, 32	6
70	1, 2, 4, 8, 16, 32, 64	7
etc.		

Consider this loop guard:

```
for (i = n; i > 0; i = int (i / 2) ) {
```

for (i = n; i > 0; i = int (i / 2)) {			
n	**Values of i:**	**Number of iterations**	**$\log_2(n)$**
10	10, 5, 2, 1	4	3.32
20	20, 10, 5, 2, 1	5	4.32
30	30, 15, 7, 3, 1	5	4.91
40	40, 20, 10, 5, 2, 1	6	5.32
50	50, 25, 14, 7, 3, 1	6	5.64
60	60, 30, 15, 7, 3, 1	6	5.91
70	70, 35, 17, 8, 4, 2, 1	7	6.13
80			
....			
10000		approx $\log_2$ (n)	13.28

BACK TO: Binary Search time step analysis:

Let N = size of array

Ignore `printf` at line 54

Ignore ±1 iterations in loop

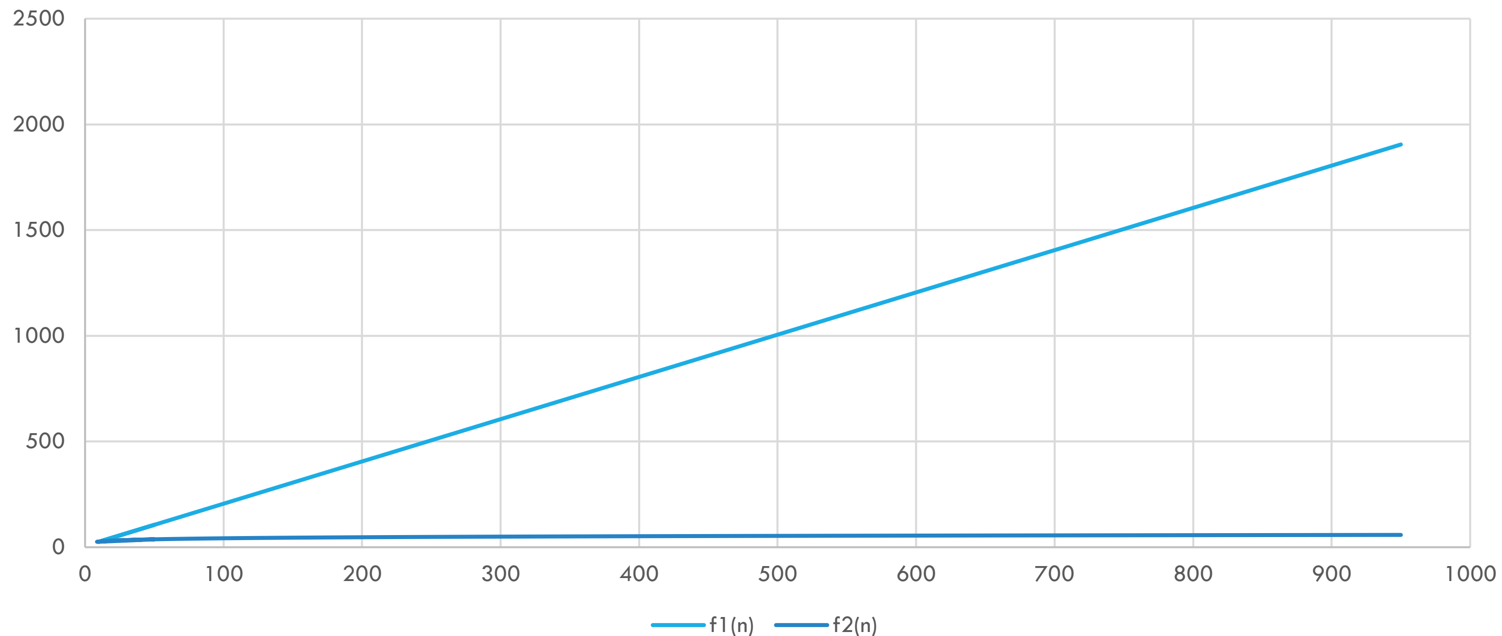
```
51
52 int binarySearch(int arrA[], int size, int item)
53 {
54     printf("*** Data is in sorted order ***");
55
56     int begSec, endSec, mid, location;
57     begSec = 0;
58     endSec = size-1;
59     mid = int((begSec+endSec)/2);
60
61     location = -1;
62
63     while (begSec <= endSec && arrA[mid] != item){
64         if(item > arrA[mid]){
65             begSec = mid+1;
66         }
67         else if (item < arrA[mid]){
68             endSec = mid-1;
69         }
70
71         mid = int((begSec+endSec)/2);
72
73     }
74     if (arrA[mid]==item){
75         location = mid;
76     }
77     return(location);
```

Line	Cost	numTimes	Cost*Num Times	Total
56-59 & 61	5	1	5	
63	1	log_2(N)	log_2(N)	
64&65 **or** 64, 67&68	(pick larger) 3	log_2(N)	$3log_2$(N)	
71	1	log_2(N)	log_2(N)	
74, 75, 77 **or** 74 and 77	(pick larger) 3	1	3	
				$5log_2$(N) + 8

HOW DOES THIS COMPARE TO LINEAR SEARCH?

Linear Search: f(N) = 2N + 5

Binary Search: f(N) = $5log_2$(N) + 8



APPROACH 2 … COUNT "TIME STEPS" SUMMARY

- *Generally,*
  - Do not count I/O steps
  - Gives a good approximation of the actual run time
  - But, requires effort in counting
  - Often we want a more generic way to compare algorithms … especially across different programming languages … without getting distracted with coefficients, additive and multiplicative constants of n (the input size)

APPROACH 3: RATE OF GROWTH

Given that n will have different values for each run it is usually the *rate of growth* or increase of f(n) that we want to analyse

For example, if timestep analysis gives us:

$F(n) = n^3 + 2n + 1234$

it is only as n gets larger that f(n) starts to get very large

n	n^3	2n	1234	$f(n) = n^3+2n+1234$
10	1000	20	1234	2254
100	1000000	200	1234	1001434
1000	1E+09	2000	1234	1000003234
10000	1E+12	20000	1234	1E+12
1000000	1E+18	2000000	1234	1E+18

COMMONLY USED RATE OF GROWTH FUNCTIONS

$f(n)$ is usually compared with some standard mathematical functions, such as:

- $\log_2 n$
- n
- $n \log_2 n$
- n^2
- n^3
- 2^n

WHAT IS THE RATE OF GROWTH OF THESE STANDARD FUNCTIONS?

Look at different values of n to see difference ….

n	log2 n	n log2 n	n^2	n^3	2^n
5	2.32	11.60	25	125	32
10	3.32	33.22	100	1000	1024
100	6.64	664.39	10000	1000000	1.27E+30
1000	9.97	9965.78	1000000	1E+09	1.1E+301
10000	13.29	132877.1	1E+08	1E+12	#NUM!

STANDARD FUNCTIONS AND ALGORITHM EXAMPLES

Class	Name	Algorithm Example & Notes
1	Constant	No dependence on n
log n	Logarithmic	Binary search
n	Linear	Linear search
n log n	Super linear	Mergesort, Quicksort
n^2	Quadratic	Typically an algorithm with a nested loop – with both loops iterating over n items. Selection, Insertion and Bubble Sort
n^3	Cubic	Typically an algorithm with 3 nested loops – with all loops iterating over n items
2^n	Exponential	Some recursive solutions and pathfinding algorithms
n!	Factorial	Unusable except for very small n

BIG-O NOTATION

Big-O notation gives a measure of rate of growth in terms of upper and lower bounds in comparison to some standard functions.

N.B. Ignores coefficients and additive and multiplicative constants.
For example:

n and $2n$ are considered the same.

n and $n + 500$ are considered the same.

n^2 and $5n^2$ are considered the same.

MORE FORMALLY:

Given f(n) for some algorithm:

f(n) is O(g(n)) means that it is always possible to find some *k* such that:

f(n) <= k g(n) for *n* >= n_0 (*large enough n*)

k g(n) is an **upper bound** on f(n)

EXAMPLES:

What is big O (upper bound) for the following functions

f(n) = 3n + 8

O(f(n)) is n: O(n)

$f(n) = \frac{n^2}{2} + 10n + 5$

O(f(n)) is n^2 $O(n^2)$

f(n) = 2103

O(f(n)) is 1: O(1)

O, Ω, Θ

We can also define similar functions for the lower bound (omega Ω) and lower and upper bounds (theta Θ).

Generally concentrate on the upper bound O as:

- knowing the lower bound (Ω) is of no practical importance (best case).
- although knowing Θ gives a more exact definition of the behaviour (on average) it can be more difficult to calculate.

DOMINANCE RELATIONS

Therefore, Big O notation can be used to describe the growth rate for any particular algorithm where the coefficients, additive and multiplicative constants of the actual *f(n)* are of very little consequence - what is important is to understand the ordering:

$n! >> 2^n >> n^3 >> n^2 >> n \log n >> n >> \log n >> 1$

Note on POLYNOMIALS:

Any algorithm whose time complexity is $O(n^x)$ when $x > 1$ is said to be of polynomial time order.

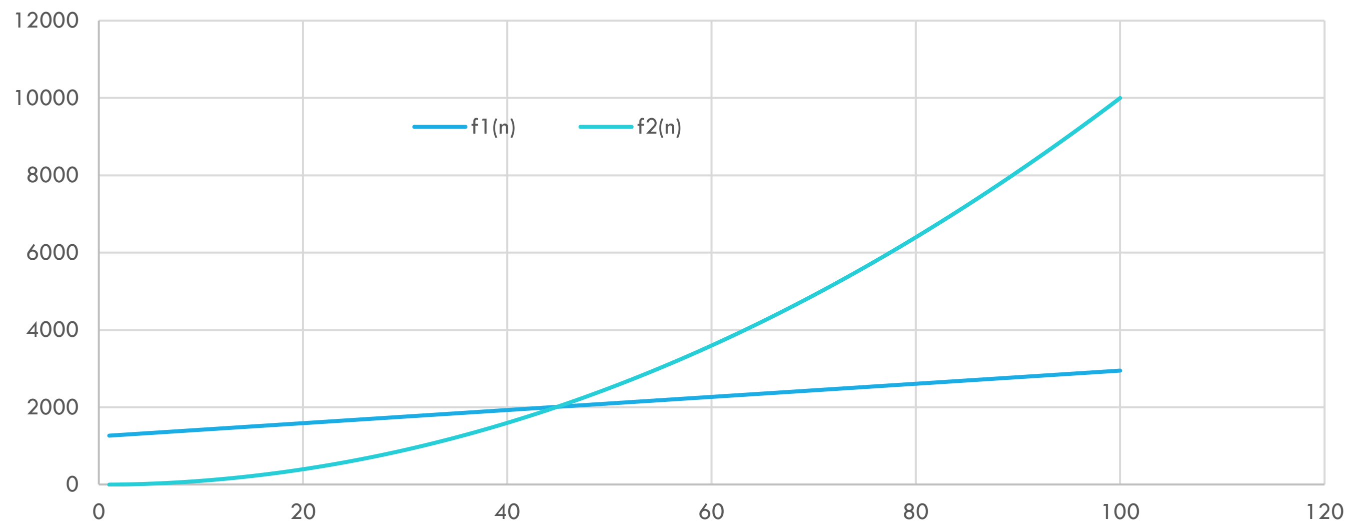
Two points on polynomials:

- Grow rapidly - for all practical problems: $O(n^3)$ algorithm is going to be much worse than an $O(n)$ algorithm so in general a linear algorithm ($O(n)$) is better
- However, for small *n*, polynomial can be better.

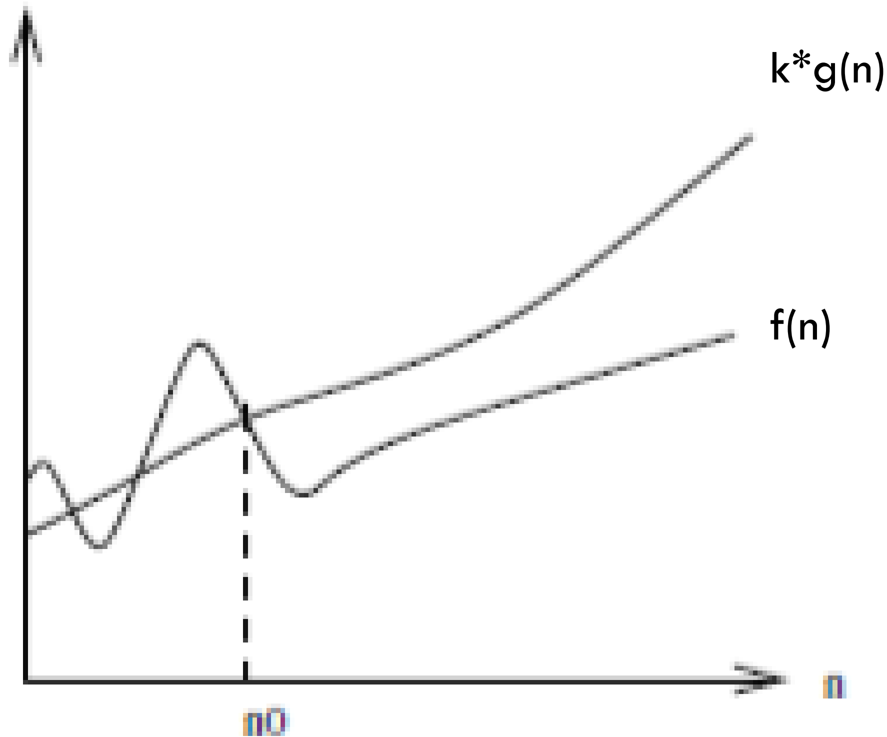
FOR EXAMPLE:

$f_1(n) = 17n + 1250$ => $f(n)$ is $O(n)$

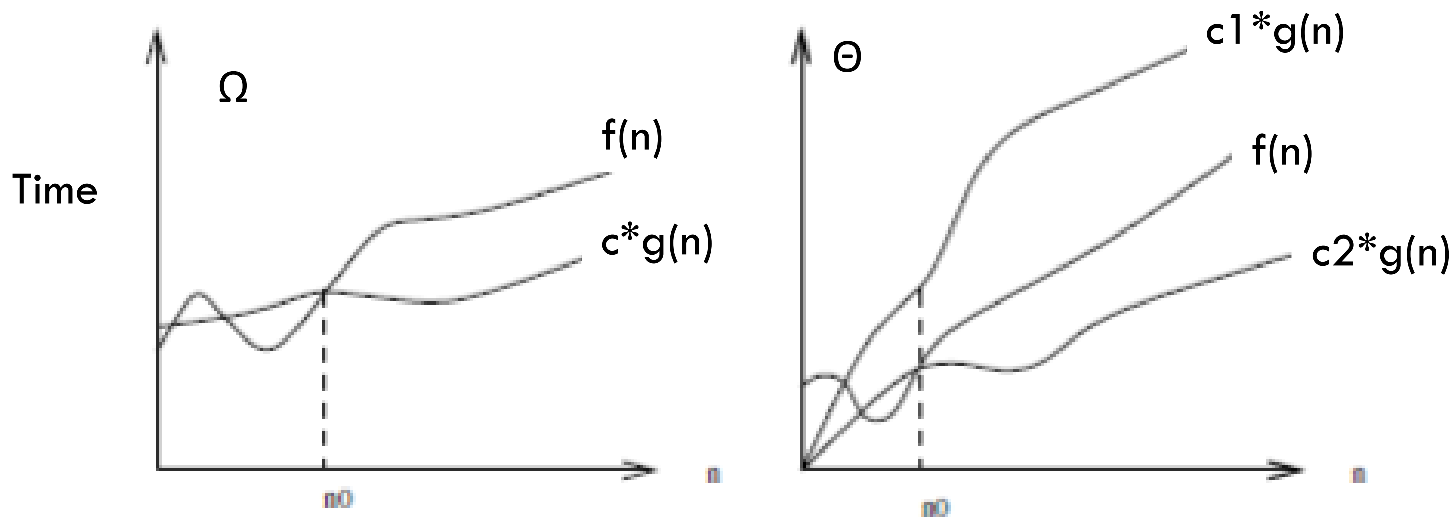
$f_2(n) = n^2 + 1$ => $f(n)$ is $O(n^2)$



But what about the constants?
i.e., For Big O, what is k and n_0 *?*



Aside: can also define similar constants for lower bound (Ω), and average (Θ)



EXAMPLES

What is Big O (upper bound) for the following functions

(listing a value for k and n_0)

1. f(n) = 3n + 8:

f(n) <= 4n for n >= 8.

O(f(n)) is O(n)   k = 4 n0 = 8

2. $f(n) = n^4 + 100n^2 + 50$

f(n) <= ?

O(f(n)) is ?   k is? n0 is ?

n	f(n)=3n+8	4n
1	11	4
2	14	8
3	17	12
4	20	16
5	23	20
6	26	24
7	29	28
8	32	32
9	35	36
10	38	40
11	41	44
12	44	48
13	47	52

List appropriate values for k and $n0$ for the function $f(n) = n^4 + 100n^2 + 50$

n	$f(n)= n^4 + 100n^2 + 50$	$2n^4$
1	151	2
2	466	32
3	1031	162
4	1906	512
5	3175	1250
6	4946	2592
7	7351	4802
8	10546	8192
9	14711	13122
10	20050	20000
11	26791	29282
12	35186	41472

O(f(n)) is O($\mathbf{n^4}$)

k is 2

n0 is 11

NOTE:

For any f(n) we can, picking some g(n) from f(n), always find some value for k and n_0

Therefore, unless we need it for other analysis, we do not need to worry about finding the values for k and n_0 (i.e., we know that it exists but we don't need to find it)

GENERAL STEPS TO FIND BIG-O RUNTIME

- Understand what the input is and represent it as **n**
- Find the maximum (worst case) number of time steps in the algorithm in terms of **n** (can ignore statements not dependent on n) representing as function f(n).
- Eliminate all but the highest order terms in f(n).
- Remove all the constant and multiplicative factors in f(n).

CLASS QUESTION:

Go to **www.menti.com** and use the code **4299 6515**

Given the following function which sums all the values in an integer array (`arrA[]`) with a given size. Perform a time step analysis of the function `_sumArray()` What is Big O (upper bound) for the function? Enter your answer in the menti room given.

```
11
12  int sumIntArray(int[], int);
13
14  int sumIntArray(int arrA[], int size) {
15
16      int i, sum;
17      sum = 0;
18      for(i = 0; i < size; i++) {
19          sum = sum + arrA[i];
20      }
21
22      return(sum);
23  }
24
```

Line	Cost	(max) numTimes	cost* numTimes	Total
16				
17				
18				
19				
22				
				f(n) =

Consider another problem that you have already seen a solution for (in CT103) … Bubble Sort

Inputs: An array arrA with given size with *n* distinct integers in unsorted order

Outputs: An array arrA with *n* distinct integers in increasing sorted order

Process:

- Get one element in correct position: "Bubble" largest element up to correct position by traversing array and comparing - and moving if necessary - adjacent elements.
- Keep doing this for all *n* items

APPROACH (“BUBBLING” LARGEST)

```
for(k = 0; k < size; k++){
    for(i = 0; i < size - 1; i++){
      if(arrA[i] > arrA[i + 1]){
            //out of order so swap values
      } //end if
   } //end inner for (i loop)
} //end outer for (k loop)
```

APPROACH ("BUBBLING" LARGEST)

```
for(k = 0; k < size; k++){
    for(i = 0; i < size - 1 - k; i++){
      if(arrA[i] > arrA[i + 1]){
            //out of order so swap values
      } //end if
  } //end inner for (i loop)
} //end outer for (k loop)
```

void bubbleSort(int[], int);

See Blackboard for function code

```
18  //Bubble Sort
19  void bubbleSort(int arrA[], int size)
20  {
21      int i, k, temp;
22
23      for (k = 0; k < size; k++){
24          for (i = 0; i < size - 1 - k; i++){
25              if (arrA[i] > arrA[i+1]){
26                  //out of order so swap
27                  temp = arrA[i];
28                  arrA[i]=arrA[i+1];
29                  arrA[i+1]=temp;
30              }
31          } //end inner i for
32      } //end outer k for
33  }
34
```

HOW TO CALL FUNCTION `bubbleSort()` WITH SAMPLE DATA?

```
13
14      int arrA[] = {22, 18, 16, 14, 2};       // array of integers
15      int size = 5;
16
17      bubbleSort(arrA, size);
18
```

ALGORITHM ANALYSIS

```
17
18    //Bubble Sort
19    void bubbleSort(int arrA[], int size)
20    {
21        int i, k, temp;
22
23        for (k = 0; k < size; k++){
24            for (i = 0; i < size - 1 - k; i++){
25                if (arrA[i] > arrA[i+1]){
26                    //out of order so swap
27                    temp = arrA[i];
28                    arrA[i]=arrA[i+1];
29                    arrA[i+1]=temp;
30                }
31            } //end inner i for
32        } //end outer k for
33    }
34
35
```

When should we start and end clock if counting actual time?

What are the key operations?

What is the worst case situation?

What is the best and worst case situation for bubble sort?

Go to **www.menti.com** and use the code **4299 6515**

RUNNING CODE:

Modifying code to count:

- How many comparisons are done
- How many swaps are done

Use two counters:

```
numSwaps
numCmprs
```

Adding in counts and clock()

Don't forget: #include "time.h"

```
17
18   //Bubble Sort
19   void bubbleSort(int arrA[], int size)
20   {
21       int i, k, temp;
22       int numSwaps = 0;
23       int numCmprs = 0;
24
25       clock_t t;
26       t = clock();
27
28       for (k = 0; k < size; k++) {
29           for (i = 0; i < size - 1 - k; i++) {
30
31               if (arrA[i] > arrA[i + 1]){
32                   //out of order so swap
33                   ++numSwaps;
34                   temp = arrA[i];
35                   arrA[i]=arrA[i + 1];
36                   arrA[i + 1]=temp;
37               }
38               ++numCmprs;
39           } //end inner i for
40       } //end outer k for
41
42       t = clock() - t;
43       double time_taken = ((double)t)/CLOCKS_PER_SEC; // in seconds
44       printf("\n In Bubble Sort: \n time taken is %f; \n number of swaps is %d; \n number of comparisons is %d", time_taken, numSwaps, numCmprs);
45   }
46
```

TIME STEP ANALYSIS

Let N = size

Assume worst case

```
17
18  //Bubble Sort
19  void bubbleSort(int arrA[], int size)
20  {
21      int i, k, temp;
22
23      for (k = 0; k < size; k++){
24          for (i = 0; i < size - 1 - k; i++){
25              if (arrA[i] > arrA[i+1]){
26                  //out of order so swap
27                  temp = arrA[i];
28                  arrA[i]=arrA[i+1];
29                  arrA[i+1]=temp;
30              }
31          } //end inner i for
32      } //end outer k for
33  }
34
35
```

Line	Cost	numTimes	Cost*numTimes	Total
21	1	1	1	
23	1	N+1	N+1	
24	1	?		
25	1	?		
27, 28, 29 *worst case*	1+1+1	?		
				f(N) =

Looking more closely at line 24:

```
22
23      for (k = 0; k < size; k++){
24          for (i = 0; i < size - 1 - k; i++){
25              if (arrA[i] > arrA[i+1]){
26                  //out of order so swap
```

1st iteration of outer loop (k=0) numTimes of line 24 = N (checking 0 to N-1-0)

2nd iteration of outer loop (k=1) numTimes of line 24 = N-1 (checking 0 to N-1-1)

3rd iteration of outer loop (k=2) numTimes of line 24 = N-2 (checking 0 to N-1-2)

… etc.

2nd last iteration of outer loop (k=N-2) numTimes of line 24 = 2 (0 to N-1-(N-2))

Last iteration of outer loop (k=N-1) numTimes of line 24 = 1 (0 to N-1-(N-1))

So adding them all up:

N + N-1+ N-2 + N-3 + ….+ 2 + 1 = sum of N integers from 1 to N

Formula for sum of N integers is $\frac{N(N+1)}{2} = \frac{N^2+N}{2}$

e.g., sum of integers from 1 to 5 (1+2+3+4+5) = $\frac{5^2+5}{2} = \frac{30}{2} = 15$

Looking more closely at line 25:

```
22
23      for (k = 0; k < size; k++){
24          for (i = 0; i < size - 1 - k; i++){
25              if (arrA[i] > arrA[i+1]){
26                  //out of order so swap
```

Line 25 (if statement) is carried out one less than line 24 so sum is from 1 to N-1:

Substituting N-1 for N in $\frac{N(N+1)}{2}$ gives: $\frac{N-1(N-1+1)}{2} = \frac{N^2-N}{2}$

i.e.., if N = 5 at line 24, then line 24 is carried out 25 times (in total) but line 25 is carried out : $\frac{5^2-5}{2} = \frac{20}{2} =$ 10 times in total

EQUIVALENTLY …

We have a general formula for the sum of N consecutive integers, starting at any integer:

Sum = N(firstNum + lastNum)/2

e.g. sum of integers from 2 to 10 (9 integers)

- 9(2+10) / 2 = (9*12)/2 = 54

From previous analysis of line 25: summing from 1 to N-1 =

N-1(1 + N-1)/2

TIME STEP ANALYSIS

Let N = size

Assume worst case

```
17
18  //Bubble Sort
19  void bubbleSort(int arrA[], int size)
20  {
21      int i, k, temp;
22
23      for (k = 0; k < size; k++){
24          for (i = 0; i < size - 1 - k; i++){
25              if (arrA[i] > arrA[i+1]){
26                  //out of order so swap
27                  temp = arrA[i];
28                  arrA[i]=arrA[i+1];
29                  arrA[i+1]=temp;
30              }
31          } //end inner i for
32      } //end outer k for
33  }
34
35
```

Line	Cost	numTimes	Cost*numTimes	Total
21	1	1	1	
23	1	N+1	N+1	
24	1	$\frac{N(N+1)}{2}$	$\frac{N^2+N}{2}$	
25	1	$\frac{N^2-N}{2}$	$\frac{N^2-N}{2}$	
27, 28, 29 *worst case*	1+1+1	$\frac{N^2-N}{2}$	$3(\frac{N^2-N}{2})$	
				$f(N) = \frac{5N^2-N}{2} + 2$

BIG-O ANALYSIS

As f(N) $= \frac{5N^2 - N}{2} + 2$ then we say Bubble sort is O(N^2) where N is the number of values in the array.

There are also a number of other sorting algorithms which have O(N^2) time complexity and we will consider these next.

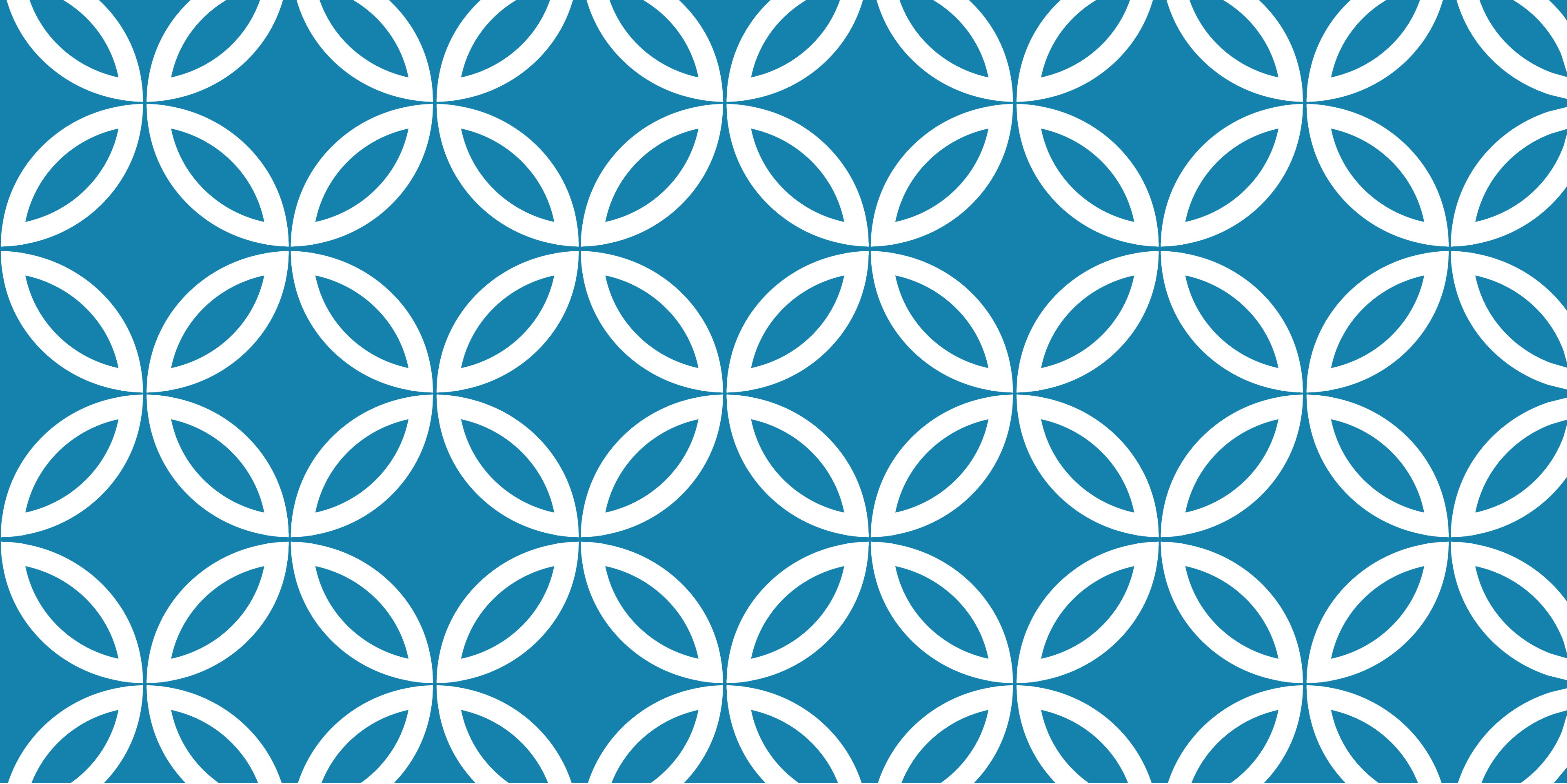
However, we will see that Bubble sort is one of the worst sorting algorithms we can use for general data.

HOW TO TEST THIS ON LARGER DATA?

We will start working with data from files once you have it covered in CT103

SUMMARY

- The complexity of an algorithm M is the function f(n) which gives the running time and/or storage space requirement of the algorithm in terms of the size *n* of the input data
- f(n) usually refers to the running time of the algorithm – and can be found by time step analysis - by counting time steps in a worst case scenario
- It is the **growth** of f(n) as n increases that is often interest – and we often concentrate on Big O notation
- In all cases, we can distinguish between best, average and worst case analysis but we consider worst case mostly.
- Although Bubble Sort is never a good choice for a sorting algorithm we have covered some important aspects of sorting in this lecture, particularly with respect to the analysis of the algorithm and considering worst case situations.
- Next we will consider two more O(N^2) sorting algorithms Insertion Sort and Selection Sort and also start to work with larger arrays



CT102: ALGORITHMS AND INFORMATION SYSTEMS

More $O(n^2)$ Sorting and Analysis

Recall: Bubble Sort code

`void bubbleSort(int[], int);`

```
17
18  //Bubble Sort
19  void bubbleSort(int arrA[], int size)
20  {
21      int i, k, temp;
22
23      for (k = 0; k < size; k++){
24          for (i = 0; i < size - 1 - k; i++){
25              if (arrA[i] > arrA[i+1]){
26                  //out of order so swap
27                  temp = arrA[i];
28                  arrA[i]=arrA[i+1];
29                  arrA[i+1]=temp;
30              }
31          } //end inner i for
32      } //end outer k for
33  }
34
35
```

TIME STEP ANALYSIS

Let N = size

Assume worst case

```
17
18  //Bubble Sort
19  void bubbleSort(int arrA[], int size)
20  {
21      int i, k, temp;
22
23      for (k = 0; k < size; k++){
24          for (i = 0; i < size - 1 - k; i++){
25              if (arrA[i] > arrA[i+1]){
26                  //out of order so swap
27                  temp = arrA[i];
28                  arrA[i]=arrA[i+1];
29                  arrA[i+1]=temp;
30              }
31          } //end inner i for
32      } //end outer k for
33  }
34
35
```

Line	Cost	numTimes	Cost*numTimes	Total
21	1	1	1	
23	1	N+1	N+1	
24	1	?		
25	1	?		
27, 28, 29 *worst case*	1+1+1	?	3 *	
				f(N) =

Looking more closely at line 24:

```
22
23    for (k = 0; k < size; k++){
24        for (i = 0; i < size - 1 - k; i++){
25            if (arrA[i] > arrA[i+1]){
26                //out of order so swap
```

1st iteration of outer loop (k=0) numTimes of line 24 = N (checking 0 to N-1-0)

2nd iteration of outer loop (k=1) numTimes of line 24 = N-1 (checking 0 to N-1-1)

3rd iteration of outer loop (k=2) numTimes of line 24 = N-2 (checking 0 to N-1-2)

… etc.

2nd last iteration of outer loop (k=N-2) numTimes of line 24 = 2 (0 to N-1-(N-2))

Last iteration of outer loop (k=N-1) numTimes of line 24 = 1 (0 to N-1-(N-1))

So adding them all up:

N + N-1+ N-2 + N-3 + ….+ 2 + 1 = sum of N integers from 1 to N

Formula for sum of N integers from 1 to N is $\frac{N(N+1)}{2} = \frac{N^2+N}{2}$

e.g., sum of integers from 1 to 5 (1+2+3+4+5) = $\frac{5^2+5}{2} = \frac{30}{2} = 15$

Looking more closely at line 25:

```
22
23      for (k = 0; k < size; k++){
24          for (i = 0; i < size - 1 - k; i++){
25              if (arrA[i] > arrA[i+1]){
26                  //out of order so swap
```

Line 25 (if statement) is carried out one less than line 24 so sum is from 1 to N-1:

Substituting N-1 for N in $\frac{N(N+1)}{2}$ gives: $\frac{N-1(N-1+1)}{2} = \frac{N^2-N}{2}$

i.e., if N = 5 at line 24, then line 24 is carried out 25 times (in total) but line 25 is carried out : $\frac{5^2-5}{2} = \frac{20}{2} = 10$ times in total

EQUIVALENTLY …

```
22
23      for (k = 0; k < size; k++){
24          for (i = 0; i < size - 1 - k; i++){
25              if (arrA[i] > arrA[i+1]){
26                  //out of order so swap
```

We have a general formula for the sum of N consecutive integers, starting at any integer:

$$\frac{N*(firstNum+lastNum)}{2}$$

e.g. sum of integers from 2 to 10 (9 integers)

- $\frac{9*(2+10)}{2} = \frac{9*12}{2} = 54$

From previous analysis of line 25: summing from 1 to N-1

$$= \frac{N-1(1+N-1)}{2} = \frac{N^2 - N}{2}$$

TIME STEP ANALYSIS

Let N = size

Assume worst case

```
17
18  //Bubble Sort
19  void bubbleSort(int arrA[], int size)
20  {
21      int i, k, temp;
22
23      for (k = 0; k < size; k++){
24          for (i = 0; i < size - 1 - k; i++){
25              if (arrA[i] > arrA[i+1]){
26                  //out of order so swap
27                  temp = arrA[i];
28                  arrA[i]=arrA[i+1];
29                  arrA[i+1]=temp;
30              }
31          } //end inner i for
32      } //end outer k for
33  }
34
35
```

Line	Cost	numTimes	Cost*numTimes	Total
21	1	1	1	
23	1	N+1	N+1	
24	1	$\frac{N(N+1)}{2}$	$\frac{N^2+N}{2}$	
25	1	$\frac{N^2-N}{2}$	$\frac{N^2-N}{2}$	
27, 28, 29 *worst case*	1+1+1	$\frac{N^2-N}{2}$	$3(\frac{N^2-N}{2})$	
				$f(N) = \frac{5N^2-N}{2} + 2$

BIG-O ANALYSIS

As $f(N) = \frac{5N^2 - N}{2} + 2$ then we say Bubble sort is O(N^2) where N is the number of values in the array.

There are also a number of other sorting algorithms which have O(N^2) time complexity and we will consider these next.

However, we will see that Bubble sort is one of the worst sorting algorithms we can use for general data.

SELECTION SORT

Searches entire array and finds (selects) the largest/smallest element and puts is where it belongs

- e.g., smallest belongs in A[0] for array A.

Searches the array looking for the second largest/smallest and puts it where it belongs

- e.g., for 2nd smallest in A[1] for array A

Searches the array looking for the third largest/smallest and puts it where it belongs

- e.g., for 3rd smallest in A[2] for array A

etc.

EXAMPLE: SORT THE FOLLOWING DATA USING SELECTION SORT

A	33	12	70	21	-3	34
	0	1	2	3	4	5

ALGORITHM OUTLINE

Input: Array A of integers with given *size*

Output: Array with data sorted in increasing order

Approach (finding smallest)

```
for (i = 0; i < size - 1; i++){
      min =
// find location of smallest value in range i to size-1
// swap values at A[min] and A[i]
}
```

HOW TO FIND MINIMUM?

Considered in Worksheet 2, Question 5:

"Given an integer array, arrA[], and its size (size) write an algorithm (code) in C to find the smallest integer in the array, printing out the integer and its position in the array. You may assume that all values in the array are distinct (i.e., there is only one smallest value)."

arrA	33	12	70	21	-3	34
	0	1	2	3	4	5

Modifying previous for Selection Sort:

We will write this as part of the Selection sort function for now to make the timestep analysis easier (i.e. not as its own function)

We want to keep finding "new" minimums until we have finished sorting, from i = 0 to size - 1

```
min = i; // for some i
//find next smallest
for (j = min + 1; j < size; j++) {
    if (arrA[min] > arrA[j]) {
        min = j;
    }
} // end j for
return(min);
}
```

arrA	33	12	70	21	-3	34
	0	1	2	3	4	5

SELECTION SORT

```
void selectionSort(int[], int);
```

```
28
29
30  //Selection Sort: integer array arrA [] of size
31  void selectionSort(int arrA[], int size) {
32
33      int i, j, min, temp;
34
35      for (i = 0; i < size - 1; i++) {
36          min = i;
37          //find next smallest
38          for (j = min + 1; j < size; j++) {
39              if (arrA[min] > arrA[j]) {
40                  min = j;
41              }
42          }
43
44          //swap values at locations i and min, if i != min
45          if (i != min) {
46              temp = arrA[i];
47              arrA[i] = arrA[min];
48              arrA[min] = temp;
49          }
50      } //end outer while
51  }
52
53
```

arrA	33	12	70	21	-3	34
	0	1	2	3	4	5

TIME STEP ANALYSIS: SELECTION SORT:

Let N = size of array
Assume worst case - values always out of order

```
29
30  //Selection Sort: integer array arrA [] of size
31  void selectionSort(int arrA[], int size) {
32
33      int i, j, min, temp;
34
35      for (i = 0; i < size - 1; i++) {
36          min = i;
37          //find next smallest
38          for (j = min + 1; j < size; j++) {
39              if (arrA[min] > arrA[j]) {
40                  min = j;
41              }
42          }
43
44          //swap values at locations i and min, if i != min
45          if (i != min) {
46              temp = arrA[i];
47              arrA[i] = arrA[min];
48              arrA[min] = temp;
49          }
50      } //end outer while
51  }
52
```

Line	Cost	Num Times	Cost*Num Times	Total
33	1	1		
35	1	N		
36	1	N-1		
38	1	?		
39	1	?		
40	1	?		
45, 46, 47, 48	4	N-1		

CONSIDER LINE 38:

```
35      for (i = 0; i < size - 1; i++) {
36          min = i;
37          //find next smallest
38          for (j = min + 1; j < size; j++) {
39              if (arrA[min] > arrA[j]) {
40                  min = j;
41              }
42          }
```

When `i = 0 (min = 0), j = 1`, the condition (`j < size`) is checked N times

When `i = 1 (min = 1), j = 2`, the condition (`j < size`) is checked N-1 times

When `i = 2 (min = 2), j = 3`, the condition (`j < size`) is checked N-2 times

When `i = size-2, j = size-1`, the condition (`j < size`) is checked twice

So adding all these up:

N+N-1+N-2 + ….. + 2 = (sum of integers from 2 to N):

$\frac{N-1(2+N)}{2} = \frac{N^2+N}{2}$ - 1

CONSIDER LINE 39: if statement

```
35      for (i = 0; i < size - 1; i++) {
36          min = i;
37          //find next smallest
38          for (j = min + 1; j < size; j++) {
39              if (arrA[min] > arrA[j]) {
40                  min = j;
41              }
42          }
```

Line 39 (if statement) iterates once less than line 38 for each time `j` loop iterates:

`i = 0, j = 1`, line 38 checked N times, so Line 39 checked N-1 times

`i = 1, j = 2`, line 38 checked N-1 times, so Line 39 checked N-2 times

`i = 2, j = 3`, line 38 checked N-2 times, so Line 39 checked N-3 times

....

`i = size-2, j = size-1`, line 38 checked twice, so Line 39 checked once

So summing up: N-1+ N-2 …. +2 + 1 $= \frac{N-1(1+N-1)}{2} = \frac{N^2-N}{2}$

CONSIDER LINE 40

```
35      for (i = 0; i < size - 1; i++) {
36          min = i;
37          //find next smallest
38          for (j = min + 1; j < size; j++) {
39              if (arrA[min] > arrA[j]) {
40                  min = j;
41              }
42          }
```

If we assume the condition `arrA[min] > arrA[j]` is always true at Line 39 then Line 40 will also take $\frac{N^2-N}{2}$ timesteps.

YOU TRY ….
PUTTING IT ALL TOGETHER

Handy algebraic expression solver: http://www.webmath.com/anything.html

Line	Cost	Num Times	Cost*Num Times	
33	1	1	1	
35	1	N	N	
36	1	N-1	N-1	
38	1	$\frac{N^2+N}{2}$ - 1	$\frac{N^2+N}{2}$ - 1	
39	1	$\frac{N^2-N}{2}$	$\frac{N^2-N}{2}$	
40	1	$\frac{N^2-N}{2}$	$\frac{N^2-N}{2}$	
45, 46, 47, 48	4	N-1	4(N-1)	
				f(N) = ?

BIG-O ANALYSIS

Selection sort is also O(N^2) where N is the number of values in the array.

Where do we need to add code to count comparisons and swaps?

What line(s) are comparisons happening at? Where will we count them?

What line(s) are swaps happening at? Where will we count them?

```
29
30  //Selection Sort: integer array arrA [] of size
31  void selectionSort(int arrA[], int size) {
32
33      int i, j, min, temp;
34
35      for (i = 0; i < size - 1; i++) {
36          min = i;
37          //find next smallest
38          for (j = min + 1; j < size; j++) {
39              if (arrA[min] > arrA[j]) {
40                  min = j;
41              }
42          }
43
44          //swap values at locations i and min, if i != min
45          if (i != min) {
46              temp = arrA[i];
47              arrA[i] = arrA[min];
48              arrA[min] = temp;
49          }
50      } //end outer while
51  }
52
```

INSERTION SORT

Scans elements in a list inserting each element into its proper position in the previously sorted list.

Steps (sorting items in an array):

- Consider first 2 elements in array and if out of order, sort those 2 (relative to each other)
- Consider 3rd element and if out of order, sort the first three elements relative to each other
- At each stage, consider new element and insert it in its correct position in the previously sorted sub-array

EXAMPLE:
SORT THE FOLLOWING DATA USING INSERTION SORT

arrA	33	12	70	21	-3	34
	0	1	2	3	4	5

ALGORITHM OUTLINE

Input: Array arrA[] of integers of given *size*

Output: Array with data sorted in increasing order

Approach:

Start at position `i=1` and let `curr = arrA[i]`

Compare `curr` with item at position `i-1`

If `curr` is out of order, find the correct position for `curr` in previously sorted sub-array

As you are finding correct position "Make room" for `curr` so that it can be inserted in the correct position (i.e. move values up by one)

Note: Must stop comparing when we reach position 0

ALGORITHM OUTLINE

```
// Given array arrA[] with size, of type integer; integer i
 int i, j, curr;
 for (i = 1; i < size; i++) {
        curr  = arrA[i];
        for (j = i - 1; j >= 0 && curr < arrA[j]; j--) {
       //move ("make room")
            arrA[j + 1] = arrA[j];
       }
       //place curr
}
```

ALGORITHM OUTLINE

```
// Given array arrA[] with size, of type integer; integer i
 int i, j, curr;
 for (i = 1; i < size; i++) {
        curr  = arrA[i];
        for (j = i - 1; j >= 0 && curr < arrA[j]; j--) {
            //move ("make room")
            arrA[j + 1] = arrA[j];
        }
        //place curr
}
```

```
for (j = i - 1; j >= 0 && curr < arrA[j]; j--) {
```

Finding correct position

Two cases are checked in `j` loop to find correct position for `curr`:

- find some element in array at position `j` which is less than `curr`:

  `curr < arrA[j]` not true

or

- have reached the start of the array (i.e. at 0) and `curr` must be inserted there (at position 0), i.e., `j >= 0` not true

```
for (j = i - 1; j >= 0 && curr < arrA[j]; j--) {
     //move ("make room")

    arrA[j + 1] = arrA[j];

}

//place curr

}
```

In addition, must "make room" as we compare so that when we find the correct position we can add it without having to do extra work:

If currently comparing `curr` to value at `arrA[j]`, then:

Let `arrA[j + 1] = arrA[j]` until correct position found for `curr`

INSERTION SORT

```
void insertionSort(int[], int);
```

```
28
29  // Insertion Sort: integer array arrA [] of size
30  void insertionSort(int arrA[], int size)
31  {
32      int i, j, curr;
33
34      for(i = 1; i < size; i++) {
35          curr  = arrA[i];
36
37          for(j = i - 1; j >= 0 && curr < arrA[j]; j--) {   //compare
38              //make room ...
39              arrA[j + 1] = arrA[j];
40          }
41
42          if (i != j + 1) // if not at the correct position already
43              arrA[j + 1] = curr;
44
45      } // end outer i for
46
47  } //return
48
```

arrA

33	12	70	21	-3	34
0	1	2	3	4	5

INSERTION SORT TIME STEP ANALYSIS:

Let N= size of array

Assume worst case …

```
28
29  // Insertion Sort: integer array arrA [] of size
30  void insertionSort(int arrA[], int size)
31  {
32      int i, j, curr;
33
34      for(i = 1; i < size; i++) {
35          curr  = arrA[i];
36
37          for(j = i - 1; j >= 0 && curr < arrA[j]; j--) {   //compare
38              //make room ...
39              arrA[j + 1] = arrA[j];
40          }
41
42          if (i != j + 1) // if not at the correct position already
43              arrA[j + 1] = curr;
44
45      } // end outer i for
46
47  } //return
48
```

Line	Cost	Num Times	Cost*Num Times	Total
32	1	1		
34	1	?		
35	1	?		
37	1	?		
39	1	?		
42&43	2	?		

for loop at line 37:

```
37        for(j = i - 1; j >= 0 && curr < arrA[j]; j--) {   //compare
38            //make room ...
39            arrA[j + 1] = arrA[j];
40        }
41
42        if (i != j + 1) // if not at the correct position already
43            arrA[j + 1] = curr;
44
```

Worst case: `curr` always belongs at `arrA[0]`:

when `i = 1, j = 0`, line 37 checked twice (once true, once not)

when `i = 2, j = 1`, line 37 checked 3 times

when `i = 3, j = 2`, line 37 checked 4 times

….

when `i = N-1, j = N-2`, line 37 checked N times

So adding all these up: N+N-1+N-2 + ….. + 2 = $\frac{N-1(2+N)}{2} = \frac{N^2+N}{2}$ - 1

LINE 39: MOVING

```
37    for(j = i - 1; j >= 0 && curr < arrA[j]; j--) {   //compare
38        //make room ...
39        arrA[j + 1] = arrA[j];
40    }
41
42    if (i != j + 1) // if not at the correct position already
43        arrA[j + 1] = curr;
44
```

Line 39 iterates 1 less for each new j value in while loop:

when i = 1, j = 0, line 37 checked twice, so Line 39 occurs once

when i = 2, j = 1, line 37 checked 3 times, so Line 39 occurs twice

when i = 3, j = 2, line 37 checked 4 times , so Line 39 occurs 3 times

….

when i = N-1, j = N-2, line 37 checked N times, so Line 39 occurs N-1 times

So adding all these up: N-1+N-2 + ….. + 1 $= \frac{N-1\,(1 \;\; + N-1)}{2} = \frac{N^2 - N}{2}$

PUTTING IT ALL TOGETHER:

```
28
29  // Insertion Sort: integer array arrA [] of size
30  void insertionSort(int arrA[], int size)
31  {
32      int i, j, curr;
33
34      for(i = 1; i < size; i++) {
35          curr  = arrA[i];
36
37          for(j = i - 1; j >= 0 && curr < arrA[j]; j--) {    //compare
38              //make room ...
39              arrA[j + 1] = arrA[j];
40          }
41
42          if (i != j + 1) // if not at the correct position already
43              arrA[j + 1] = curr;
44
45      } // end outer i for
46
47  } //return
48
```

Line	Cost	Num Times		
32	1			
34	1			
35	1			
37	1			
39	1			
42&43	2			

BIG-O ANALYSIS

Insertion sort is also O(N^2) where N is the number of values in the array.

However we can see differences in terms of comparisons and swaps …

Where do we need to add code to count comparisons and swaps?

What line(s) are comparisons happening at? Where will we count them?

What line(s) are swaps happening at? Where will we count them?

```
28
29  // Insertion Sort: integer array arrA [] of size
30  void insertionSort(int arrA[], int size)
31  {
32      int i, j, curr;
33
34      for(i = 1; i < size; i++) {
35          curr  = arrA[i];
36
37          for(j = i - 1; j >= 0 && curr < arrA[j]; j--) {   //compare
38              //make room ...
39              arrA[j + 1] = arrA[j];
40          }
41
42          if (i != j + 1) // if not at the correct position already
43              arrA[j + 1] = curr;
44
45      } // end outer i for
46
47  } //return
48
```

Sorting 1000 integers
Some results for a "typical" run (all sorting the same integers)

```
1000 items have been read
In Insertion Sort:
time taken is: 0.002000 seconds;
number of swaps is: 257335;
number of comparisons is: 258334_
```

```
1000 items have been read
In Selection Sort:
time taken is 0.003000;
number of swaps is 999;
number of comparisons is 499500_
```

```
1000 items have been read
In Bubble Sort:
time taken is 0.007000;
number of swaps is 257335;
number of comparisons is 499500
```

WHAT DOES THE DATA LOOK LIKE?

For a 1000 “random” integer numbers-

- Good mixed distribution
- Were not sorted

QUESTIONS

1. Would you expect performance to be different if:

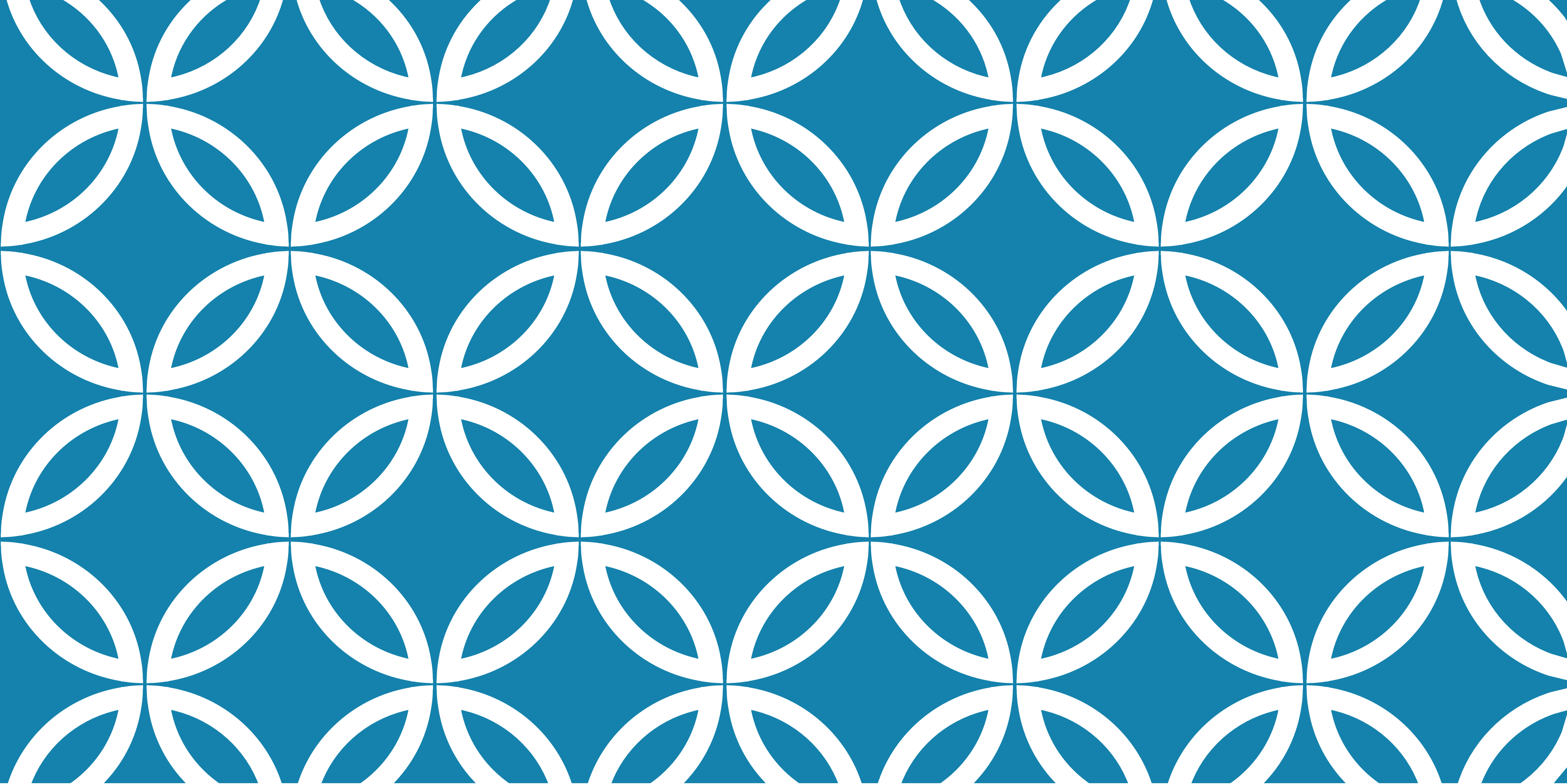
Integers in file were already sorted

Integers in file were sorted in descending sorte d order?

2. When you add code to count steps and comparisons should the time step analysis be updated to include the new code?

SUMMARY

Bubble, Selection and Insertion Sort are similar types of sorting algorithms – they work by comparing and swapping/moving data - and are characterised by a nested loop that gives a quadratic function dependent on N, the number of values to sort.



CT102: ALGORITHMS

Parallel Arrays & Merging Sorted Arrays

PARALLEL ARRAYS

Parallel arrays refer to multiple arrays of the same size used to store records.

A separate array, with data of the same type, is used for each field of the record.

Each array must have the same size but may have different data types.

Values for a record are located at the same index value in each array.

MOTIVATION

Consider the case where you want to hold different types of data for the same occurrence:

For example:

- Name and assignment marks per student in a class.
- Average, minimum, maximum and stdev per exam in a year.

With parallel arrays we can imagine data relating to something of interest in a 'column' of array entries.

An index value can access the same locations in different arrays, e.g. location 0, 1, 2, etc.

EXAMPLE:

Data held on students is:

```
name, id, examScore
```

with

name a string,

id an integer

examScore an integer

If using arrays, then can use 3 parallel arrays to hold this information …

EXAMPLE: 3 Parallel arrays to hold name, id and exam score

name	Julie	Sam	Ron	Ann	Sue
id	123	432	35	415	515
examScore	40	65	68	70	85

For an individual student:

- `name[i]` gives student name
- `id[i]` gives the corresponding id of that student
- `examScore[i]` gives the corresponding exam score of that student

e.g,

- `name[1]` is student named Sam, his id can be found at `id[1]` (432) and his exam score can be found at `examScore[1]` (65)

STEPS WHEN CREATING PARALLEL ARRAYS

Declare each array, specifying names, data types and size:

- The size of all arrays should be the same
- Data types of arrays can be different

Populate the arrays in parallel, i.e., put values into each array at location 0, location 1, location 2, etc.

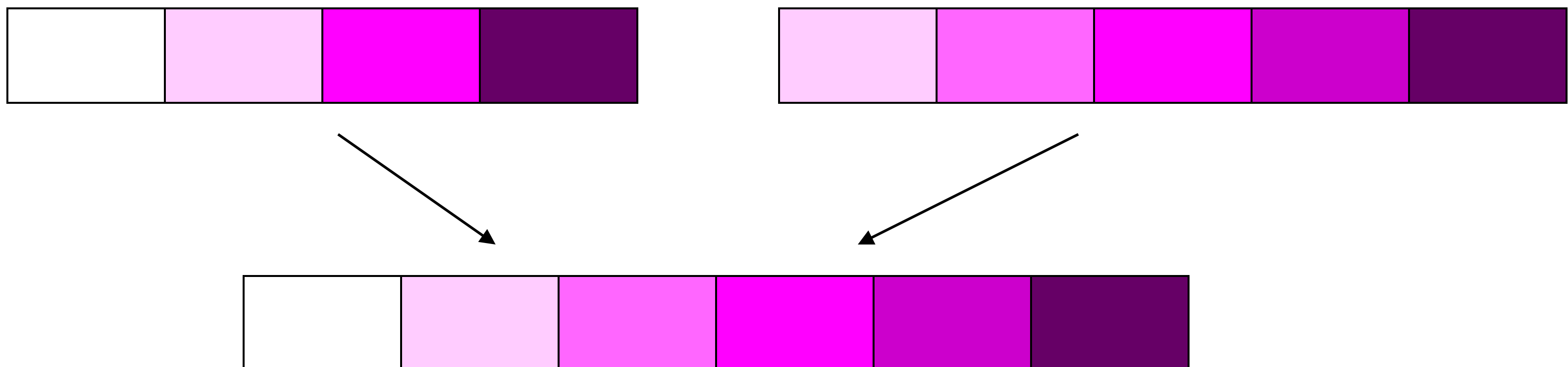
IN C:

```
int i;
int size = 5;
char *names[] = {"Julie","Sam","Ron","Ann","Sue"};
int  id[] = {123, 432, 35, 415,515};
int  examScore[] = {40, 65, 68, 70, 85};

    for (i = 0; i < size; i++) {
            printf("Name: %s, ID: %d, Exam Score: %d \n",
            names[i], id[i], examScore[i]);
}
```

MERGING SORTED ARRAYS

Given two sorted lists of data (possibly with duplicate values) merging involves combining the values from both lists, in sorted order, into a single sorted list.



INPUTS, OUTPUTS AND ASSUMPTIONS

Inputs:

Sorted array `arrA[]` of size `sizeA`, unique values

Sorted array `arrB[]` of size `sizeB`, unique values

Outputs:

Sorted array `arrC[]` of size `sizeC` containing data from `arrA[]` and `arrB[]`, unique values

Assumptions:

Duplicates are not included in `arrC[]`, i.e. each value is only present once

Note: Can easily modify code to include duplicates later.

EXAMPLE:
What values are in `arrC[]`?

arrA

1	12	17	19	29	44	49

arrB

2	4	12	14	24

arrC

INDEXING 3 ARRAYS

Maintain 3 indexes - one for each array:

`i` is index for `arrA`

`j` is index for `arrB`

`k` is index for `arrC`

`i` and `j` represent the index of the next values to be compared in the arrays to merge.

`k` represents the next position to be filled in the new array `arrC`

Size of `arrC`?

- Can be no larger than `sizeA + sizeB`

STEPS:

while not at the end of `arrA` and `ArrB` compare `arrA[i]` and `arrB[j]` putting smallest into `arrC[k]`

increment `i`, `j` and `k` appropriately.

if at end of `arrA` and there are values left in `arrB`, put values from `arrB` into `arrC` (incrementing `j` and `k`)

else if at end of `arrB` and there are values left in `arrA`, move values from `arrA` into `arrC` (incrementing `i` and `k`)

GETTING STARTED....

```
void merge (int arrA[], int sizeA, int arrB[], int sizeB) {

   int i, j, k;
   int sizeC;

   i = j = k = 0;
```

GETTING STARTED....

```c
// Setting size of C
sizeC = sizeA + sizeB;

 // declare arrC of size sizeC using malloc (memory allocation)
 int *arrC;
 arrC = (int*) malloc(sizeC * sizeof(int));
```

COMPARING ....

```
19
20          while (i < sizeA && j < sizeB) {
21
22              if (arrA[i] < arrB[j]) {
23                  arrC[k] = arrA[i];
24                  i++;
25                  k++;
26              }
27              else if (arrB[j] < arrA[i]) {
28                  arrC[k] = arrB[j];
29                  j++;
30                  k++;
31              }
32               else if (arrB[j] == arrA[i]) {     // can replce with else
33                  arrC[k] = arrA[i];
34                  i++;
35                  j++;
36                  k++;
37              }
38
39          } //end while
40
```

FINISHED COMPARING ….

```
41
42         //reached the end of one of the arrays at this point
43         if (i == sizeA) {       // all of arrA written to arrC already
44
45             while(j < sizeB) {
46                 arrC[k] = arrB[j];
47                 j++;
48                 k++;
49             }
50         }
51
52         else if (j == sizeB){   //all of arrB written to arrC already
53
54             while (i < sizeA) {
55                 arrC[k] = arrA[i];
56                 i++;
57                 k++;
58             }
59         }
60         sizeC = k; //correct value of sizeC
61
```

TIME STEP ANALYSIS
let P = sizeA and M = sizeB

Line	Cost	Num Times	Cost*Num Times	Total
10, 11, 13, 14, 17, 18	6	1 each	6	
20	1	P+1 or M+1 Assume P+1 without loss of generality	P+1	
22 or 22 & 27 or 22, 27 & 32	3	P times	3P	
23, 24, 25 or 28, 29, 30 or 33, 34, 35, 36	4	P times	4P	
				8P+7

```
 5
 6  // Function to merge the values in 2 integer arrays - not keeping duplicates
 7  // Assumes data sorted in arrA[] and arrB[]
 8  void merge (int arrA[], int sizeA, int arrB[], int sizeB) {
 9
10          int i, j, k;
11          int sizeC;
12
13          i = j = k = 0;
14          sizeC = sizeA + sizeB;
15
16          // declare arrC of size sizeC
17          int *arrC;
18          arrC = (int*) malloc(sizeC * sizeof(int));
19
20          while (i < sizeA && j < sizeB) {
21
22              if (arrA[i] < arrB[j]) {
23                  arrC[k] = arrA[i];
24                  i++;
25                  k++;
26              }
27              else if (arrB[j] < arrA[i]) {
28                  arrC[k] = arrB[j];
29                  j++;
30                  k++;
31              }
32               else if (arrB[j] == arrA[i]) {      // can replce with else
33                  arrC[k] = arrA[i];
34                  i++;
35                  j++;
36                  k++;
37              }
38
39          } //end while
40
```

TIME STEP ANALYSIS *ctd.*

```
41
42        //reached the end of one of the arrays at this point
43        if (i == sizeA) {       // all of arrA written to arrC already
44
45            while(j < sizeB) {
46                arrC[k] = arrB[j];
47                j++;
48                k++;
49            }
50        }
51
52        else if (j == sizeB){   //all of arrB written to arrC already
53
54            while (i < sizeA) {
55                arrC[k] = arrA[i];
56                i++;
57                k++;
58            }
59        }
60        sizeC = k; //correct value of sizeC
61
```

Line	Cost	Num Times	Cost*Num Times	Total
(previous)				8P + 7
43 or 43 and 52	2	1	2	
45 or 54 (only one)	1	M-j + 1 *Assume j = 0 worst case*	M+1	
46, 47, 48 (or 55, 56, 57)	3	M-j *Assume j = 0 worst case*	3M	
60	1	1	1	1
				8P+4M+11
				O(P+M)

Modifications needed if we have non unique values in each array and want to keep all duplicates?

Same code at start

```
19
20      while (i < sizeA && j < sizeB) {
21
22          if (arrA[i] < arrB[j]) {
23              arrC[k] = arrA[i];
24              i++;
25              k++;
26          }
27          else if (arrB[j] < arrA[i]) {
28              arrC[k] = arrB[j];
29              j++;
30              k++;
31          }
32           else if (arrB[j] == arrA[i]) {     // can replce with else
33              arrC[k] = arrA[i];
34              i++;
35              j++;
36              k++;
37          }
38
39      } //end while
40
```

Change operator to <=

Delete Lines 32 to 36

Keep remaining code

HOW DOES TIME STEP ANALYSIS DIFFER?

Line	Cost	Num Times	Cost*Num Times	Total
10, 11, 13, 14, 17, 18	6	1 each	6	
20	1	P+1 or M+1 Assume P+1 without loss of generality	P+1	
22 or 22 & 27 ~~or 22, 27 & 32~~	2	P times	2P	
23, 24, 25 or 28, 29, 30 ~~or 33, 34, 35, 36~~	3	P times	3P	
				6P+7

```
19
20      while (i < sizeA && j < sizeB) {
21
22          if (arrA[i] < arrB[j]) {
23              arrC[k] = arrA[i];
24              i++;
25              k++;
26          }
27          else if (arrB[j] < arrA[i]) {
28              arrC[k] = arrB[j];
29              j++;
30              k++;
31          }
32           else if (arrB[j] == arrA[i]) {
33              arrC[k] = arrA[i];
34              i++;
35              j++;
36              k++;
37          }
38
39      } //end while
```

TIME STEP ANALYSIS *ctd.*

```
41
42      //reached the end of one of the arrays at this point
43      if (i == sizeA) {       // all of arrA written to arrC already
44
45          while(j < sizeB) {
46              arrC[k] = arrB[j];
47              j++;
48              k++;
49          }
50      }
51
52      else if (j == sizeB){   //all of arrB written to arrC already
53
54          while (i < sizeA) {
55              arrC[k] = arrA[i];
56              i++;
57              k++;
58          }
59      }
60      sizeC = k; //correct value of sizeC
61
```

Line	Cost	Num Times	Cost*Num Times	Total
(previous)				6P + 7
43 or 43 and 52	2	1	2	
45 or 54 (only one)	1	M-j + 1 *Assume j = 0* *worst case*	M+1	
46, 47, 48 (or 55, 56, 57)	3	M-j *Assume j = 0* *worst case*	3M	
60	1	1	1	1
				6P+4M+11
				O(P+M)

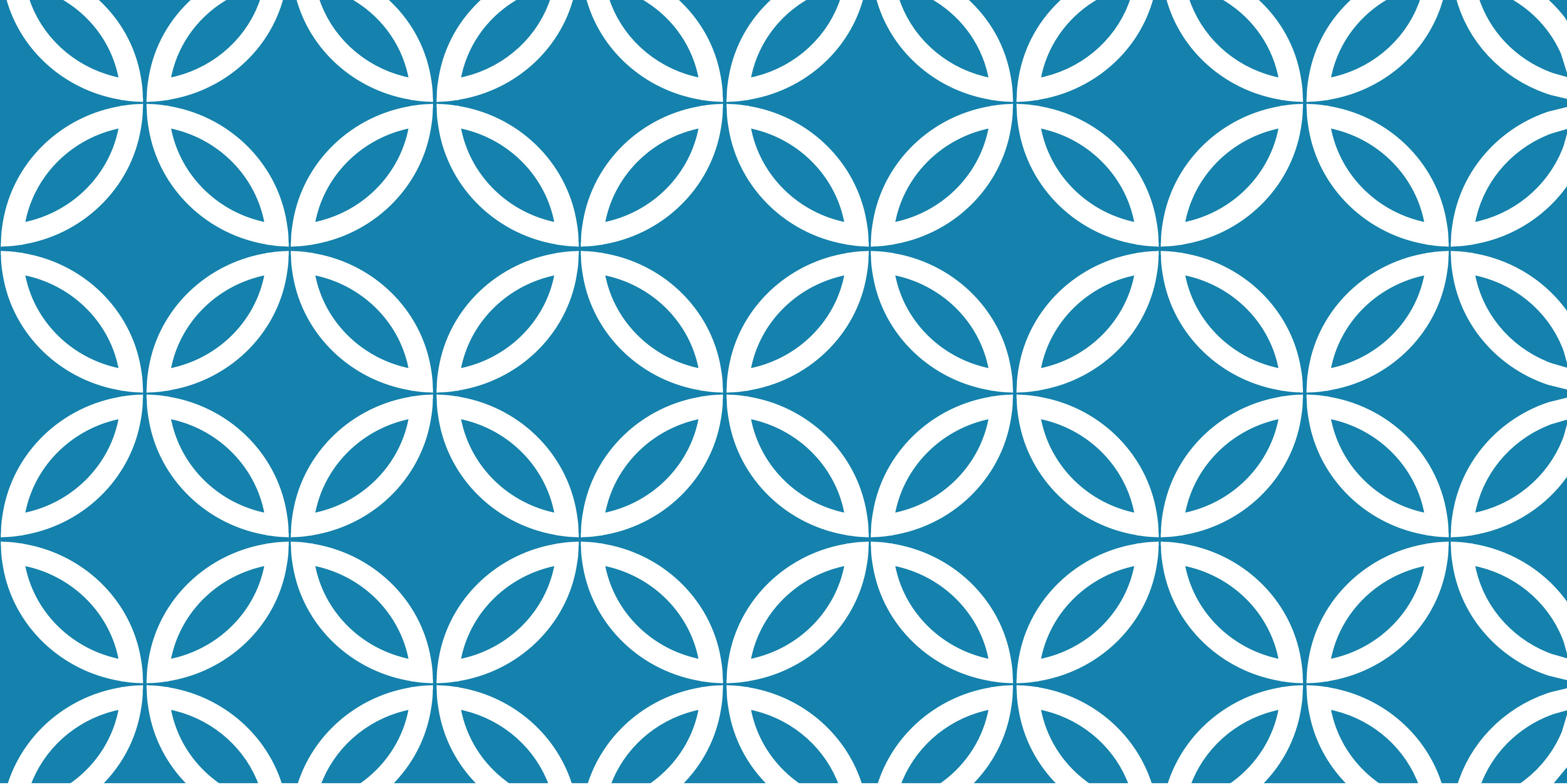
SUMMARY

We often want to keep data in sorted order (so that a binary search can be performed more efficiently than a linear search)

However we must also be able to easily add data to our sorted data (i.e., maintaining the sorted order).

One way to do this is to add new data to a temporary file (adding it in sorted order, or sorting after it has been added), and then merge this sorted data with the existing sorted data … for this we need a merge algorithm.

Because Big-O is linear ($O(P+M)$) this is always better than merging the data in unsorted order and re-sorting it.



CT102: ALGORITHMS

Topic: Counting Positive Integers

COUNTING

Many complex systems are built on components that involve *counting*

For example, some data mining approaches, search engines (as seen with *tf*idf*), information entropy and information gain, compression techniques (as seen with frequencies), etc.

In general, counting involves finding the number of occurrences, or *frequency*, of one or more items, or groups of items, in a collection.

YOU TRY …
Counting integers in arrays

Given an array arrA[] of size N, containing (unsorted) positive integer values in the range [0-6] write a function to count the frequency of each integer in the array:

For example,

arrA[15] = {1, 1, 1, 2, 2, 4, 3, 1, 1, 6, 5, 4, 1, 1, 6};

APPROACH 1

Check each value and update appropriate count

```
void version1(int arrA[], int size){
    int i;
    int count = 0;

    for (i = 0; i < size; i++) {
        if (arrA[i] == 1) {
            ++count;
        }
    }
    printf(" \n The numnber of 1's is %d:", count);
    // etc. for all other numbers
}
```

Approach 1 is not very efficient! Alternatives?

Approach 2: Don't loop for each value – can check for each value within one loop and update appropriate count, e.g. count0, count1, count2, etc.

```
void version2(int arrA[], int size){
    int i;
    int count0, count1, count2, count3, count4, count5, count6;
    count0 = count1 = count2 = count3 = count4 = count5 = count6 = 0;

    for (i = 0; i < size; i++) {
        if (arrA[i] == 0) {
            ++count0;
        }
        else if (arrA[i] == 1) {
            ++count1;
        }
        else if (arrA[i] == 2) {
            ++count2;
        }
        else if (arrA[i] == 3) {
            ++count3;
        }
        // etc for all other numbers
    }
    printf(" \n The numnber of 0's is %d:", count0);
    printf(" \n The numnber of 1's is %d:", count1);
    printf(" \n The numnber of 2's is %d:", count2);
    // etc. for all other numbers
}
```

Approach 2 is not very efficient! Especially if we are counting many different values
Alternatives?

Approach 3 We can use an integer array (`count[]`) to keep track of the counts for us as we loop through array (or file). In the case of counting positive integers:

- Index position 0 holds count of 0s
- Index position 1 holds count of 1s
- Index position 2 holds count of 2s
- etc.

The size of this array will be dictated by how many distinct values nee to be counted.

All locations in the array must be initialised to 0 at start

EXAMPLE: using array count[] to hold counts

arrA[15] = {1, 1, 1, 2, 2, 4, 3, 1, 1, 6, 5, 4, 1, 1, 6};

Initially:

count[]	0	0	0	0	0	0	0
	0	*1*	*2*	*3*	*4*	*5*	*6*

After counting:

count[]	0	7	2	1	2	1	2
	0	*1*	*2*	*3*	*4*	*5*	*6*

i.e.,

count[0] will contain the number of times 0 occurs in arrA[] (0)

count[1] will contain the number of times 1 occurs in arrA[] (7)

count[2] will contain the number of times 1 occurs in arrA[] (2) *etc.*

APPROACH 3 ADVANTAGES

A more important advantage of approach 3 is that we do not need an if statement to explicitly check if we have a 0 or a 1 or a 2 etc.

How to count then?

The value in the original array (`arrA[]`) becomes the index value of the counting array (`count[]`):

- When value is 0 in `arrA[]`, go to index position 0 in `count[]`, update: `++count[0]`
- When value is 1 in `arrA[]`, go to index position 1 in `count[]`, update count `++count[1]`
- When value is 2 in `arrA[]`, go to index position 2 in `count[]`, update count `++count[2]`
- At each stage: `++count[arrA[i]]`

ARRAY count[]

Consider example again …

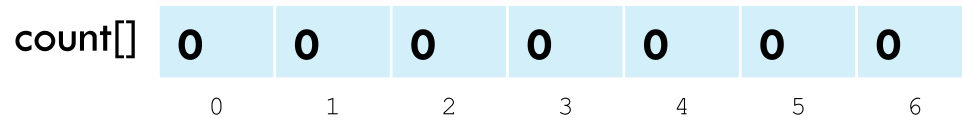
arrA[15] = {1, 1, 1, 2, 2, 4, 3, 1, 1, 6, 5, 4, 1, 1, 6};

```
int i;
int size = 15;
int arrA[15] = {1, 1, 1, 2, 2, 4, 3, 1, 1, 6, 5, 4, 1, 1, 6};
int count[7] = {0, 0, 0, 0, 0, 0, 0};

for (i = 0; i < size; i++) {
    ++count[arrA[i]];
}
```

EXAMPLE:

Initially:



arrA[15] = {1, 1, 1, 2, 2, 4, 3, 1, 1, 6, 5, 4, 1, 1, 6};

i	arrA[i]	++count[arrA[i]]	Value at count[arrA[i]]
0	arrA[0] is 1	++count[arrA[0]] ++count[1]	1
1	arrA[1] is 1	++count[arrA[1]] ++count[1]	2
2	arrA[2] is 1	++count[arrA[2]] ++count[2]	3
3	arrA[3] is 2	++count[arrA[3]] ++count[2]	1
4	arrA[4] is 2	++count[arrA[4]] ++count[2]	2
etc			

WORKSHEET QUESTIONS

Array indexes:

Given the following arrays: arrA[] and freq[]:

```
arrA[10] = {4, 5, 4, 3, 0, 1, 4, 5, 5, 4};
freq[6] = {0, 0, 0, 0, 0, 0};
```

What value do each of the following have?

```
arrA[2]
freq[2]
arrA[ freq[2] ]
++arrA[ freq[2] ]
++freq[ arrA[1] ]
```

UPDATING `count[]` with `++count[ arrA[i] ];`

`++count[arrA[i]]` works IF and ONLY IF:

- `arrA[]` stores positive integers (but can be modified for negative integers)
- `count[]` has the correct size so that any potential value in `arrA[]` has a corresponding index in `count[]`

WHAT IS THE SIZE OF `count[]`?

For positive integers, the size of `count[]` must be one more than the largest value we are counting (i.e. the max value in `arrA[]`) so that there is an index for this max value

e.g.,

if max value is 100 then `count[]` must be of size 101

In previous example, max value was 6 so size of `count[]` is 7

We need to be able to dynamically create the array `count[]` with the correct size.

BACK TO CODE FOR COUNTING:

GIVEN ARRAY `arrA[]` OF SIZE *`size`* WITH POSITIVE INTEGERS:

no "if" required

1. Find maxVal (max value in `arrA[]`)
2. Declare and initialise (to 0) an array `count[maxVal + 1]` which will hold the counts for each integer in arrA[].
3. Loop from i = 0 to size-1 inclusive and at each stage update by 1 the location `arrA[i]` in `count[]`:

```
++count[ arrA[i] ];
```

APPROACH 3:

I am going to use the name freq[] for the array that holds the counts – it is of size freqSize

```
// assuming we have arrA[] and its size
// assuming we have declared freq to be of size freqSize

// initialise freq[] to 0
for (i = 0; i < freqSize; i++) {
        freq[i] = 0;
}

// start counting
for (i = 0; i < size; i++) {
    ++freq[ arrA[i] ];
}

// output counts
for (i = 0; i < freqSize; i++) {
        printf("\n The number of %d's is %d: ", i, freq[i]);
}
```

Declaring freq[] given maxVal, the maximum value

Use malloc() to dynamically set the size of the array:

```
// assuming you have found maxVal
int *freq;
int freqSize;
freqSize = maxVal + 1;
freq = (int*)malloc(freqSize * sizeof(int));   //create
```

FULL FUNCTION

```
Number of 0's is: 0
Number of 1's is: 7
Number of 2's is: 2
Number of 3's is: 1
Number of 4's is: 2
Number of 5's is: 1
Number of 6's is: 2
```

```
27
28  // count the frequency of each integer in an array and print
29  void count(int arrA[], int size, int maxVal) {
30
31      int i, val;
32      int *freq;
33      int freqSize;
34
35      freqSize = maxVal + 1;
36      freq = (int*)malloc(freqSize * sizeof(int));
37
38
39      //initialise freq[] to 0
40      for (i = 0; i < freqSize; i++) {
41          freq[i] = 0;
42      }
43
44      // start counting
45      for (i = 0; i < size; i++) {
46          ++freq[ arrA[i] ];
47      }
48
49
50      //output counts
51      for (i = 0; i < freqSize; i++) {
52          printf("\n The number of %d's is %d", i, freq[i]);
53      }
54
55  }
56
```

APPROACH 3

Timestep analysis and Big-O analysis

(Ignore output – lines 51&52)

Let N = size

maxVal is maximum value in arrA[]

```
27
28  // count the frequency of each integer in an array and print
29  void count(int arrA[], int size, int maxVal) {
30
31      int i, val;
32      int *freq;
33      int freqSize;
34
35      freqSize = maxVal + 1;
36      freq = (int*)malloc(freqSize * sizeof(int));
37
38
39      //initialise freq[] to 0
40      for (i = 0; i < freqSize; i++) {
41          freq[i] = 0;
42      }
43
44      // start counting
45      for (i = 0; i < size; i++) {
46          ++freq[ arrA[i] ];
47      }
48
49
50      //output counts
51      for (i = 0; i < freqSize; i++) {
52          printf("\n The number of %d's is %d", i, freq[i]);
53      }
54
55  }
56
```

Line	Cost	numTimes	Cost*numTimes	
31-33 35,36	5	1	5	
40	1	maxVal + 2	maxVal + 2	
41	1	maxVal + 1	maxVal + 1	
45	1	N + 1	N + 1	
46	1	N	N	
				F(N) = 2N + 2maxVal + 9
				Big-O: O(N + maxVal)

How to find maxVal?

This will require another scan of arrA[] (all N elements) but this will also be O(N) so will not affect the linear time complexity overall

We have seen the code for finding the minimum value previously (as part of Selection Sort), so will need to modify this slightly to find the maximum value, if it is not already known.

MODIFICATIONS

- Consider how/if the algorithm can be modified if the minVal is much greater than 0, e.g.,

```
arrA[8] = {40, 50, 40, 30, 80, 50, 40, 50};
```

- Consider how/if the algorithm can be modified to count both negative and positive integers, e.g.,

```
arrA[8] = {-4, -10, 0, -4, -2, 0, -2, 10};
```

APPLICATIONS

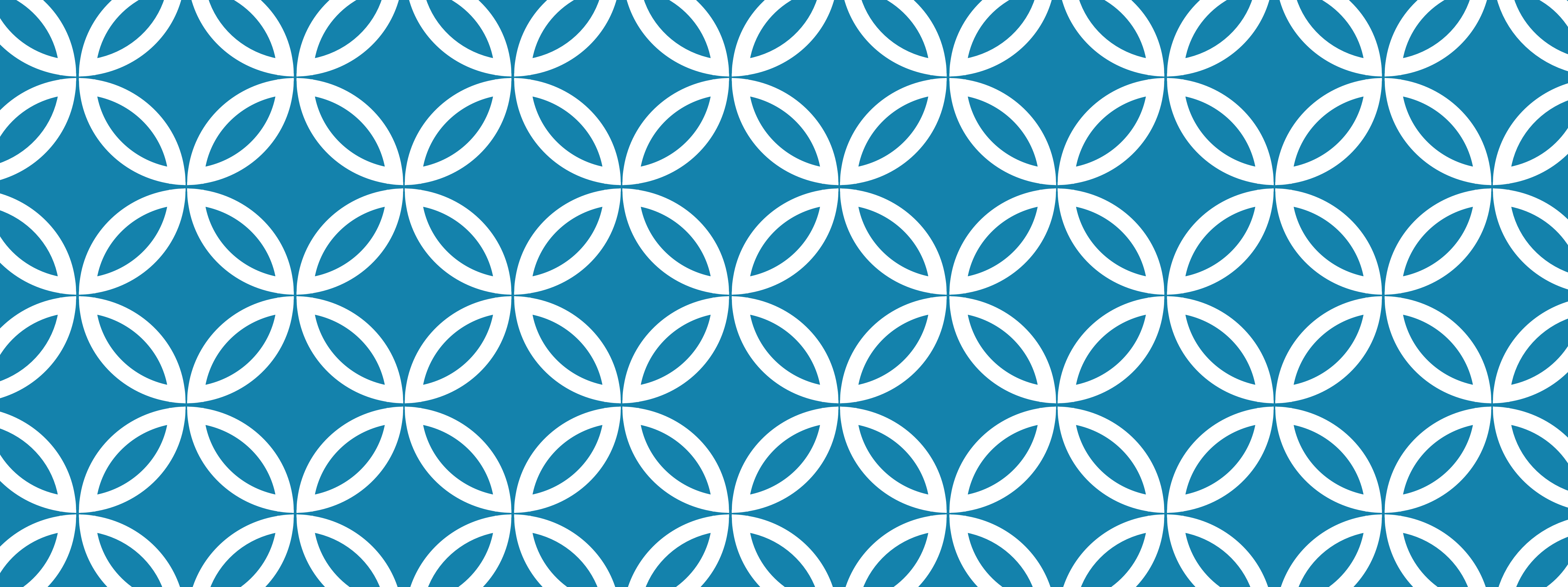
Can you solve the following problems in linear order time complexity?

Given an array of N integers in unsorted order with *mostly* unique values and values are in the range [0-100]. Write an algorithm which will find and print the any value that is present more than once.

Given an array of N integers in the range [0-100] and in unsorted order, write an algorithm to check whether all values in the array are unique.

SUMMARY

- If you have a small range then counting integers can be done very efficiently using an integer array to keep track of counts (other ways to store counts include hash maps and dictionaries)
- This forms the basis of the next sorting algorithm we will consider … countSort



TOPIC:
COUNTING AND COUNT SORT

CT102: Algorithms

RECALL: Counting

If you have a small range (maxVal is small and minVal is 0) then counting N positive integers can be done very efficiently using an integer array to keep track of the counts: O(N + maxVal)

```
27
28  // count the frequency of each integer in an array and print
29  void count(int arrA[], int size, int maxVal) {
30
31      int i, val;
32      int *freq;
33      int freqSize;
34
35      freqSize = maxVal + 1;
36      freq = (int*)malloc(freqSize * sizeof(int));
37
38
39      //initialise freq[] to 0
40      for (i = 0; i < freqSize; i++) {
41          freq[i] = 0;
42      }
43
44      // start counting
45      for (i = 0; i < size; i++) {
46          ++freq[ arrA[i] ];
47      }
48
49
50      //output counts
51      for (i = 0; i < freqSize; i++) {
52          printf("\n The number of %d's is %d", i, freq[i]);
53      }
54
55  }
56
```

MODIFICATIONS

- Consider how/if the algorithm can be modified if the minVal is much greater than 0, e.g.,

```
arrA[8] = {40, 80, 30, 41, 52, 52, 41, 52};
```

- Consider how/if the algorithm can be modified to count both negative and positive integers, e.g.,

```
arrA[8] = {-4, -10, 0, -4, -2, 0, -2, 10};
```

SOLUTION

To work with negative integers (or to work with positive integers in a non-0 based range) we need to know both the minVal and the maxVal.

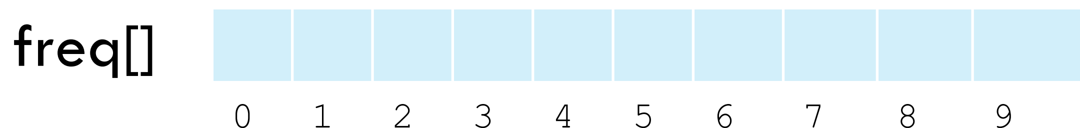
The idea then is to store the minVal at index 0 and to offset the location of all other values based on this minVal.

The size of freq will be `maxVal - minVal + 1`

- freq[0] will store the count of the minVal
- freq[freqSize-1] will store the count of the maxVal

For any integer in arrA[], the associated location in freq[] will be:

`arrA[i] - minVal`



EXAMPLE WITH POSITIVE & NEGATIVE INTEGERS

```
arrA[8] = {-4, -10, 0, -4, -2, 0, -2, 10};
```

In the above example, the minVal is -10 and the maxVal is 10 i.e., the range of integers is [-10, 10] then:

freqSize = 10-(-10)+1 = 21

At each stage: `++freq[arrA[i] – minVal]`

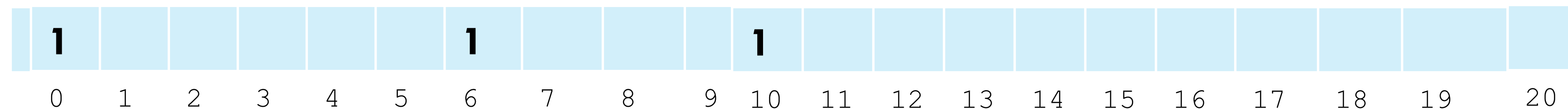
When `i=0 arrA[i] = -4 =>  ++freq[-4 --10] => ++freq[6]`

When `i=1 arrA[i] = -10 => ++freq[-10 --10] => ++freq[0]`

When `i=2 arrA[i] =  0  => ++freq[0 --10] => ++freq[10]`

etc.

freq[]



NON-ZERO BASED POSITIVE RANGE
? Question ... why bother with this?

```
arrA[8] = {40, 80, 30, 41, 52, 52, 41, 52};
```

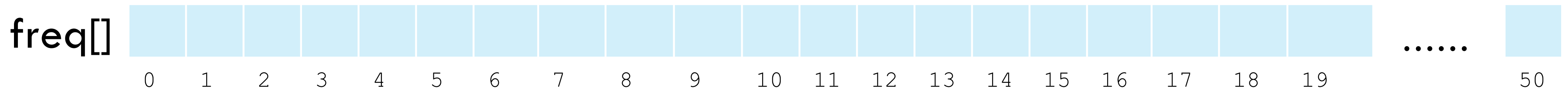
In the above example, the minVal is 30 and the maxVal is 80 i.e., the range of integers is [30, 80] then:

freqSize = 80-(30)+1 = 51

At each stage: `++freq[arrA[i] – minVal]`

```
When i=0 arrA[i] = 40 => ++freq[40 - 30] => ++freq[10]
When i=1 arrA[i] = 80 => ++freq[80 - 30] => ++freq[50]
When i=2 arrA[i] = 30  => ++freq[30 - 30] => ++freq[0]
```

etc.



CHANGES NEEDED TO FUNCTION?

1. Function definition:

```
void countSortRange(int arrA[], int size, int minVal, int maxVal) {
```

2. The size of array freq[]:

```
freqSize = maxVal - minVal + 1;
```

3. Counting

```
++freq[arrA[i] - minVal];
```

APPLICATIONS

1. Given an array of N integers in unsorted order with *mostly* unique values and values are in the range [0-100]. Write an algorithm which will find and print the any value that is present more than once.

2. Given an array of N integers in the range [0-100] and in unsorted order, write an algorithm to check whether all values in the array are unique.

1. Given an array of N integers in unsorted order with *mostly unique* values and values are in the range [0-100]. Write an algorithm which will find and print any value that is present more than once.

Solution:

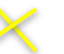
All integers are positive and the maxVal value passed should be 100

Previous solution will work up to line 51

When printing out values, or storing them, (line 52) only want to print/stor duplicates:

```
//output repeating values
for(i = 0; i < freqSize; i++) {
    if (freq[i] > 1) {
        printf("\n Number %d occurs %d times", i, freq[i]);
    }
}
```

```
27
28  // count the frequency of each integer in an array and print
29  void count(int arrA[], int size, int maxVal) {
30
31      int i, val;
32      int *freq;
33      int freqSize;
34
35      freqSize = maxVal + 1;
36      freq = (int*)malloc(freqSize * sizeof(int));
37
38
39      //initialise freq[] to 0
40      for (i = 0; i < freqSize; i++) {
41          freq[i] = 0;
42      }
43
44      // start counting
45      for (i = 0; i < size; i++) {
46          ++freq[ arrA[i] ];
47      }
48
49
50      //output counts
51      for (i = 0; i < freqSize; i++) {
52          printf("\n The number of %d's is %d", i, freq[i]);
53      }
54
55  }
56
```



2. Given an array of N integers in the range [0-100] and in unsorted order, write an algorithm to check whether all values in the array are unique.

Solution:

Previous solution will work up to line 51 but may want to re-write as a function which will return a Boolean (true for unique), false otherwise:

```
bool isUnique(int arrA[], int size, int maxVal){
```

Modify code to check if all values in freq[] are either 0 or 1

```
//check for any non-unique value
bool isUnique = true;
for(i = 0; i < freqSize && isUnique; i++) {
    if (freq[i] > 1) {
        isUnique = false;
    }
}
```

```
// checks for unique values
bool isUnique(int arrA[], int size, int maxVal){

    int i, val;
    int *freq;
    int freqSize;

    freqSize = maxVal + 1;
    freq = (int*)malloc(freqSize * sizeof(int));

    //initialise freq[]
    for(i = 0; i < freqSize; i++) {
        freq[i] = 0;
    }

    //count
    for(i = 0; i < size; i++) {
        ++freq[arrA[i]];
    }

    //check for any non-unique value
    bool isUnique = true;
    for(i = 0; i < freqSize && isUnique; i++) {
        if (freq[i] > 1) {
            isUnique = false;
        }
    }
    return(isUnique);
}
```

NOTE:

As `isUnique()` will be false once we find any repeated number, we do not need to do a full scan of `freq[]` and can incorporate the new code in to the counting.

```
// checks for unique values
bool isUnique(int arrA[], int size, int maxVal){

    int i, val;
    int *freq;
    int freqSize;
    bool isUnique = true;

    freqSize = maxVal + 1;
    freq = (int*)malloc(freqSize * sizeof(int));

    // initialise freq[]
    for(i = 0; i < freqSize; i++) {
        freq[i] = 0;
    }

    // count and check for uniqueness
    for(i = 0; i < size && isUnique; i++) {
        ++freq[arrA[i]];
        if (freq[i] > 1) {
            isUnique = false;
        }
    }
    return(isUnique);
}
```

New Problem: Given an array of non-unique N integers in unsorted order, write an algorithm to sort the integers.

Idea:

If we know the frequency of each value, we can figure out where the value belongs in the sorted array.

For example, if we know that 0 occurs 5 times, then the first 5 locations (locations 0, 1, 2, 3 and 4) in the sorted array are 0.

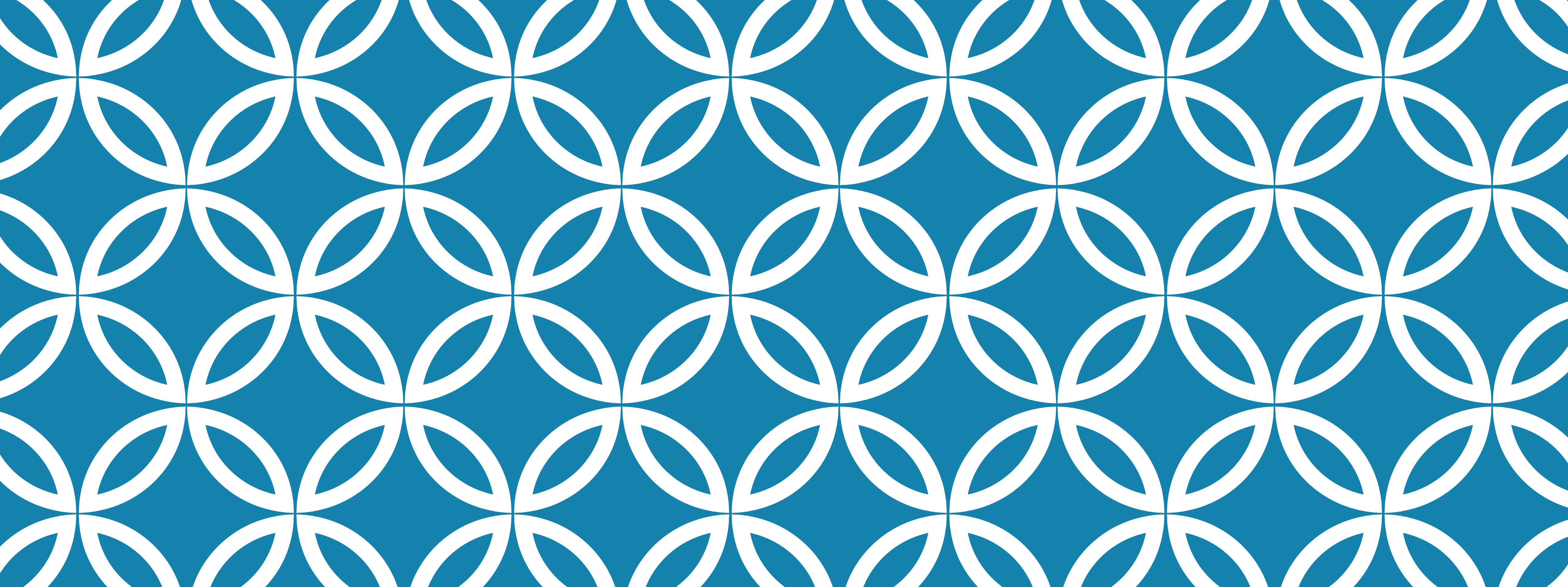
However, if possible we want to avoid using a nested loop or an if statement when we place items

New Idea:

If we have the freq of the number of values <= any value in the array we can place the current value in index position one less than this and decrement the freq

For example, if we come across a 0 in the original array and know that there are 5 values <= 0, then the 0 we are at can be placed in position 4 and decrement freq to 4

Next time we come across a 0 in the original array we see that there are now 4 values <= 0, and we can place the next 0 in position 3



TOPIC:
COUNTING AND COUNT SORT

CT102: Algorithms

RECALL: Counting

If you have a small range (maxVal is small and minVal is 0) then counting N positive integers can be done very efficiently using an integer array to keep track of the counts: O(N + maxVal)

```
27
28  // count the frequency of each integer in an array and print
29  void count(int arrA[], int size, int maxVal) {
30
31      int i, val;
32      int *freq;
33      int freqSize;
34
35      freqSize = maxVal + 1;
36      freq = (int*)malloc(freqSize * sizeof(int));
37
38
39      //initialise freq[] to 0
40      for (i = 0; i < freqSize; i++) {
41          freq[i] = 0;
42      }
43
44      // start counting
45      for (i = 0; i < size; i++) {
46          ++freq[ arrA[i] ];
47      }
48
49
50      //output counts
51      for (i = 0; i < freqSize; i++) {
52          printf("\n The number of %d's is %d", i, freq[i]);
53      }
54
55  }
56
```

Problem Statement: Given an array of non-unique N integers in unsorted order, write an algorithm to sort the integers.

Idea:

If we know the frequency of each value, we can figure out where the value belongs in the sorted array.

For example, if we know that 0 occurs 5 times, then the first 5 locations (locations 0, 1, 2, 3 and 4) in the sorted array are 0.

However, if possible we want to avoid using a nested loop or an if statement when we place items

New Idea:

If we have the freq of the number of values <= any value in the array we can place the current value in index position one less than this and decrement the freq

For example, if we come across a 0 in the original array and know that there are 5 values <= 0, then the 0 we are at can be placed in position 4 and decrement freq to 4

Next time we come across a 0 in the original array we see that there are now 4 values <= 0, and we can place the next 0 in position 3

EXAMPLE:

Given arrA[] = `{1, 2, 1, 3, 2, 4, 6, 1, 1, 6};`

freq[]

0	0	0	0	0	0	0
0	1	2	3	4	5	6

freq[]

0	4	2	1	1	0	2
0	1	2	3	4	5	6

Number of values <= 0 is **0**

Number of values <= 1 is **4** = freq[1] + freq[0] and update freq[1]

Number of values <= 2 is **6** = freq[2] + freq[1] and update freq[2]

Number of values <= 3 is **7** = freq[2] + freq[1] and update freq[3]

etc.

freq[]

0	4	6	7	8	8	10
0	1	2	3	4	5	6

HOW CAN WE USE THIS TO SORT?

Given arrA[10] = {1, 2, 1, 3, 2, 4, 6, 1, 1, 6};

freq[]

0	4	6	7	8	8	10
0	1	2	3	4	5	6

arrB[]

0	1	2	3	4	5	6	7	8	9

Starting at index 0 in arrA[] and traversing to end of arrA[]:

arrA[0] = 1, Number of values <= 1 is **4** so arrB[3] = 1 and --freq[1], it is now 3

arrA[1] = 2, Number of values <= 2 is **6** so arrB[5] = 2 and --freq[2], it is now 5

arrA[2] = 1, Number of values <= 1 is **?** so arrB[?] = 1 and --freq[1], it is now ?

arrA[3] = 3, Number of values <= 3 is **7** so arrB[6] = 3 and --freq[3], it is now 6

arrA[4] = 2, Number of values <= 2 is **?** so arrB[?] = 2 and --freq[2], it is now ?

WHAT IS HAPPENING AT EACH STAGE?

Given arrA[10] = {1, 2, 1, 3, 2, 4, 6, 1, 1, 6};

freq[]

0	1	2	3	4	5	6
0	4	6	7	8	8	10

arrB[]

0	1	2	3	4	5	6	7	8	9

```
value = arrA[i];              //value to sort
count = freq[value];          //<= freq of value
arrB[count - 1] = value;      //place in arrB
--freq[value];                //update <= count
```

FINDING CORRECT POSITION FOR EACH VALUE:

```
int arrA[10] = {1, 2, 1, 3, 2, 4, 6, 1, 1, 6};
```

freq[]

0	4	6	7	8	8	10
0	1	2	3	4	5	6

arrB[]

			1						
0	1	2	3	4	5	6	7	8	9

When i = 0:

`value = arrA[i]`: value = arrA[0] = 1

`count = freq[value]`: count = freq[1] = 4

`arrB[count-1] = value`: arrB[3] = 1

`--freq[value]`: freq[1] is 3

freq[]

0	3	6	7	8	8	10
0	1	2	3	4	5	6

FINDING CORRECT POSITION FOR EACH VALUE:

```
int arrA[10] = {1, 2, 1, 3, 2, 4, 6, 1, 1, 6};
```

freq[]

0	3	6	7	8	8	10
0	1	2	3	4	5	6

arrB[]

			1		2				
0	1	2	3	4	5	6	7	8	9

When i = 1:

`value = arrA[1];` value = arrA[1] = 2

`count = freq[value];` count = freq[2] = 6

`arrB[count-1] = value`: arrB[5] = 2

`--freq[value];` freq[2] is 5

freq[]

0	3	5	7	8	8	10
0	1	2	3	4	5	6

FINDING CORRECT POSITION FOR EACH VALUE:

```
int arrA[10] = {1, 2, 1, 3, 2, 4, 6, 1, 1, 6};
```

freq[]

0	3	5	7	8	8	10
0	1	2	3	4	5	6

arrB[]

		1	1		2				
0	1	2	3	4	5	6	7	8	9

When i = 2:

`value = arrA[2];` value = arrA[2] = 1

`count = freq[value];` count = freq[1] = 3

`arrB[count-1] = value`: arrB[2] = 1

`--freq[value]`: freq[1] is 2

freq[]

0	2	5	7	8	8	10
0	1	2	3	4	5	6

COUNT SORT:
Inputs and Outputs

Inputs:

Array arrA[] of size N with n positive integer values in the range 0 - k.

Outputs:

Array arrB[] (same size as arrA[]) to hold the sorted values (originally empty).
*(** Note need this additional array to hold the sorted data)*

Assumptions:

(Strong!) assumption: integer data

Initially we will assume the data is positive but negative integers can also be easily sorted with a small adjustment/offset

Count Sort Steps:

1. Find `maxVal` as before, if not given
2. Create `freq[maxVal+1]` to hold count of each number in `arrA[]` as before
3. [new] Modify `freq[]` to hold the number of elements which are less than or equal to each value arrA[i]. For i >= 1:

   ```
   freq[i] = freq[i] + freq[i-1]
   ```

4. Sort: For i from 0 to end of `arrA[]`:

   ```
   value = arrA[i];              //value to sort
   count = freq[value];          //<= freq of value
   arrB[count-1] = value;        //place in arrB
   --freq[value];                //update counts
   ```

C CODE FRAGMENTS

```
//count
for(i = 0; i < size; i++) {
    ++freq[ arrA[i] ];
}
//get <= in freq[]
for(i = 1; i < freqSize; i++) {
    freq[i] = freq[i] + freq[i - 1];
}
// place values from arrA into arrB; update freq[]
for(i = 0; i < size; i++) {
   value = arrA[i];         //value to sort
   count = freq[value];     //<= freq of value
   arrB[count-1] = value;   //place value in arrB
   --freq[value];           //decrement freq[]
} //next value in arrA
```

NOTE:

no need for extra variables `count` and `value` but they increase readability

Alternative:

```
// place values from arrA into arrB; update freq[]
for(i = 0; i < size; i++) {
    arrB[ freq[ arrA[i] ] - 1] = arrA[i];
    --freq[arrA[i]];
}
```

FINAL STEP: Write back values to `arrA[]`

Remember arrA[] remains unsorted up to this point and arrB[] contains the sorted data.

```
//write back sorted values to arrA[] now that sorting is finished
for (i = 0; i < size; i++) {
    arrA[i] = arrB[i];
}
```

void countSort(int[], int, int);

```
27
28  // sort an array of integers using countSort algorithm from 0 to maxVal
29  void countSort(int arrA[], int size, int maxVal)
30  {
31
32      int i, value, count;
33      int *freq, *arrB;
34      int freqSize;
35
36      freqSize = maxVal + 1;
37
38      freq = (int*)malloc(freqSize * sizeof(int));   //create freq[]
39      arrB = (int*)malloc(size * sizeof(int));       //create arrB[] same size as arrA[]
40
41      // initialise freq[]
42      for (i = 0; i < freqSize; i++) {
43          freq[i] = 0;
44      }
45
46      // count
47      for (i = 0; i < size; i++) {
48          ++freq[ arrA[i] ];
49      }
50
51      // get <= in freq[]
52      for (i = 1; i < freqSize; i++) {
53          freq[i] = freq[i] + freq[i - 1];
54      }
55
56      // place values from arrA into arrB; update freq[]
57      for (i = 0; i < size; i++) {
58          value = arrA[i];            //value to sort
59          count = freq[value];        //<= freq of value
60          arrB[count - 1] = value;    //place value in arrB
61          --freq[value];              //decrement freq[]
62      } //next value in arrA
63
64      //write back sorted values to arrA[] now that sorting finished
65      for (i = 0; i < size; i++) {
66          arrA[i] = arrB[i];
67      }
68  }
69
```

TIME STEP ANALYSIS:

Let N = size Let K = freqSize (maxVal+1)

Line	Cost	Num Times	Cost*Num Times
32-39	6	1	6
42	1	K+1	K+1
43 (initialise freq)	1	K	K
47	1	N+1	N+1
48 (count)	1	N	N
52	1	K	K
53 (calculate <=)	1	K-1	K-1
57	1	N+1	N+1
58-61	4	N	4N
65	1	N+1	N+1
66 (write back)	1	N	N
		TOTAL	**9N + 4K + 9**
		Big-O	**O(N + K)**

```
28  // sort an array of integers using countSort algorithm from 0 to maxVal
29  void countSort(int arrA[], int size, int maxVal)
30  {
31
32      int i, value, count;
33      int *freq, *arrB;
34      int freqSize;
35
36      freqSize = maxVal + 1;
37
38      freq = (int*)malloc(freqSize * sizeof(int));   //create freq[]
39      arrB = (int*)malloc(size * sizeof(int));       //create arrB[] same size as arrA[]
40
41      // initialise freq[]
42      for (i = 0; i < freqSize; i++) {
43          freq[i] = 0;
44      }
45
46      // count
47      for (i = 0; i < size; i++) {
48          ++freq[ arrA[i] ];
49      }
50
51      // get <= in freq[]
52      for (i = 1; i < freqSize; i++) {
53          freq[i] = freq[i] + freq[i - 1];
54      }
55
56      // place values from arrA into arrB; update freq[]
57      for (i = 0; i < size; i++) {
58          value = arrA[i];            //value to sort
59          count = freq[value];        //<= freq of value
60          arrB[count - 1] = value;    //place value in arrB
61          --freq[value];              //decrement freq[]
62      } //next value in arrA
63
64      //write back sorted values to arrA[] now that sorting finished
65      for (i = 0; i < size; i++) {
66          arrA[i] = arrB[i];
67      }
68  }
```

Dependent on size of original array and maxVal – the smaller maxVal the better

Time taken?
Number of comparisons and swaps?

```
freqsize is 127
For Count Sort with 1000 values sorted time taken is 0.000000
```

Will be quick relative to other Sorting techniques

Number of comparisons and swaps?

```
1000 items have been read
In Insertion Sort:
time taken is: 0.002000 seconds;
number of swaps is: 257335;
number of comparisons is: 258334
```

```
1000 items have been read
In Selection Sort:
time taken is 0.003000;
number of swaps is 999;
number of comparisons is 499500
```

```
1000 items have been read
In Bubble Sort:
time taken is 0.007000;
number of swaps is 257335;
number of comparisons is 499500
```

ADDITIONAL QUESTIONS/WORK

1. Consider how you might modify the algorithm to count both positive and negative integers.
2. Add additional code to test countSort() with a larger file
3. Add additional code to check the actual running time of countSort()
4. Are there any benefits to be gained if the data is already sorted or partially sorted?

COUNT SORT FOR NEGATIVE INTEGERS

To w rk with negative integers we need to know both the minVal and the maxVal.

The size of freq will be maxVal-(minVal)+1

freq[0] will store the count of the minVal

freq[freqSize-1] will store the count of the maxVal

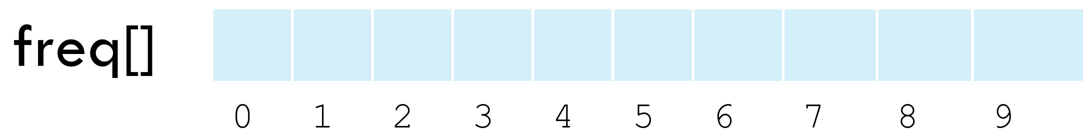
For example, if the minVal is -5 and the maxVal is 4 (i.e., the range of integers is [-5, 4] then:

freqSize = 4-(-5)+1 = 10

freq[0] stores the count of -5

…..

freq[9] stores the count of 4



CHANGES NEEDED TO CODE:

```
// declaring freqSize
freqSize = maxVal - minVal + 1;

//counting
for(i = 0; i < size; i++) {
    ++freq[arrA[i] - minVal];
}
//sorting and updating freq
   arrB[freq[arrA[i] - minVal] - 1] = arrA[i];
   --freq[arrA[i] - minVal];
```

PUTTING IT ALL TOGETHER? … YOU TRY ….

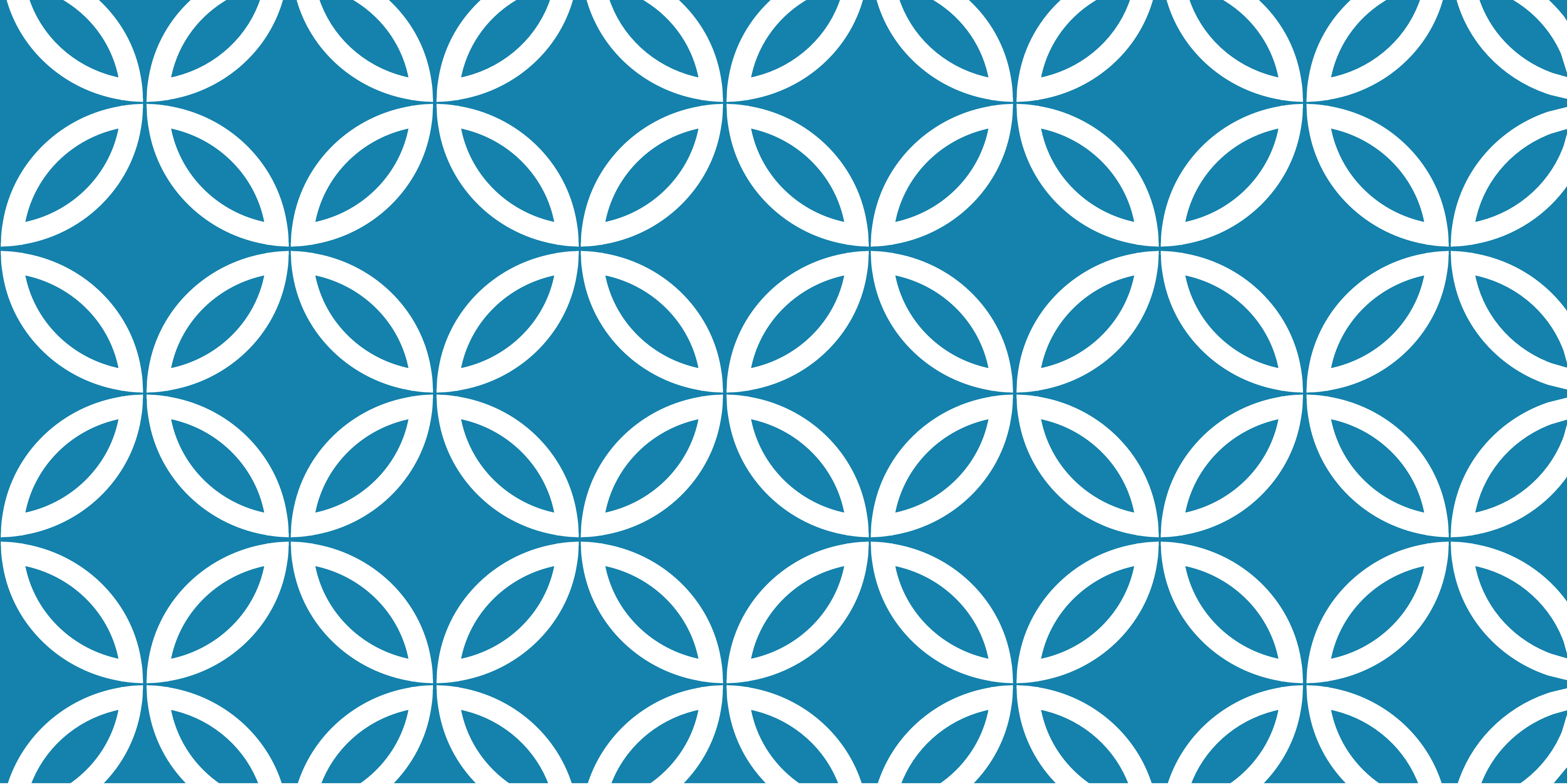
Summary

Count Sort is the best Sorting technique we have seen so far – linear order complexity - O(N+K)

But … can only be used with integer data and need to know maxVal (and minVal potentially) – all other approaches can be used with any data type and do not need to know range of data in advance.

Not a *comparison* algorithm like the other three sorting algorithms we have considered.

Not an *in-place* algorithm like the other three sorting algorithms we have considered - Uses two extra arrays – one to hold the frequencies and a second to hold the sorted data.



TOPIC: PROBLEM SOLVING WITH RECURSION

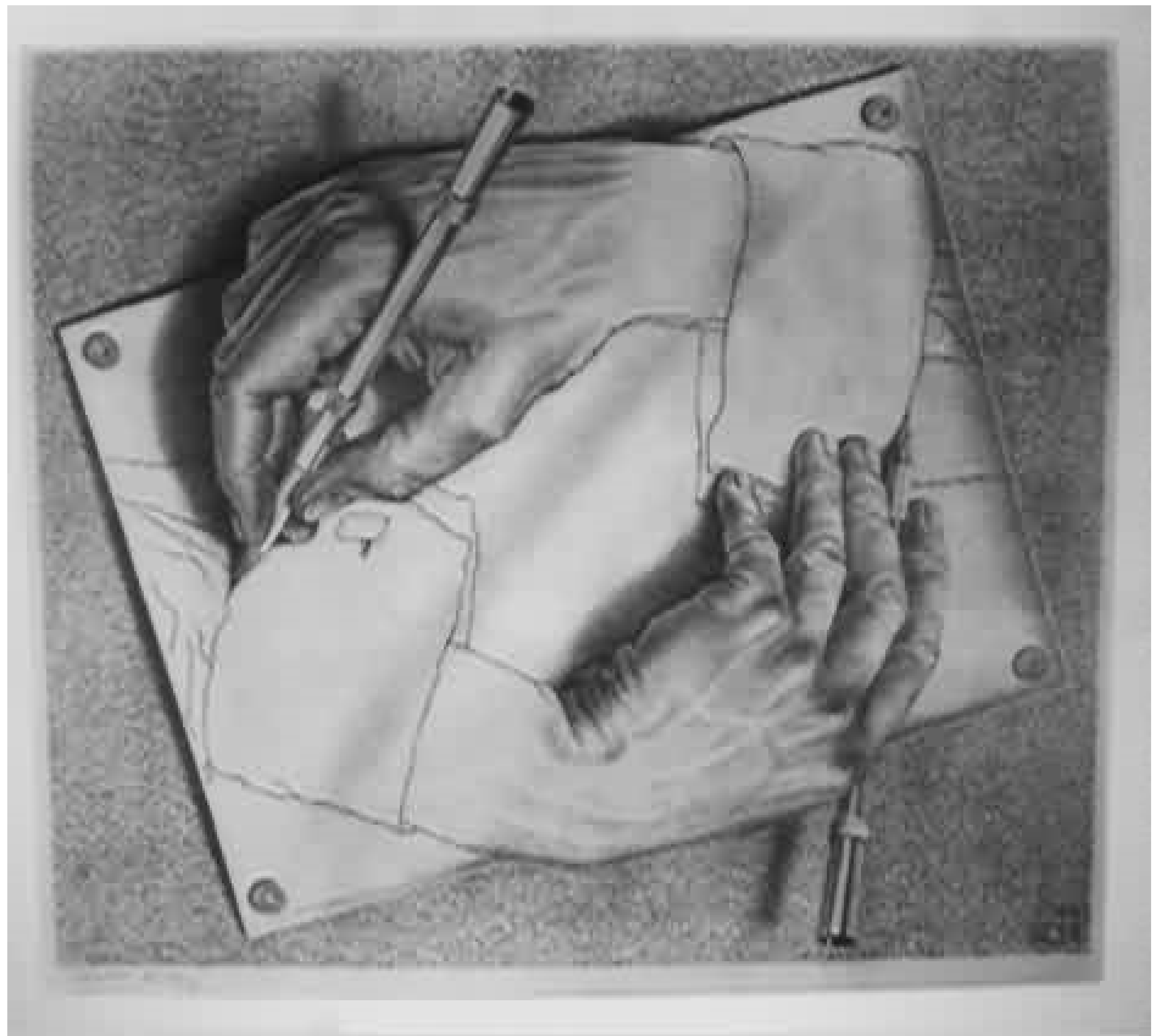
CT102 Algorithms

RECURSION

Recursion means a
reference to itself

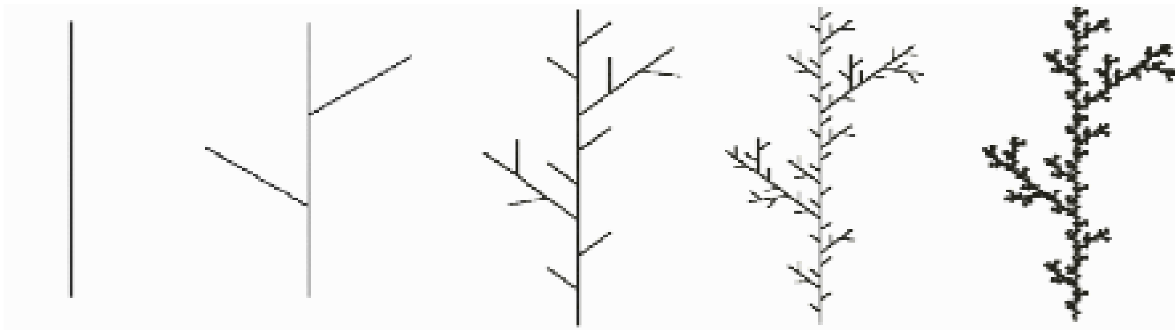
Droste
HAARLEM-HOLLAND
cacao
Netto 250 g

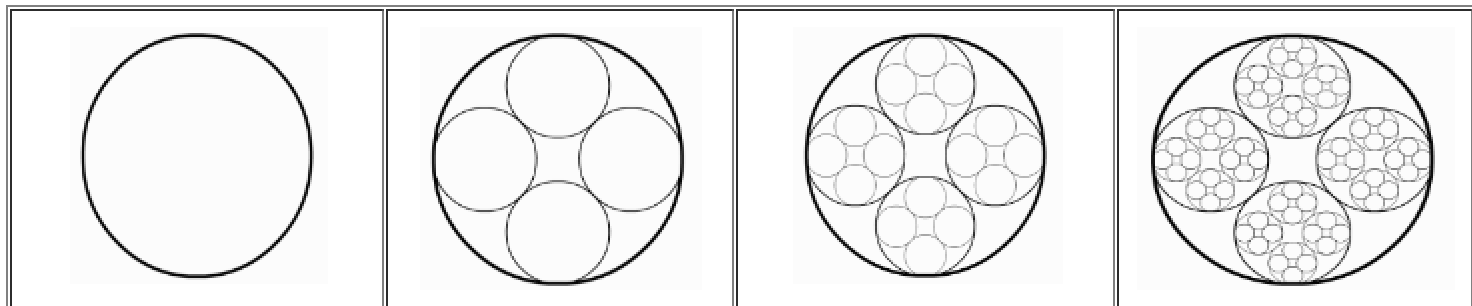
“Reference to itself”



Drawing Hands
Esher, 1948

“Reference to itself”





From: http://symmetry-us.com/Journals/bridges2005/burns/index.html

RECURSION* – Computing Definition:

Functions (and procedures) whose definition involves a reference to themselves (a call to themselves)

Specifically:

A recursive function A is one that calls itself or calls another function which calls function A

* I know recursion has been introduced already in CT103!

RECURSION IN COMPUTING

- We use function calls and recursion instead of loops and iteration to solve problems.
- Some programming languages support recursion better than others.
- Some problems are particularly suited to a recursive solution.
- The general idea is to solve a problem by solving a smaller version of the problem and continue this until we are at a trivial case (a "divide and conquer" approach).

RECALL:
When creating your own function need:

1. Function declaration
2. Function definition
3. Function call

RECALL: FLOW OF CONTROL

When a function is called, the program control is transferred to the called function.

A called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns the program control back to the *calling* environment.

RECURSIVE PROBLEM 1:

```
18
19  void puzzle(int);
20
21  void main(){
22      int number;
23      printf("\n Enter a number to test");
24      scanf("%d", &number);
25      puzzle(number);
26  }
27
28  void puzzle(int num){
29      printf("\n num ia now %d", num);
30      puzzle(num - 1);
31
32  }
33
```

1st call to function puzzle()

Function puzzle() calling puzzle()

“recursive on all control paths”

⚠ 2 warning C4717: 'puzzle' : recursive on all control paths, function will cause runtime stack overflow

Problem:

- Will never stop because there is no stopping condition
- “Runtime stack overflow”
- We say such a function is **not** *well defined*

```
18
19  void puzzle(int);
20
21  void main(){
22      int number;
23      printf("\n Enter a number to test");
24      scanf("%d", &number);
25      puzzle(number);
26  }
27
28  void puzzle(int num){
29      printf("\n num ia now %d", num);
30      puzzle(num - 1);
31
32  }
33
```

WELL DEFINED RECURSIVE FUNCTIONS

As with iteration, must ensure that a recursive function will not continue to run indefinitely.

If using recursion we must ensure that:

- There are certain criteria, called **base criteria**, for which the function does not call itself (stopping conditions)
- Each time the function does call itself (directly or indirectly), it must be **closer** to the base criteria.

A recursive function with these two properties is said to be **well defined**.

WELL-DEFINED:

Base Case: Must have a condition for which the function will not call itself (and will stop and usually give a result)

Reduce: For each recursive call, must move towards the base case (reduce)

ACTIVATIONS

With recursion, we often have the illusion of multiple copies of the function existing. These are referred to as **activations**. These activations appear and disappear as the program advances.

A number of activations may exist at the same time. However, of the activations existing at any given time, only one is actively progressing. The others are effectively in limbo, each waiting for another activation to terminate before it can terminate.

RUN TIME STACK

A stack data structure is used to keep track of activations – the data structure restricts where insertions and deletions can take place in a *Last In First Out* (LIFO) manner.

The function on top of the stack is the current active one.

When the current active function completes it is popped off stack (deleted) and activation moves to next function on the stack.

Each activation has its own environment with its own set of values for variables (local scope).

RECURRENCE/RECURSIVE TREE

A diagram which visualises the recursive calls and the work done for each recursive call and allows timestep analysis

Creating a well-defined version of `puzzle();`

1. add a stopping condition

2. add return statements

```
34
35  int puzzle(int num) {
36
37      printf("\n num is now %d", num);
38
39      if (num <= 0) {
40          return(0);
41      }
42      else {
43          return( puzzle(num - 1) );
44      }
45  }
46
47
```

Counting how many times `puzzle()` is called at line 43 for any on-zero num? Let n = num

Go to **www.menti.com**

use the code **3481 2905**

```
34
35  int puzzle(int num) {
36
37      printf("\n num is now %d", num);
38
39      if (num <= 0) {
40          return(0);
41      }
42      else {
43          return( puzzle(num - 1) );
44      }
45  }
46
47
```

PROBLEM 2: A new function passed an integer array and its size

Is function test() well defined?

Go to www.menti.com

use the code **3481 2905**

```
99
100  int test(int arrA[], int size) {
101
102      if (size == 1) {
103          return( arrA[0] );
104      }
105      else {
106          return( arrA[size - 1] + test(arrA, size - 1) );
107      }
108  }
```

PROBLEM 2: test()

What does the function test() do?

e.g., Check with: `test(A,5)` as given in `main()`

```
 99
100  int test(int arrA[], int size) {
101
102      if (size == 1) {
103          return( arrA[0] );
104      }
105      else {
106          return( arrA[size - 1] + test(arrA, size - 1) );
107      }
108  }
```

```
18
19  void main(void){
20
21      // testing _test()_
22      int ans;
23      int A[5] = {2, 4, 6, 8, 10};
24      ans = test(A, 5);
25
26
```

Go to www.menti.com

use the code 3481 2905

Example 2: Analysis

- Run time Stack with `test(A,5)`

```
99
100  int test(int arrA[], int size) {
101
102      if (size == 1) {
103          return( arrA[0] );
104      }
105      else {
106          return( arrA[size - 1] + test(arrA, size - 1) );
107      }
108  }
```

```
18
19  void main(void){
20
21      // testing _test()_
22      int ans;
23      int A[5] = {2, 4, 6, 8, 10};
24      ans = test(A, 5);
25
26
```

GENERAL APPROACH TO SOLVING PROBLEMS RECURSIVELY …

1. What is the base case?
2. What should the answer be when we are at the base case?
3. How do you reduce to get to this base case?
4. What other work needs to be done for each function call?
5. How can these steps be put together?

GENERAL STRUCTURE

Usually an if/else structure or if/else if/else:

```
if (base case is true)
 return  //stop recursion
else
  reduce to base case and solve problem
```

```
if (base case 1 is true)
  return
else if (condition is true)
  return or reduce to base case
else
  reduce to base case
```

MISTAKES TO AVOID:

- Wrong number of arguments passed to function … must match function declaration at all times
- Not having a base case
- Not reducing to the base case

PROBLEM 3 … already seen in ct103

- Write a recursive function which finds the factorial of a number n

Recall:

- The factorial of a non-negative integer *n* is the product of all integers less than or equal to *n*.
- The factorial of 0 is 1 and factorial of 1 is 1

Steps for Factorial
int factorial(int n)

Base case: `n <= 1` return 1

Reduce: `return(n * factorial(n - 1));`

```
int factorial(int n) {

    if(        ) {
        return(   );
    }
    else {
        return(        );
    }
}
```

factorial()

```
177
178  int factorial (int n) {
179
180      if(n <= 1) {
181          return(1);
182      }
183      else {
184          return( n * factorial(n - 1) );
185      }
186  }
187
```

TIME STEP ANALYSIS

```
177
178 int factorial (int n) {
179
180     if(n <= 1) {
181         return(1);
182     }
183     else {
184         return( n * factorial(n - 1) );
185     }
186 }
187
```

Line	Cost	Num Times	Cost*Num Times	Total
180	1	n	n	
181	1	1	1	
184	1 or 2?	n-1	2n – 2	
				3n - 1
				O(n)

Note: The issue with factorial is the limit in terms of the factorial of n being stored (even using the maximum size int possible)

Try test yourself to see what is the max n you can find factorial for:

```
int long long ans;

ans = factorial(number);
```

PROBLEM 4: Fibonacci Sequence ... already seen in CT103

Famous sequence (from 13th Century!) whose numbers grow very large, very quickly

For some function fib(n) that finds the nth Fibonacci number, the function can be defined recursively as:

```
fib(0) = 0
fib(1) = 1
fib(n) = fib(n-2) + fib(n-1) for n > 2
```

```
215
216   //fibonacci number of nth term - recursive
217   int fib (int n) {
218
219       if (n <= 1) {
220           return(n);
221       }
222       else {
223           return( fib(n - 1) + fib(n - 2));
224       }
225   }
226
```

WHY IS THIS AN INEFFICIENT SOLUTION?

Go to **www.menti.com**

use the code **3481 2905**

```
215
216  //fibonacci number of nth term - recursive
217  int fib (int n) {
218
219      if (n <= 1) {
220          return(n);
221      }
222      else {
223          return( fib(n - 1) + fib(n - 2));
224      }
225  }
226
```

TIME STEP ANALYSIS

```
215
216 //fibonacci number of nth term - recursive
217 int fib (int n) {
218
219     if (n <= 1) {
220         return(n);
221     }
222     else {
223         return( fib(n - 1) + fib(n - 2));
224     }
225 }
226
```

Line	Cost	Num Times	Cost*Num Times	Total
219	1	?		
220	1	?		
223	2	?		
				O(?)

CONSIDERING LINE 223 IN MORE DETAIL

```
215
216  //fibonacci number of nth term - recursive
217  int fib (int n) {
218
219      if (n <= 1) {
220          return(n);
221      }
222      else {
223          return( fib(n - 1) + fib(n - 2));
224      }
225  }
226
```

For each recursive call, there will be 2 function calls fib(n – 1) and fib (n – 2)

Assuming n on entry:

1st recursive call: 2 more function calls $= 2 = 2^1$

2nd recursive calls: each of the 2 previous calls will have 2 calls each $= 4 = 2^2$

3rd recursive calls: each of the 4 previous calls will have 2 calls each $= 8 = 2^3$

4th recursive calls: each of the 8 previous calls will have 2 calls each $= 16 = 2^4$

…

At some stage, $n = 1$ and $n = 0$ for *some parts* of the expansion and there will be 2 less calls before all parts eventually complete

CONSIDERING LINE 223 IN MORE DETAIL

```
215
216  //fibonacci number of nth term - recursive
217  int fib (int n) {
218
219      if (n <= 1) {
220          return(n);
221      }
222      else {
223          return( fib(n - 1) + fib(n - 2));
224      }
225  }
226
```

So:

1st recursive call: 2^1 calls

2nd recursive calls 2^2calls

3rd recursive calls 2^3calls

4th recursive calls 2^4calls

…

So it looks like 2^ncalls but in fact will be less than this

We can say number of calls is always $< \ 2^n$

In fact, instead of 2 we have the golden ratio number for large n 1.618^n

So is O(1.618^n)

PROBLEM 5: LINEAR SEARCH
(Recursive Solution)

Write a recursive function which searches for an item in an array (of unsorted items) returning the position of the first occurrence of the item in the array if it exists or else -1.

Note: we have already seen an iterative version of this

Recursive idea for linear search:

- For each function call, if there are values remaining to check/search:
  - Check if item is at the last position (size-1)
  - If not search again, with sub-array of size one less (size-1)

STEPS:

```
int search(int arrA[], int size, int item)
```

Two Base Cases:

- Nothing left to search: `size == 0` return -1 to indicate not found
- Have found item: `arrA[size - 1] == item` return position which is `size-1`

Reduce:

```
search(arrA, size - 1, item);
```

LINEAR SEARCH

```
69
70  //linear search, starting at size-1. returns -1 if not found
71  int search(int arrA[], int size, int item) {
72
73      if(     ) {
74          return(    );
75      }
76      else if (     ) {
77          return(    );
78      }
79      else {
80          return(       );
81      }
82  }
```

LINEAR SEARCH – CLASS WORK

```
69
70  //linear search, starting at size-1. returns -1 if not found
71  int search(int arrA[], int size, int item) {
72
73      if (size == 0) {
74          return (-1);
75      }
76      else if (arrA[size - 1] == item) {
77          return(size - 1);
78      }
79      else {
80          return( search(arrA, size - 1, item) );
81      }
82  }
83
```

TIME STEP ANALYSIS OF LINEAR SEARCH

```
69
70  //linear search, starting at size-1. returns -1 if not found
71  int search(int arrA[], int size, int item) {
72
73      if (size == 0) {
74          return (-1);
75      }
76      else if (arrA[size - 1] == item) {
77          return(size - 1);
78      }
79      else {
80          return( search(arrA, size - 1, item) );
81      }
82  }
83
```

Worst case?

N = ?

Line	Cost	Num Times	Cost*Num Times	Total
73	1	N+1		
74	1	1		
76	1	N		
77 In worst case won't happen				
80	1	N		
				3N+2

IS BIG-O OF RECURSIVE LINEAR SEARCH DIFFERENT TO THAT OF ITERATIVE LINEAR SEARCH?

Go to www.menti.com

use the code 3481 2905

PROBLEM 6: BINARY SEARCH

RECURSIVE VERSION

```
137
138 // Searching for location of item in sorted array
139 // Must be called with begSec= 0 and endSec = size-1
140 int binarySearch(int arrA[], int begSec, int endSec, int item) {
141
142     int mid = int((begSec + endSec)/2);
143
144     if  (begSec > endSec) {
145         return(-1);
146     }
147     else if (arrA[mid] == item) {
148         return mid;
149     }
150     else if (item < arrA[mid]) {
151         return(binarySearch(arrA, begSec, mid - 1, item));
152     }
153     else {
154         return(binarySearch(arrA, mid + 1, endSec, item));
155     }
156 }
157
```

BINARY SEARCH TIME STEP ANALYSIS

```
137
138 // Searching for location of item in sorted array
139 // Must be called with begSec= 0 and endSec = size-1
140 int binarySearch(int arrA[], int begSec, int endSec, int item) {
141
142     int mid = int((begSec + endSec)/2);
143
144     if  (begSec > endSec) {
145         return(-1);
146     }
147     else if (arrA[mid] == item) {
148         return mid;
149     }
150     else if (item < arrA[mid]) {
151         return(binarySearch(arrA, begSec, mid - 1, item));
152     }
153     else {
154         return(binarySearch(arrA, mid + 1, endSec, item));
155     }
156 }
157
```

How does it differ to the iterative version?

Would you expect the Big-O of the recursive binary search to be different to the iterative binary search?

Can you remember what is the Big-O time complexity of the iterative binary search?

LINEAR AND BINARY SEARCH TIME STEP ANALYSIS

```
137
138 // Searching for location of item in sorted array
139 // Must be called with begSec= 0 and endSec = size-1
140 int binarySearch(int arrA[], int begSec, int endSec, int item) {
141
142     int mid = int((begSec + endSec)/2);
143
144     if  (begSec > endSec) {
145         return(-1);
146     }
147     else if (arrA[mid] == item) {
148         return mid;
149     }
150     else if (item < arrA[mid]) {
151         return(binarySearch(arrA, begSec, mid - 1, item));
152     }
153     else {
154         return(binarySearch(arrA, mid + 1, endSec, item));
155     }
156 }
157
```

Same questions:

What is worst case situation? … item not in array

How much of the array needs to be searched in order to find this out? …

- For linear search … all n values
- For binary search … approx. log_2 n values

What is best case situation? … item found after first comparison (wherever that happens to be)

Problem 7: Variation of Binary Search …

What is happening?

Assuming an array of sorted, unique values

```
199
200
201 // Searching for location of item in sorted array
202 // Must be called with begSec= 0 and endSec = size-1
203 int anotherSearch(int arrA[], int begSec, int endSec, int item) {
204     int mid1 = (begSec + int(endSec-begSec)/3);
205     int mid2 = (endSec - int(endSec-begSec)/3);
206
207     if  (begSec > endSec) {
208         return(-1);
209     }
210     else if (arrA[mid1] == item) {
211         return mid1;
212     }
213     else if (arrA[mid2] == item) {
214         return mid2;
215     }
216     else if (item < arrA[mid1]) {
217         return(anotherSearch(arrA, begSec, mid1-1, item));
218     }
219     else if (item > arrA[mid2]) {
220         return(anotherSearch(arrA, mid2 + 1, endSec, item));
221     }
222     else {
223         return(anotherSearch(arrA, mid1 + 1, mid2 - 1, item));
224     }
225 }
226
```

Problem 7:
What is happening?

```
199
200
201 // Searching for location of item in sorted array
202 // Must be called with begSec= 0 and endSec = size-1
203 int anotherSearch(int arrA[], int begSec, int endSec, int item) {
204     int mid1 = (begSec + int(endSec-begSec)/3);
205     int mid2 = (endSec - int(endSec-begSec)/3);
206
207     if  (begSec > endSec) {
208         return(-1);
209     }
210     else if (arrA[mid1] == item) {
211         return mid1;
212     }
213     else if (arrA[mid2] == item) {
214         return mid2;
215     }
216     else if (item < arrA[mid1]) {
217         return(anotherSearch(arrA, begSec, mid1-1, item));
218     }
219     else if (item > arrA[mid2]) {
220         return(anotherSearch(arrA, mid2 + 1, endSec, item));
221     }
222     else {
223         return(anotherSearch(arrA, mid1 + 1, mid2 - 1, item));
224     }
225 }
226
```

A ternary Search

Considering "thirds" of the array rather than "halves"

Have two midpoints, and three areas where item might be for each search.

Complexity is O(log_3n) but note that we have more comparisons - extra check for equality, extra check to find correct portion to search again

RECALL PROBLEM 2 AGAIN:

```
47
48
49  int test(int arrA[], int size){
50
51      if(size == 1){
52          return(arrA[0]);
53      }
54      else{
55          return(arrA[size-1] + test(arrA, size-1));
56      }
57  }
58
59
```

```
18
19  void main(void){
20
21      // testing _test()_
22      int ans;
23      int A[5] = {2, 4, 6, 8, 10};
24      ans = test(A, 5);
25
26
```

PROBLEM 2 ALTERNATIVE VERSION ….
What's the difference?

Assume `tempsum` has value 0 when function first called

```
154
155  // sum of numbers in an integer array of size
156  int sum (int arrA[], int size, int tempsum) {
157
158      if (size == 0) {
159          return (tempsum);
160      }
161      else {
162          return (sum ( arrA, size - 1, tempsum + arrA[size - 1] ));
163      }
164  }
```

```
47
48
49  int test(int arrA[], int size){
50
51      if(size == 1){
52          return(arrA[0]);
53      }
54      else{
55          return(arrA[size-1] + test(arrA, size-1));
56      }
57  }
58
59
```

Call with `test(A, 5, 0);`

`A[5] = {2, 4, 6, 8, 10};`

```
154
155  // sum of numbers in an integer array of size
156  int sum (int arrA[], int size, int tempsum) {
157
158      if (size == 0) {
159          return (tempsum);
160      }
161      else {
162          return (sum ( arrA, size - 1, tempsum + arrA[size - 1] ));
163      }
164  }
```

What is happening?

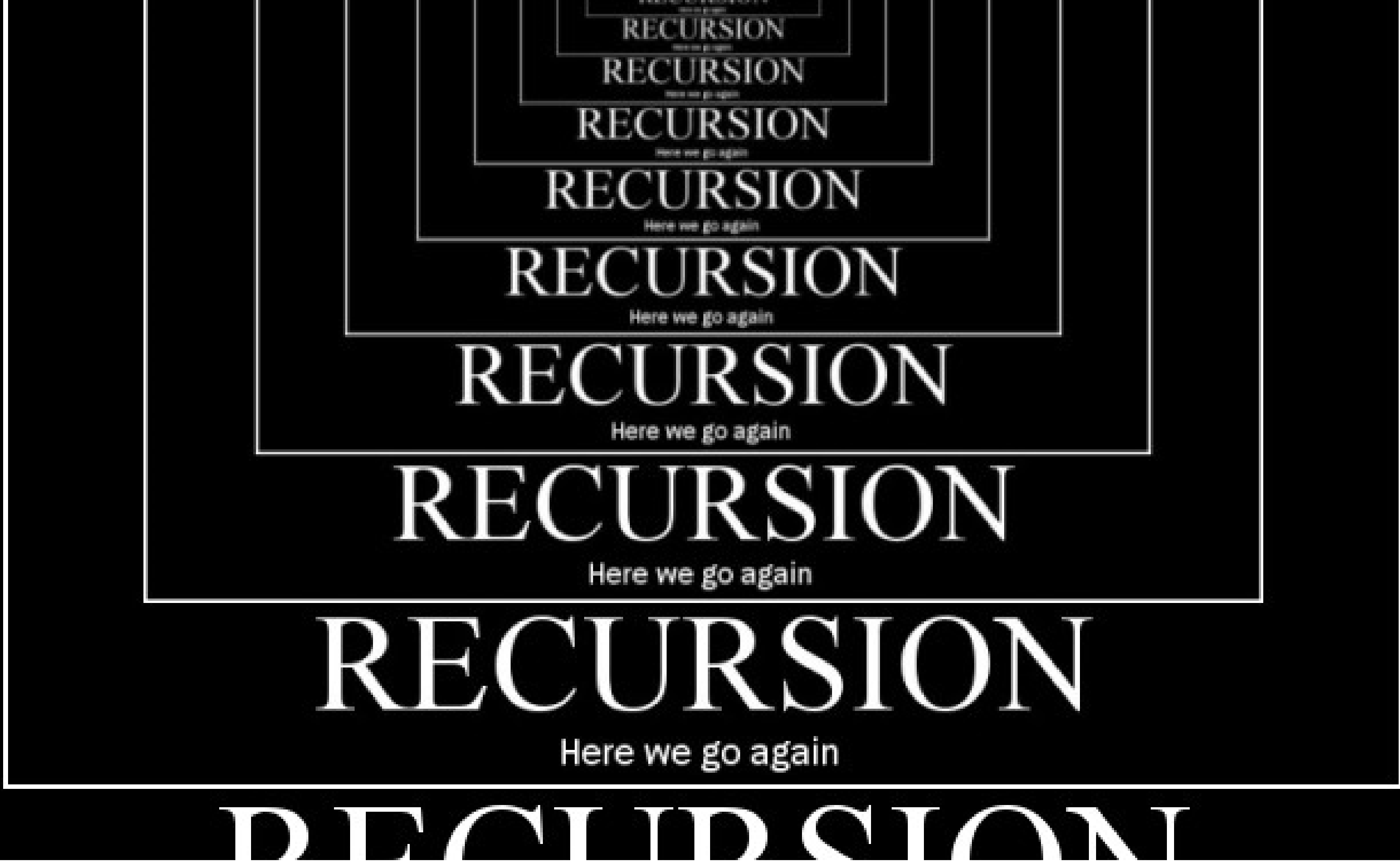
There is no "work left to do" for waiting activations/functions – each recursive call sends the temporary result as part of the recursive call

SUMMARY

Recursion allows us an “easy” was to solve problems by a “divide and conquer” approach.
Recursive solutions may not always be the most efficient solutions however.

Some programming languages offer better support for recursive solutions – C is not one of those languages!

We will continue with recursion when we consider merge sort and quick sort, two sorting algorithms which can be expressed very succinctly when using recursion in comparison to the iterative versions.



RECURSION & MERGE SORT

CT102 Algorithms

Recall: RECURSION IN COMPUTING

- We use function calls and recursion instead of loops and iteration to solve problems.
- Some programming languages support recursion better than others.
- Some problems are particularly suited to a recursive solution.
- The general idea is to solve a problem by solving a smaller version of the problem and continue this until we are at a trivial case (a “divide and conquer” approach).

WELL-DEFINED:

Base Case: Must have a condition for which the function will not call itself (and will stop and usually give a result)

Reduce: For each recursive call, must move towards the base case (reduce)

PROBLEM 5 LINEAR SEARCH – RECURSIVE SOLUTION

```
69
70  //linear search, starting at size-1. returns -1 if not found
71  int search(int arrA[], int size, int item) {
72
73      if (size == 0) {
74          return (-1);
75      }
76      else if (arrA[size - 1] == item) {
77          return(size - 1);
78      }
79      else {
80          return( search(arrA, size - 1, item) );
81      }
82  }
83
```

PROBLEM 6: BINARY SEARCH
RECURSIVE VERSION

```
137
138 // Searching for location of item in sorted array
139 // Must be called with begSec= 0 and endSec = size-1
140 int binarySearch(int arrA[], int begSec, int endSec, int item) {
141
142     int mid = int((begSec + endSec)/2);
143
144     if  (begSec > endSec) {
145         return(-1);
146     }
147     else if (arrA[mid] == item) {
148         return mid;
149     }
150     else if (item < arrA[mid]) {
151         return(binarySearch(arrA, begSec, mid - 1, item));
152     }
153     else {
154         return(binarySearch(arrA, mid + 1, endSec, item));
155     }
156 }
157
```

LINEAR AND BINARY SEARCH TIME STEP ANALYSIS

```
137
138 // Searching for location of item in sorted array
139 // Must be called with begSec= 0 and endSec = size-1
140 int binarySearch(int arrA[], int begSec, int endSec, int item) {
141
142     int mid = int((begSec + endSec)/2);
143
144     if  (begSec > endSec) {
145         return(-1);
146     }
147     else if (arrA[mid] == item) {
148         return mid;
149     }
150     else if (item < arrA[mid]) {
151         return(binarySearch(arrA, begSec, mid - 1, item));
152     }
153     else {
154         return(binarySearch(arrA, mid + 1, endSec, item));
155     }
156 }
157
```

Same questions:

What is worst case situation? … item not in array

How much of the array needs to be searched in order to find this out? …

For linear search … all n values

For binary search … approx. log_2 n values

What is best case situation?

Problem 7: Variation of Binary Search …

What is happening?

Assuming an array of sorted, unique values

```
199
200
201 // Searching for location of item in sorted array
202 // Must be called with begSec= 0 and endSec = size-1
203 int anotherSearch(int arrA[], int begSec, int endSec, int item) {
204     int mid1 = (begSec + int(endSec-begSec)/3);
205     int mid2 = (endSec - int(endSec-begSec)/3);
206
207     if  (begSec > endSec) {
208         return(-1);
209     }
210     else if (arrA[mid1] == item) {
211         return mid1;
212     }
213     else if (arrA[mid2] == item) {
214         return mid2;
215     }
216     else if (item < arrA[mid1]) {
217         return(anotherSearch(arrA, begSec, mid1-1, item));
218     }
219     else if (item > arrA[mid2]) {
220         return(anotherSearch(arrA, mid2 + 1, endSec, item));
221     }
222     else {
223         return(anotherSearch(arrA, mid1 + 1, mid2 - 1, item));
224     }
225 }
226
```

Problem 7:
What is happening?

```
199
200
201 // Searching for location of item in sorted array
202 // Must be called with begSec= 0 and endSec = size-1
203 int anotherSearch(int arrA[], int begSec, int endSec, int item) {
204     int mid1 = (begSec + int(endSec-begSec)/3);
205     int mid2 = (endSec - int(endSec-begSec)/3);
206 
207     if  (begSec > endSec) {
208         return(-1);
209     }
210     else if (arrA[mid1] == item) {
211         return mid1;
212     }
213     else if (arrA[mid2] == item) {
214         return mid2;
215     }
216     else if (item < arrA[mid1]) {
217         return(anotherSearch(arrA, begSec, mid1-1, item));
218     }
219     else if (item > arrA[mid2]) {
220         return(anotherSearch(arrA, mid2 + 1, endSec, item));
221     }
222     else {
223         return(anotherSearch(arrA, mid1 + 1, mid2 - 1, item));
224     }
225 }
226
```

A ternary Search

Considering "thirds" of the array rather than "halves"

Have two midpoints, and three areas where item might be for each search.

Complexity is $O(log_2 n)$ but note that we have more comparisons extra check for equality, extra check to find correct portion to search again

RECALL PROBLEM 2 AGAIN:

```
47
48
49  int test(int arrA[], int size){
50
51      if(size == 1){
52          return(arrA[0]);
53      }
54      else{
55          return(arrA[size-1] + test(arrA, size-1));
56      }
57  }
58
59
```

```
18
19  void main(void){
20
21      // testing _test()_
22      int ans;
23      int A[5] = {2, 4, 6, 8, 10};
24      ans = test(A, 5);
25
26
```

PROBLEM 2 ALTERNATIVE VERSION .... What's the difference?

Assume `tempsum` has value 0 when function first called

```
154
155   // sum of numbers in an integer array of size
156   int sum (int arrA[], int size, int tempsum) {
157
158       if (size == 0) {
159           return (tempsum);
160       }
161       else {
162           return (sum ( arrA, size - 1, tempsum + arrA[size - 1] ));
163       }
164   }
```

```
47
48
49  int test(int arrA[], int size){
50
51      if(size == 1){
52          return(arrA[0]);
53      }
54      else{
55          return(arrA[size-1] + test(arrA, size-1));
56      }
57  }
58
59
```

Call with `test(A, 5, 0);`

`A[5] = {2, 4, 6, 8, 10};`

```
154
155  // sum of numbers in an integer array of size
156  int sum (int arrA[], int size, int tempsum) {
157
158      if (size == 0) {
159          return (tempsum);
160      }
161      else {
162          return (sum ( arrA, size - 1, tempsum + arrA[size - 1] ));
163      }
164  }
```

A recursive sorting algorithm: MERGE SORT

- A "divide and conquer" approach to sorting which divides the sorting problem in to smaller and smaller sorting sub-problems; solving the sorting task for the smaller case first before merging back the sorted numbers
- Developed by John von Neumann in 1945

APPROACH:

- Instead of considering full array at one time, consider two sub-arrays with size as equal as possible.
- For each sub-array, consider two further sub-arrays with size as equal as possible.
- Keep considering smaller sub-arrays until you are considering sub-arrays of size 1.
- For each sub-array of size 1 (in sorted order), **merge** back with next sub-array **in sorted order**

INPUT AND OUTPUT

Inputs: Array arrA[] of size integers in unsorted order with:

- lower bound *lb (initially 0)*
- upper bound *ub (initially size – 1)*

Outputs: Array arrA[] of size integers in ascending sorted order

STEPS:

Two main steps:

- Part 1: "dividing": continuously reduce array and sub-arrays until you have sub-arrays of size 1 (trivially in sorted order).
- Part 2: "conquering": continuously merge back sorted sub-arrays in sorted order.

EXAMPLE: PART 1: "dividing"

Keeping splitting, as evenly as possible:

arrA[8]

7	17	25	3	7	10	6	17
0	1	2	3	4	5	6	7

mid = int (0+7)/2 = 3 so separately consider:

7	17	25	3
0	1	2	3

7	10	6	17
4	5	6	7

mid = int (0+3)/2 = 1

mid = int (4+7)/2 = 5

So separately consider:

7	17
0	1

25	3
2	3

So separately consider:

7	10
4	5

6	17
6	7

Finally considering all single values which are trivially sorted (relative to themselves):

7
0

17
1

25
2

3
3

7
4

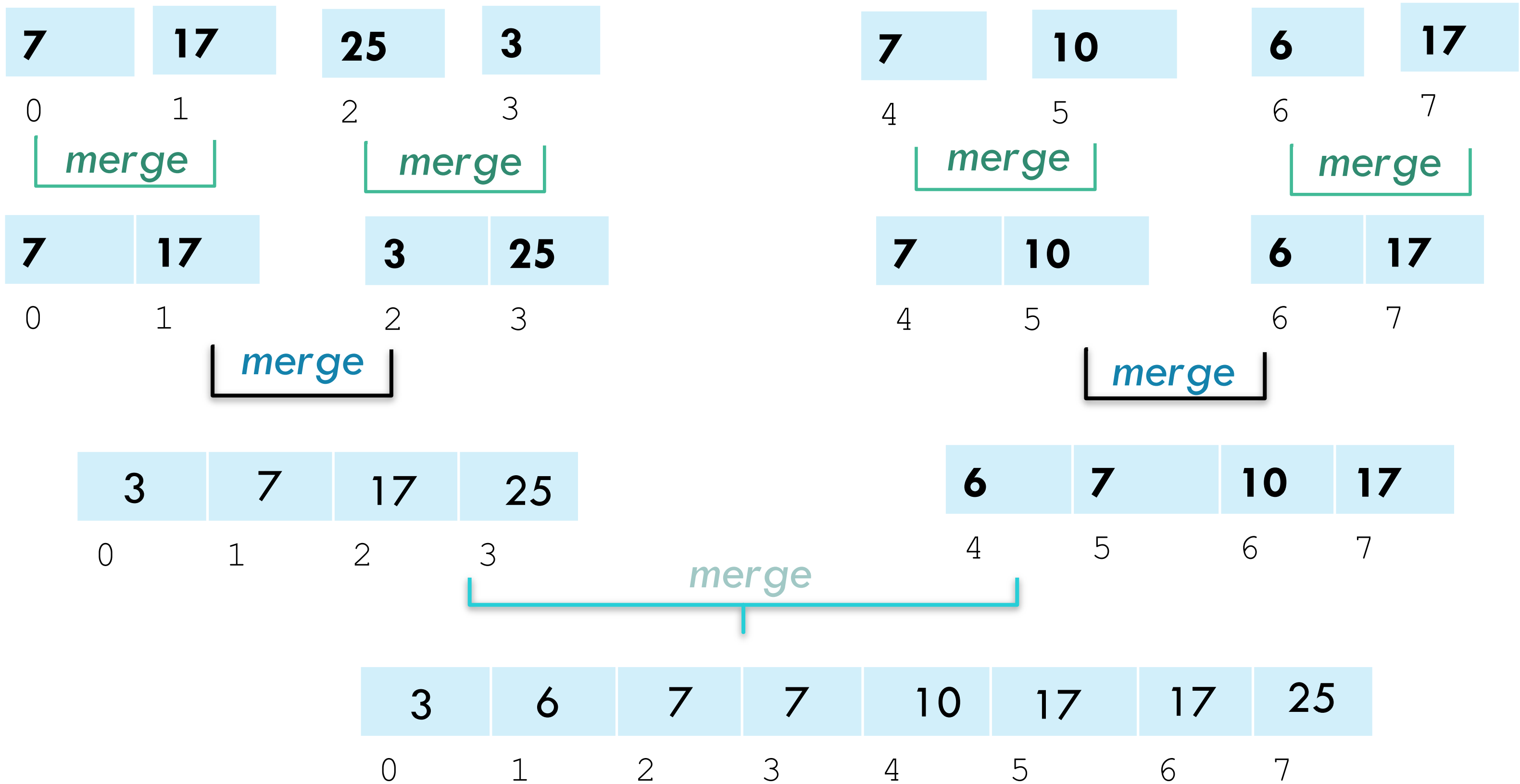
10
5

6
6

17
7

EXAMPLE: PART 2: “conquering”

Recombine in sorted order



FUNCTIONS FOR BOTH STEPS:

- *Dividing*: array, current lower bound and upper bound (to give correct portion of array being considered):

```
mergeSort(int [], int, int);
```

- *Merging*: array, current lower bound, mid and upper bound (to give correct portions of array being merged):

```
merge(int [], int, int, int);
```

mergeSort() an integer array A[]

```
void mergeSort(int [], int, int);
```

```
35  //mergeSort to sort values in an integer array arrA[]
36  // lb = 0 and ub = size - 1 for the first call
37  void mergeSort (int arrA[], int lb, int ub) {
38
39      int mid;
40
41      if (lb < ub) {
42          mid = int((lb + ub) /2);
43          mergeSort (arrA, lb, mid);
44          mergeSort (arrA, mid + 1, ub);
45          merge (arrA, lb, mid, ub);
46      }
47  }
48
```

HOW DOES THIS PROGRESS?

WHAT LINE DOES THE SORTING HAPPEN AT?

```
35  //mergeSort to sort values in an integer array arrA[]
36  // lb = 0 and ub = size - 1 for the first call
37  void mergeSort (int arrA[], int lb, int ub) {
38
39      int mid;
40
41      if (lb < ub) {
42          mid = int((lb + ub) /2);
43          mergeSort (arrA, lb, mid);
44          mergeSort (arrA, mid + 1, ub);
45          merge (arrA, lb, mid, ub);
46      }
47  }
48
```

```
int A[8] = {7,  17,  25,  3,  7,  10,  6, 17};
```

```
void merge (int[], int, int, int);
```

The actual sorting work, "conquering", takes place when merging the sorted sub-arrays.

Have seen an iterative solution to this already which now can be modified …

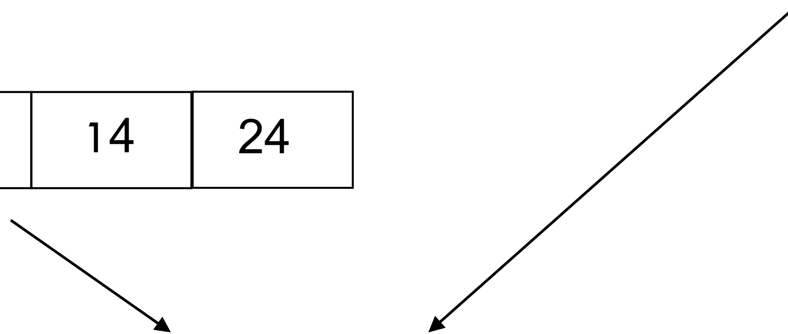
RECALL: Example of merging 2 sorted arrays:

arrB

1	12	17	19	29	44	49

arrA

2	4	12	14	24



arrC

RECALL CODE :

to merge two sorted arrays

```
5
6   // Function to merge the values in 2 integer arrays - not keeping duplicates
7   // Assumes data sorted in arrA[] and arrB[]
8   void merge (int arrA[], int sizeA, int arrB[], int sizeB) {
9
10          int i, j, k;
11          int sizeC;
12
13          i = j = k = 0;
14          sizeC = sizeA + sizeB;
15
16          // declare arrC of size sizeC
17          int *arrC;
18          arrC = (int*) malloc(sizeC * sizeof(int));
19
20          while (i < sizeA && j < sizeB) {
21
22              if (arrA[i] < arrB[j]) {
23                  arrC[k] = arrA[i];
24                  i++;
25                  k++;
26              }
27              else if (arrB[j] < arrA[i]) {
28                  arrC[k] = arrB[j];
29                  j++;
30                  k++;
31              }
32               else if (arrB[j] == arrA[i]) {     // can replce with else
33                  arrC[k] = arrA[i];
34                  i++;
35                  j++;
36                  k++;
37              }
38
39          } //end while
40

41
42          //reached the end of one of the arrays at this point
43          if (i == sizeA) {       // all of arrA written to arrC already
44
45              while(j < sizeB) {
46                  arrC[k] = arrB[j];
47                  j++;
48                  k++;
49              }
50          }
51
52          else if (j == sizeB){   //all of arrB written to arrC already
53
54              while (i < sizeA) {
55                  arrC[k] = arrA[i];
56                  i++;
57                  k++;
58              }
59          }
60          sizeC = k; //correct value of sizeC
61
```

Modifications required to previous merge function

- Both arrays are in fact different parts of one array so only need to pass one array to the function and <u>ensure that the index values are set up correctly:</u>
  - `lb` to `mid`
  - `mid + 1` to `ub`
- This time <u>we will want to keep duplicates</u>
- Note that although we still need an array to store the values after each comparison, we must also write back the contents of the temporary array over the correct range in the original array so that future calls of the `mergeSort()` algorithm will have the correctly sorted sub-arrays.
- Note that although `mergeSort()` is recursive this version of `merge()` is iterative.

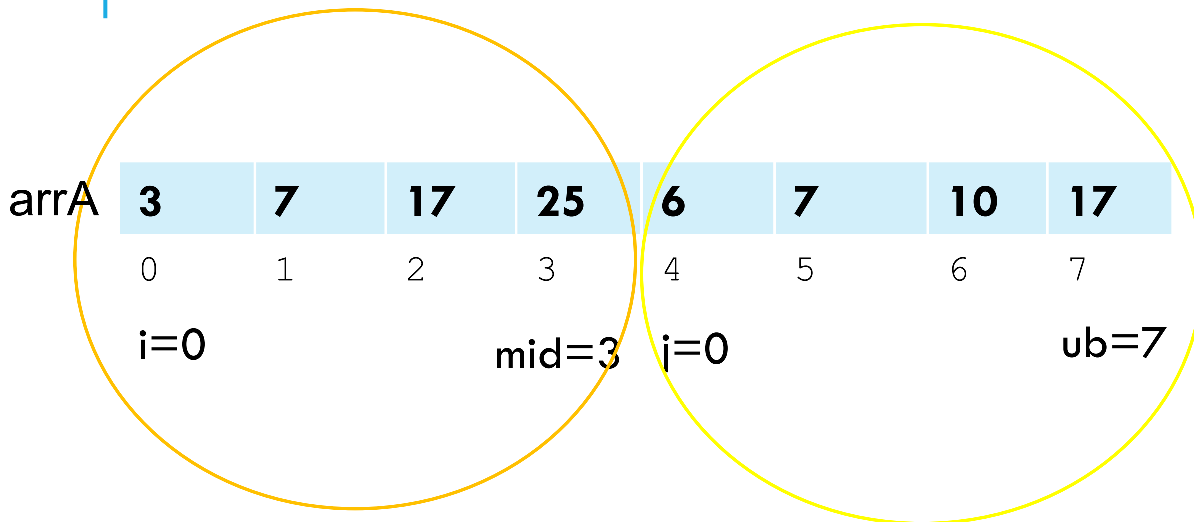
Setting up Indexes and Comparing

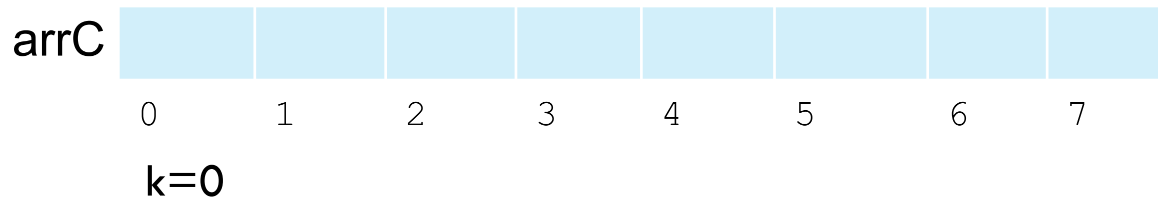
- Create `arrC[]` which should be the same size as the portion of `arrA[]` being merged; index `k` will be used to traverse `arrC[]`
- Initialise indexes to the start of both portions of the array and also correctly initialise k
  - `i = lb;`
  - `j = mid + 1;`
  - `k = 0;`
- The upper bounds (after which loop should stop) are at:
  - `mid` for `i`
  - `ub` for `j`
- At each stage, compare values at `arrA[i]` and `arrA[j]`, moving smaller value into new array `arrC[k]` and updating relevant indexes (`i, j, k`)

Once comparisons have finished ...

- At some stage, will have reached the end of one portion of the array (and all its values will have been copied in order to `arrC[]`).
- At this stage, the remaining values from the second portion of the array can be written straight to `arrC[]` without comparison.
- When finished, the sorted portion must be written back from `arrC[]` to `arrA[]` from lb to ub in `arrA[]`

e.g., for final merge:





C CODE

```c
// merge two sorted portions of an integer array arrA[]:
// portions are lb to mid and mid+1 to ub
void merge (int arrA[], int lb, int mid, int ub) {

      int i, j, k;
      int size = ub - lb + 1;
      int *arrC;
      //create arrC[] to be of size needed for current merge
      arrC = (int*) malloc(size * sizeof(int));

      i = lb;
      j = mid + 1;
      k = 0;

      while (i <= mid && j <= ub) {
        if(arrA[i] <= arrA[j]) {
            arrC[k] = arrA[i];
            i++;
        }
        else {
            arrC[k] = arrA[j];
            j++;
        }
        k++;
    } //end while

      // write out anything left in i part
      while (i <= mid) {
          arrC[k] = arrA[i];
          i++;
          k++;
      }
      // write out anything left in j part
      while (j <= ub) {
          arrC[k] = arrA[j];
          j++;
          k++;
      }

      //write back from arrC to arrA so correct values are in place for next merge
      i = lb;
      k = 0;
      while ( i <= ub ) {
          arrA[i] = arrC[k];
          i++;
          k++;
      }
}
```

merge()

Time step analysis

Line	Cost	numTimes	cost* numTimes	
110-118	7	1	7	
120	1	$\frac{n}{2}+1$	$\frac{n}{2}+1$	
121	1	$\frac{n}{2}$	$\frac{n}{2}$	
122-123 or 126-127 & 129	3	$\frac{n}{2}$	$3\frac{n}{2}$	
133 or 139	1	$\frac{n}{2}$	$\frac{n}{2}$	
134-136 or 140-142	3	$\frac{n}{2}$	$3\frac{n}{2}$	
146-147	2	1	2	
148	1	$n+1$	$n+1$	
149-151	3	n	$3n$	$4n+\frac{9n}{2}+10$
				$17n+20$

```
105
106 // merge two sorted portions of an integer array arrA[]:
107 // portions are lb to mid and mid+1 to ub
108 void merge (int arrA[], int lb, int mid, int ub) {
109
110         int i, j, k;
111         int size = ub - lb + 1;
112         int *arrC;
113         //create arrC[] to be of size needed for current merge
114         arrC = (int*) malloc(size * sizeof(int));
115
116         i = lb;
117         j = mid + 1;
118         k = 0;
119
120         while (i <= mid && j <= ub) {
121             if(arrA[i] <= arrA[j]) {
122                 arrC[k] = arrA[i];
123                 i++;
124             }
125             else {
126                 arrC[k] = arrA[j];
127                 j++;
128             }
129             k++;
130         } //end while
131
132         // write out anything left in i part
133         while (i <= mid) {
134             arrC[k] = arrA[i];
135             i++;
136             k++;
137         }
138         // write out anything left in j part
139         while (j <= ub) {
140             arrC[k] = arrA[j];
141             j++;
142             k++;
143         }
144
145         //write back from arrC to arrA so correct values are in place for next merge
146         i = lb;
147         k = 0;
148         while ( i <= ub ) {
149             arrA[i] = arrC[k];
150             i++;
151             k++;
152         }
153 }
```

COMPLEXITY ANALYSIS

But how many times is `merge()` and `mergeSort()` carried out once the initial call to `mergeSort()` occurs?

HOW MANY TIMES?

```
35 //mergeSort to sort values in an integer array arrA[]
36 // lb = 0 and ub = size - 1 for the first call
37 void mergeSort (int arrA[], int lb, int ub) {
38
39     int mid;
40
41     if (lb < ub) {
42         mid = int((lb + ub) /2);
43         mergeSort (arrA, lb, mid);
44         mergeSort (arrA, mid + 1, ub);
45         merge (arrA, lb, mid, ub);
46     }
47 }
48
```

Line	Cost	numTimes	cost*numTimes	Total
L39	1			
L41	1			
L42	1			
L43	f($\frac{n}{2}$)	?		
L44	f($\frac{n}{2}$)	?		
L45	17n + 20	?		

Consider first call to `mergeSort()` when n = size

```
35 //mergeSort to sort values in an integer array arrA[]
36 // lb = 0 and ub = size - 1 for the first call
37 void mergeSort (int arrA[], int lb, int ub) {
38
39     int mid;
40
41     if (lb < ub) {
42         mid = int((lb + ub) /2);
43         mergeSort (arrA, lb, mid);
44         mergeSort (arrA, mid + 1, ub);
45         merge (arrA, lb, mid, ub);
46     }
47 }
48
```

Line	Cost
L39	1
L41	1
L42	1
L43	$f(\frac{n}{2})$
L44	$f(\frac{n}{2})$
L45	$17n + 20$
	$f(n) = 2\,f(\frac{n}{2}) + 17n + 23$

What happens for next 2nd call?

Substitute for *f(n)*

```
35 //mergeSort to sort values in an integer array arrA[]
36 // lb = 0 and ub = size - 1 for the first call
37 void mergeSort (int arrA[], int lb, int ub) {
38
39     int mid;
40
41     if (lb < ub) {
42         mid = int((lb + ub) /2);
43         mergeSort (arrA, lb, mid);
44         mergeSort (arrA, mid + 1, ub);
45         merge (arrA, lb, mid, ub);
46     }
47 }
48
```

Call:	Cost	
1	$f(n) = 2\ f(\frac{n}{2}) + 17n + 23$	
	We will ignore constants 17 and 23 calling them c and const	
	$f(n) = 2\ f(\frac{n}{2}) + c\ n + const$	
2	$f(n) = 2\ (f(\frac{n}{4}) + f(\frac{n}{4}) + \frac{cn}{2} + const) + cn + const$	
	$f(n) = 2\ (2f(\frac{n}{4}) + \frac{cn}{2} + const) + cn + const$	
	$f(n) = 4f(\frac{n}{4}) + \frac{2cn}{2} + const) + cn + const$	
	$f(n) = 4f(\frac{n}{4}) + 2cn + const$	

What happens for 3rd call?

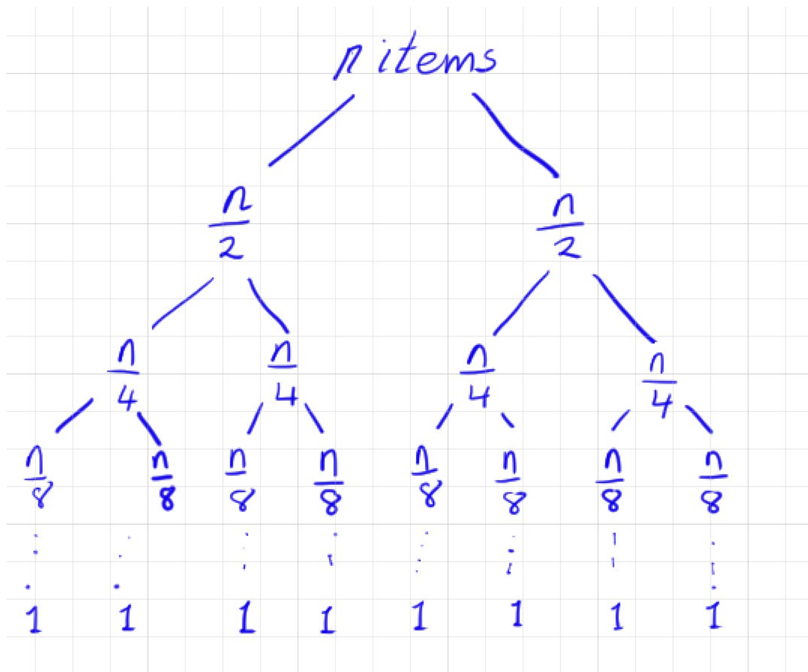
Again, substitute for *f(n)*

$f(n) = 2\,f(\frac{n}{2}) + c\,n + const$

```
35  //mergeSort to sort values in an integer array arrA[]
36  // lb = 0 and ub = size - 1 for the first call
37  void mergeSort (int arrA[], int lb, int ub) {
38
39      int mid;
40
41      if (lb < ub) {
42          mid = int((lb + ub) /2);
43          mergeSort (arrA, lb, mid);
44          mergeSort (arrA, mid + 1, ub);
45          merge (arrA, lb, mid, ub);
46      }
47  }
48
```

Call:	Cost
2	$f(n) = 4f(\frac{n}{4}) + 2cn + const$
3	$f(n) = 4\,(f(\frac{n}{8}) + f(\frac{n}{8}) + \frac{cn}{4} + const) + 2cn + const$
	$f(n) = 4\,(2f(\frac{n}{8}) + \frac{cn}{4} + const) + 2cn + const$
	$f(n) = 8f(\frac{n}{8}) + cn + const) + 2cn + const$
	$f(n) = 8f(\frac{n}{8}) + 3cn + const$
In general:	$f(n) = 2^k\, f(\frac{n}{2^k}) + k\,cn + const$
i.e., for 3rd call:	$f(n) = 2^3\, f(\frac{n}{2^3}) + 3cn + const$

Solve for k



Eventually, $\frac{n}{2^k}$ will be equal to 1 (at the final recursive call), i.e., $\frac{n}{2^k} = 1$

So therefore multiplying across: n $= 2^k$

Solve for k with $n = 2^k$

```
35 //mergeSort to sort values in an integer array arrA[]
36 // lb = 0 and ub = size - 1 for the first call
37 void mergeSort (int arrA[], int lb, int ub) {
38
39     int mid;
40
41     if (lb < ub) {
42         mid = int((lb + ub) /2);
43         mergeSort (arrA, lb, mid);
44         mergeSort (arrA, mid + 1, ub);
45         merge (arrA, lb, mid, ub);
46     }
47 }
48
```

	$f(n) = 2^k f(\frac{n}{2^k}) + c\,k\,n + const$
	$f(n) = n\,f(\frac{n}{n}) + c\,k\,n + const$
	$f(n) = n\,f(1) + c\,k\,n + const$
Let f(1) = const time	$f(n) = n + c\,k\,n + const$

Now we must solve for k:

If $n = 2^k$, multiply both sides by $\log_2$ to get rid of power of 2:

$\log_2 n = k$

Can now substitute for k in $f(n) = n + c\,k\,n + const$

Giving: $n + c\,\log_2 n\ \ n + const$

$O(n \log_2 n)$

COMPLEXITY ANALYSIS SUMMARY

- Average and worst case performance is O$(n\ log_2 n)$
- Generally has fewer comparisons than quicksort (which we will see next)
- Is a general purpose sorting technique – works on any data type where a comparison is possible
- However, it does not sort **in place**. That is, it requires an array of the same size to hold the values temporarily and also requires a "write-back" stage. Therefore in practice, because it has poor space complexity it is not used for sorting data in arrays despite its good time complexity

Question from exam paper on mergeSort()

Sec. A Q. 3.

The following two functions, `mergeSort()` and `merge()`, sort integer values in the array `arrA[]` with associated size (`size`). (Line numbers are included).

```
L1   // must be called initially with lb = 0 and ub = size - 1
L2   void mergeSort (int arrA[], int lb, int ub)
L3   {
L4          int mid;
L5          if (lb < ub) {
L6              mid = int((lb + ub) /2);
L7              mergeSort (arrA, lb, mid);
L8              mergeSort (arrA, mid + 1, ub);
L9              merge (arrA, lb, mid, ub);
L10     }
L11  }
L12
L13  void merge (int arrA[], int lb, int mid, int ub)
L14  {
L15      int i, j, k;
L16      int *arrC;
L17      int size = ub - lb + 1;
L18      arrC = (int*) malloc(size * sizeof(int));
L19
L20      for (i = lb, j = mid + 1, k = 0; i <= mid && j <= ub; k++) {
L21         if (arrA[i] <= arrA[j])
L22            arrC[k] = arrA[i++];
L23         else
L24            arrC[k] = arrA[j++];
L25      }
L26      while (i <= mid)
L27         arrC[k++] = arrA[i++];
L28      while (j <= ub)
L29         arrC[k++] = arrA[j++];
L30      for (i = lb, k = 0; i <= ub; i++, k++)
L31         arrA[i] = arrC[k];
L32  }
```

(a) Using some sample data, and with reference to the code line numbers, explain, in your own words, how the `mergeSort()` function works. *(4 marks)*

(b) Using some sample data, and with reference to the code line numbers, explain, in your own words, how the `merge()` function works. *(4 marks)*

(a) Using some sample data, and with reference to the code line numbers, explain, in your own words, how the mergeSort() function works. *(4 marks)*

L1	// must be called initially with lb = 0 and ub = size - 1
L2	void mergeSort (int arrA[], int lb, int ub)
L3	{
L4	int mid;
L5	if (lb < ub) {
L6	mid = int((lb + ub) /2);
L7	mergeSort (arrA, lb, mid);
L8	mergeSort (arrA, mid + 1, ub);
L9	merge (arrA, lb, mid, ub);
L10	}
L11	}

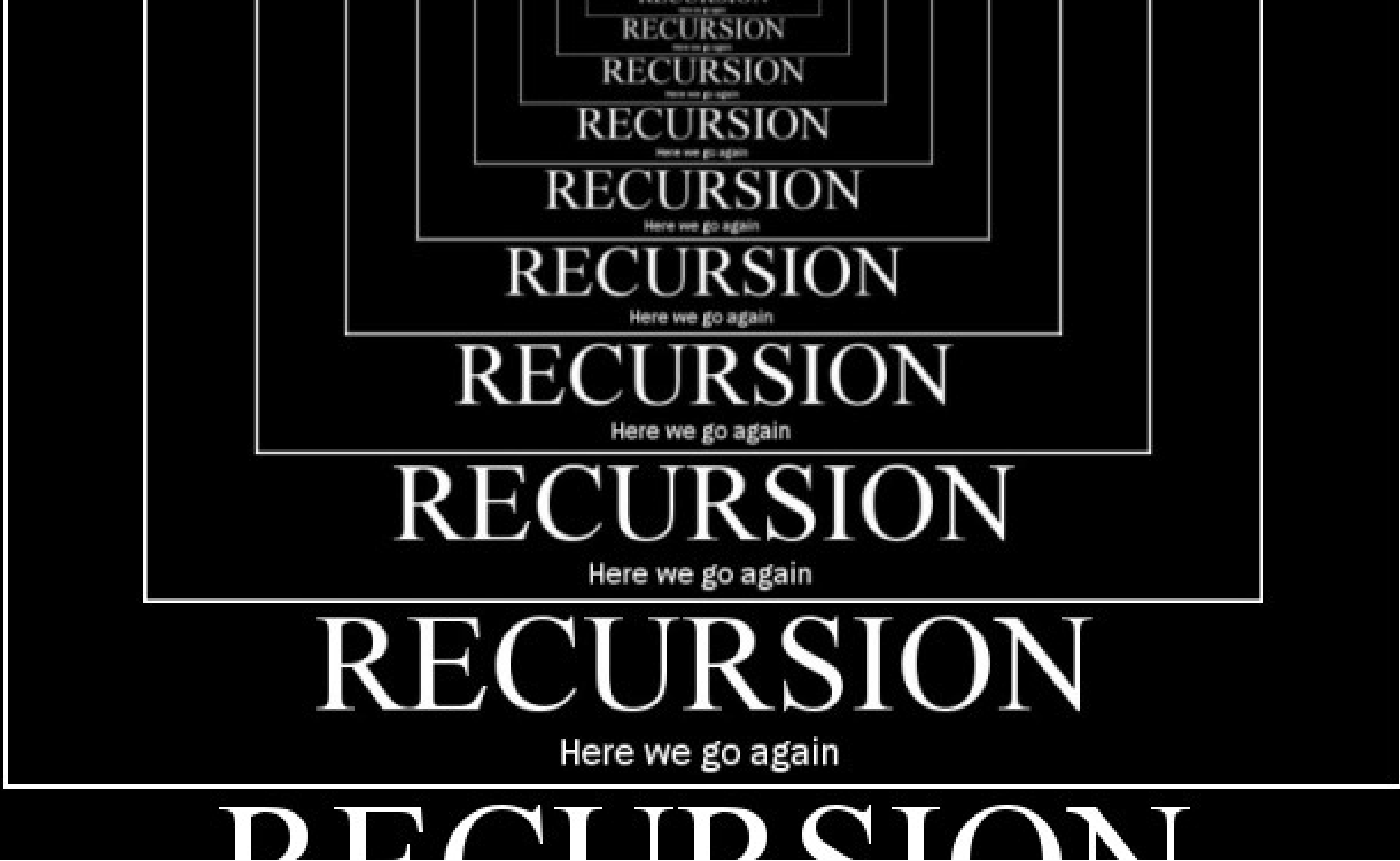
(b) Using some sample data, and with reference to the code line numbers, explain, in your own words, how the merge() function works. *(4 marks)*

```
L13 void merge (int arrA[], int lb, int mid, int ub)
L14 {
L15     int i, j, k;
L16     int *arrC;
L17     int size = ub - lb + 1;
L18     arrC = (int*) malloc(size * sizeof(int));
L19
L20     for (i = lb, j = mid + 1, k = 0; i <= mid && j <= ub; k++) {
L21         if (arrA[i] <= arrA[j])
L22             arrC[k] = arrA[i++];
L23         else
L24             arrC[k] = arrA[j++];
L25     }
L26     while (i <= mid)
L27         arrC[k++] = arrA[i++];
L28     while (j <= ub)
L29         arrC[k++] = arrA[j++];
L30     for (i = lb, k = 0; i <= ub; i++, k++)
L31         arrA[i] = arrC[k];
L32 }
```

ANOTHER QUESTION TO CONSIDER?

How can we re-write mergeSort() iteratively?

Will consider this later after looking at quickSort



Iterative versions of QUICKSORT AND MERGE SORT

CT102 Algorithms

ITERATIVE QUICKSORT

How to modify the quicksort function to remove the recursive calls?

```
void quickSort(int arrA[], int startval, int endval) {
    if((endval - startval) < 1) {
        return;
    }
    else {
        int k = partition(arrA, startval, endval);
        quickSort(arrA, startval, k - 1);   //left partition
        quickSort(arrA, k + 1, endval);    //right partition
    }
}
```

WHAT NEEDS TO CHANGE?

Partition part will not need to change (as is already iterative).

We need a way to keep track of the correct sub-portions of the array that are to partitioned, i.e. `startval` and `endval` and these will need to be updated as we continue with left and right sub-portions.

```
void quickSort(int arrA[], int startval, int endval) {
    if((endval - startval) < 1){
        return;
    }
    else {
        int k = partition(arrA, startval, endval);
        quickSort(arrA, startval, k - 1);  //left partition
        quickSort(arrA, k + 1, endval);    //right partition
    }
}
```

ONE APPROACH: Use an additional array to keep track of `startval` and `endval`

This array will always hold pairs of values (`startval, endval`)

We will continuously add and delete from this array to get the current values of startval and endval to send to the partition function.

Call this array `next[]` and we will only access it from the "top" where the most recent pair of values have been added.

Use variable `top` to access pairs of values.

How to make sure you are accessing correct pairs?

To get current pair of values:

```
endval = next[top--]; (or endval = next[top]; top--;)
startval = next[top--];
```

To add next pair of values (if something left to add) and have k

```
// left of pivot:
next[++top] = startval;
next[++top] = k - 1;
// right of pivot:
next[++top] = k + 1;
next[++top] = endval;
```

N.B.
use -- and ++
correctly

N.B. they are added in the correct order

To add next pair of values (if something left to add):

Given we have value k returned from partition, then

```
if (k - 1 > startval) {
// there are values to left of pivot
if (k + 1 < endval) {
// there are values to right of pivot
```

COMBINING ...

```
if (k - 1 > startval) {
    // left of pivot:
    next[++top] = startval;
    next[++top] = k - 1;
}
if (k + 1 < endval) {
    // right of pivot:
    next[++top] = k + 1;
    next[++top] = endval;
}
```

WHEN TO STOP?

Initially on entry, top = -1;

We add the first `startval` and `endval` to `next[]` so then, `top` should be 1.

When all `startval` and `endval` pairs are removed from `next[]` all the work will be finished and `top` should have value -1 again.

Therefore, keep going:

```
while (top >= 0) {
```

FULL FUNCTION

```
void quickSortIt(int arrA[], int startval, int endval) {

    int size = endval - startval + 1;
    int *next;
    next = (int*) malloc(size * sizeof(int));

    // initialize top to access next[]
    int top = -1;

    // add initial values of startval and endaval to next[]
    next[++top] = startval;
    next[++top] = endval;

    // Start partitioning while some values left in next[]
    while (top >= 0) {
        endval = next[top--];
        startval = next[top--];

        // partition and get k
        int k = partition1(arrA, startval, endval);

        // Values on lhs of pivot? if so, add to next[]
        if (k - 1 > startval) {
            next[++top] = startval;
            next[++top] = k - 1;
        }

        // Values on rhs of pivot? if so, add to next[]
        if (k + 1 < endval) {
            next[++top] = k + 1;
            next[++top] = endval;
        }
    }
}
```

EXAMPLE:
(using 1st position for pivot)

arrA[8]

10	12	25	3	7	11	6	17
0	1	2	3	4	5	6	7

next[8]

0	7						
0	1	2	3	4	5	6	7

arrA[8]

6	3	7	10	25	11	12	17
0	1	2	3	4	5	6	7

next[8]

0	2	4	7				
0	1	2	3	4	5	6	7

arrA[8]

6	3	7	10	11	12	25	25
0	1	2	3	4	5	6	7

After 2nd partition

k = 7

next[8]

0	2	4	6				
0	1	2	3	4	5	6	7

```
154
155 void quickSortIt(int arrA[], int startval, int endval) {
156
157     int size = endval - startval + 1;
158     int *next;
159     next = (int*) malloc(size * sizeof(int));
160
161     // initialize top to access next[]
162     int top = -1;
163
164     // add initial values of startval and endaval to next[]
165     next[++top] = startval;
166     next[++top] = endval;
167
168     // Start partitioning while some values left in next[]
169     while (top >= 0) {
170         endval = next[top--];
171         startval = next[top--];
172
173         // partition and get k
174         int k = partition1(arrA, startval, endval);
175
176         // Values on lhs of pivot? if so, add to next[]
177         if (k - 1 > startval) {
178             next[++top] = startval;
179             next[++top] = k - 1;
180         }
181
182         // Values on rhs of pivot? if so, add to next[]
183         if (k + 1 < endval) {
184             next[++top] = k + 1;
185             next[++top] = endval;
186         }
187     }
188 }
189
```

EXAMPLE:
(using 1st position for pivot)

After 3rd partition k = 6

arrA[8]

6	3	7	10	11	12	17	25
0	1	2	3	4	5	6	7

next[8]

0	2	4	5				
0	1	2	3	4	5	6	7

arrA[8]

6	3	7	10	11	12	17	25
0	1	2	3	4	5	6	7

next[8]

0	2						
0	1	2	3	4	5	6	7

arrA[8]

3	6	7	10	11	12	17	25
0	1	2	3	4	5	6	7

```
154
155 void quickSortIt(int arrA[], int startval, int endval) {
156
157     int size = endval - startval + 1;
158     int *next;
159     next = (int*) malloc(size * sizeof(int));
160
161     // initialize top to access next[]
162     int top = -1;
163
164     // add initial values of startval and endaval to next[]
165     next[++top] = startval;
166     next[++top] = endval;
167
168     // Start partitioning while some values left in next[]
169     while (top >= 0) {
170         endval = next[top--];
171         startval = next[top--];
172
173         // partition and get k
174         int k = partition1(arrA, startval, endval);
175
176         // Values on lhs of pivot? if so, add to next[]
177         if (k - 1 > startval) {
178             next[++top] = startval;
179             next[++top] = k - 1;
180         }
181
182         // Values on rhs of pivot? if so, add to next[]
183         if (k + 1 < endval) {
184             next[++top] = k + 1;
185             next[++top] = endval;
186         }
187     }
188 }
189
```

ANALYSIS

Still have the same behaviour in terms of splitting and placing values, so still $O(n\ log_2 n)$

However, the trade-off is having an extra array to keep track of the `startval` and `endval` pairs but this would be more efficient than the recursive stack generally (for programming languages not particularly suited to recursion).

QUESTION

Can you now modify the iterative version to include a call to insertion sort for small sub-arrays? At what line(s) do we need to add this?

```
154
155 void quickSortIt(int arrA[], int startval, int endval) {
156
157     int size = endval - startval + 1;
158     int *next;
159     next = (int*) malloc(size * sizeof(int));
160
161     // initialize top to access next[]
162     int top = -1;
163
164     // add initial values of startval and endaval to next[]
165     next[++top] = startval;
166     next[++top] = endval;
167
168     // Start partitioning while some values left in next[]
169     while (top >= 0) {
170         endval = next[top--];
171         startval = next[top--];
172
173         // partition and get k
174         int k = partition1(arrA, startval, endval);
175
176         // Values on lhs of pivot? if so, add to next[]
177         if (k - 1 > startval) {
178             next[++top] = startval;
179             next[++top] = k - 1;
180         }
181
182         // Values on rhs of pivot? if so, add to next[]
183         if (k + 1 < endval) {
184             next[++top] = k + 1;
185             next[++top] = endval;
186         }
187     }
188 }
189
```

ITERATIVE MERGE SORT

Recall two main steps in merge sort:

- Part 1: "dividing": continuously reduce array and sub-arrays until you have sub-arrays of size 1 (trivially in sorted order)
- Part 2: "conquering": continuously merge back sorted sub-arrays in sorted order

```
void mergeSort(int [], int, int);
```

```
35 //mergeSort to sort values in an integer array arrA[]
36 // lb = 0 and ub = size - 1 for the first call
37 void mergeSort (int arrA[], int lb, int ub) {
38
39     int mid;
40
41     if (lb < ub) {
42         mid = int((lb + ub) /2);
43         mergeSort (arrA, lb, mid);
44         mergeSort (arrA, mid + 1, ub);
45         merge (arrA, lb, mid, ub);
46     }
47 }
48
```

Recall: Line 43 and 44 “just” consider smaller and smaller sub-portions of the array until Line 41 is false, then the merging (and comparisons) start (Line 45)

For an iterative version, we want a way to reduce to these smaller sub-portions using a loop instead of Line 43 and 44.

ONE APPROACH:

Given an array arrA[] with size values:

- first merge all sub-arrays of size 1 to create sorted subarrays of size 2
- then merge all sorted sub-arrays of size 2 to create sorted sub-arrays of size 4
- then merge all sorted sub-arrays of size 4 to create sorted sub-arrays of size 8

… etc

- finally merge the two sorted sub-arrays, each of size/2 to create the sorted array

```
int currSize;
int lb, mid, ub;
```

Use a variable `currSize` which should begin at size 1 (considering only one value) and increment by a factor of 2 for each iteration

```
for (currSize=1; currSize <= size - 1; currSize = 2 * currSize) {
```

At each stage, will need to know the current `lb` for the sub-portions of the array being considered based on `currSize`

```
for (lb = 0; lb < size - 1; lb = lb + 2 * currSize) {
```

Once we have lb we can calculate mid and ub based on this lb and currSize

As long as the values don't go past the end of the array then:

```
mid =  lb + currSize - 1 ;
ub =   lb + 2 * currSize - 1;
```

PUTTING ALL THIS TOGETHER:

```
289
290  // Iterative version of mergeSort()
291  void mergeSortIt(int arrA[], int size) {
292
293      int currSize;
294      int lb, mid, ub;
295
296      for (currSize = 1; currSize <= size - 1; currSize = 2 * currSize) {
297
298          for (lb = 0; lb < size - 1; lb = lb + 2*currSize) {
299
300              if (lb + currSize - 1 < size - 1) {
301                  mid = lb + currSize - 1;
302              }
303              else {
304                  mid = size - 1;
305              }
306
307              if (lb + 2 * currSize - 1 < size - 1) {
308                  ub = lb + 2 * currSize - 1;
309              }
310              else {
311                  ub = size - 1;
312              }
313
314              merge(arrA, lb, mid, ub);
315          }
316      }
317  }
```

EXAMPLE

```
for (currSize=1; currSize <= size - 1; currSize = 2*currSize) {
  for (lb = 0; lb < size - 1; lb = lb + 2 * currSize) {
     if (lb + currSize - 1 < size - 1)  mid = lb + currSize - 1;
     else mid = size - 1;
     if (lb + 2 * currSize - 1 < size - 1) ub = lb + 2 * currSize - 1;
     else ub = size - 1;
```

Consider at start, currSize =1

arrA[8]

7	17	25	3	7	10	6	17
0	1	2	3	4	5	6	7

currSize	lb	mid	ub	arrA
1	0	0	1	{7, 17, 25, 3, 7, 10, 6, 17}
1	2	2	3	{7, 17, 3, 25, 7, 10, 6, 17}
1	4	4	5	{7, 17, 3, 25, 7, 10, 6, 17}
1	6	6	7	{7, 17, 3, 25, 7, 10, 6, 17}

EXAMPLE

```
for (currSize=1; currSize <= size - 1; currSize = 2*currSize) {
  for (lb = 0; lb < size - 1; lb = lb + 2 * currSize) {
    if (lb + currSize - 1 < size - 1)  mid = lb + currSize - 1;
    else mid = size - 1;
    if (lb + 2 * currSize - 1 < size - 1) ub = lb + 2 * currSize - 1;
    else ub = size - 1;
```

Now update currSize: currSize = 2*currSize = 2*1 = 2

arrA[8]	7	17	3	25	7	10	6	17
	0	1	2	3	4	5	6	7

currSize	lb	mid	ub	arrA
2	0	1	3	{3, 7, 17, 25, 7, 10, 6, 17}
2	4	5	7	{3, 7, 17, 25, 6, 7, 10, 17}

EXAMPLE

```
for (currSize=1; currSize <= size - 1; currSize = 2*currSize) {
  for (lb = 0; lb < size - 1; lb = lb + 2 * currSize) {
     if (lb + currSize - 1 < size - 1)  mid = lb + currSize - 1;
     else mid = size - 1;
     if (lb + 2 * currSize - 1 < size - 1) ub = lb + 2 * currSize - 1;
     else ub = size - 1;
```

Now update currSize: currSize = 2*currSize = 2*2 = 4

arrA[8]	3	7	17	25	6	7	10	17
	0	1	2	3	4	5	6	7

currSize	lb	mid	ub	arrA
4	0	3	7	{3, 6, 7, 7, 10, 17, 17, 25}

Now update currSize: currSize = 2*currSize = 2*4 = 8

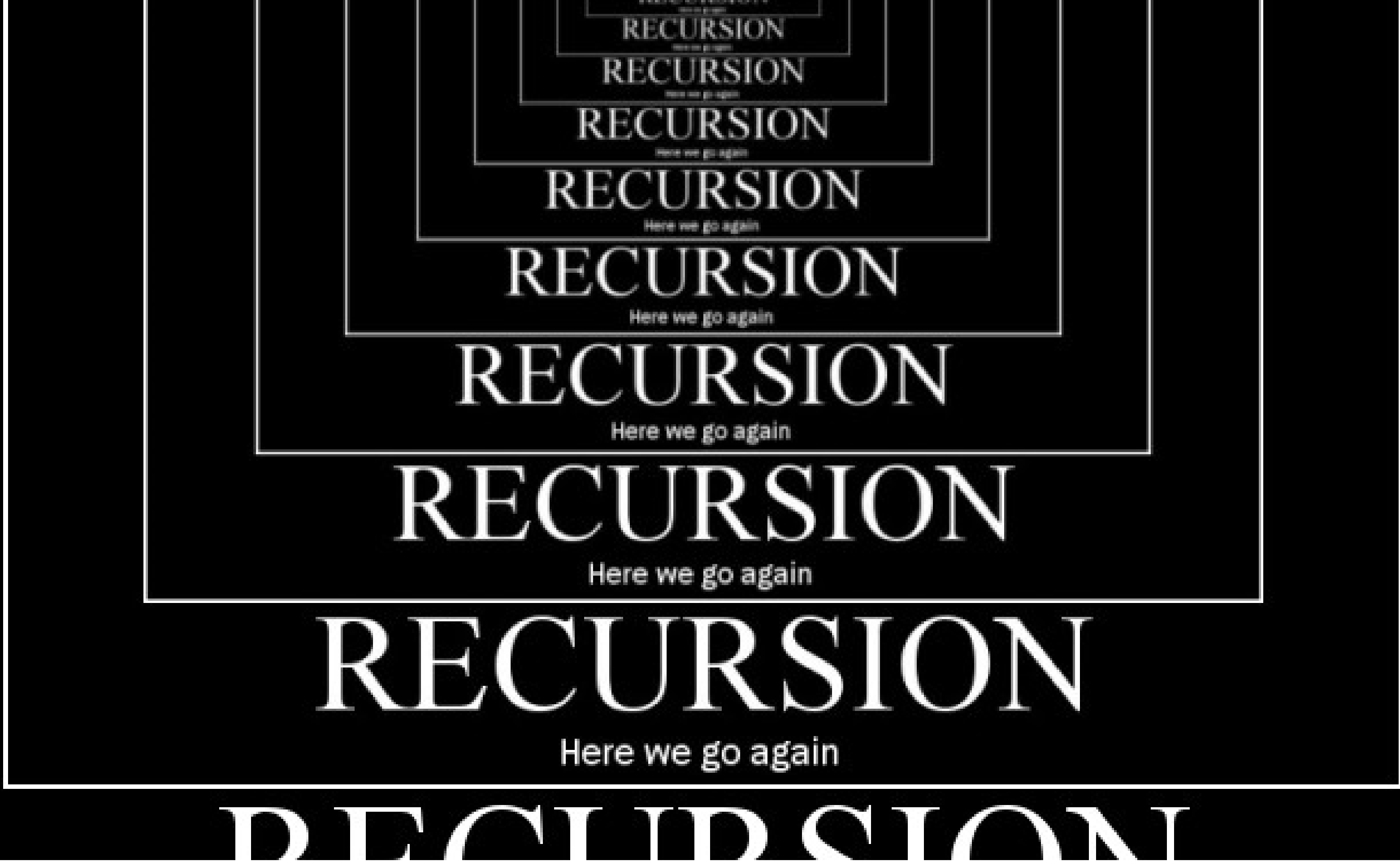
SUMMARISING:

arrA[8]	7	17	25	3	7	10	6	17
	0	1	2	3	4	5	6	7

currSize	lb	mid	ub	arrA
1	0	0	1	{7, 17, 25, 3, 7, 10, 6, 17}
1	2	2	3	{7, 17, 3, 25, 7, 10, 6, 17}
1	4	4	5	{7, 17, 3, 25, 7, 10, 6, 17}
1	6	6	7	{7, 17, 3, 25, 7, 10, 6, 17}
2	0	1	3	{3, 7, 17, 25, 7, 10, 6, 17}
2	4	5	7	{3, 7, 17, 25, 6, 7, 10, 17}
4	0	3	7	{3, 6, 7, 7, 10, 17, 17, 25}

SUMMARY

- Quick sort and Merge Sort give the best performance on average ($O(n\ log_2 n)$) when sorting general purpose data
- Although the recursive implementation is easy to understand (and contains less code) both algorithms are often implemented iteratively.
- Note that the merge() and partition() functions are unchanged in the iterative versions.



QUICKSORT

CT102 Algorithms

menti.com
Code: 7869 6363

Questions/Issues?

Completed studentsurvey.ie?

Topics for revision?

Another "divide and conquer" Recursive Sorting algorithm: quick sort

Widely used in practice in any application needing to sort data – operating systems, database systems, built-in libraries, and methods.

On average the algorithm provides the **fastest** comparison sort for *general data.*

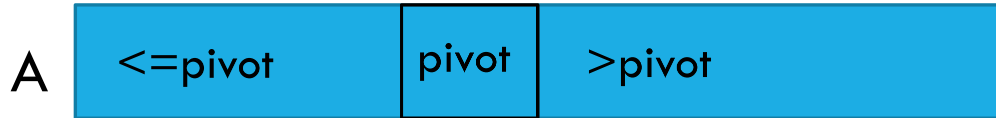
We will consider a partially recursive solution, although it can be re-written to be fully iterative.

SORTING IN C
(Outside of CT102 and CT103!)

- The built in sorting function in C is a variant of quicksort and sorts data in an array
- In stdlib.h

void qsort(void *base, size_t nitems, size_t size, int (*compare)(const void *, const void*))

- Sorts the **nitems** of the array pointed by **base.** Every element has a size of **size-t** bytes long. The qsort function will sort according to the comparator function specified in **compare**
- The qsort function does not return a value. The array data is modified in the specified order



APPROACH:

Sorting values in an integer array arrA[]

Quicksort works by splitting – or *partitioning* values in array A - by choosing a pivot such that:

- All items to left of pivot are <= pivot
- All items to right of pivot are > pivot
- We *hope* that about half the items will be less than the pivot value and half the items will be greater than it

Then the two sub parts (left and right of pivot) are sorted separately using the same approach

Base case is when there is only 1 element left and the data will then be sorted

WHY DOES THIS WORK?

Consider splitting/partitioning an array into two halves using previous idea and using selection sort to sort each of the two halves.

The total time required to sort the two sub-arrays is only half the time that would have been required to sort the original array … i.e. half the number of comparisons are required.

EXAMPLE

Given the following integer array of size 8.

Pick the value at index location 0 to be pivot value (i.e., 10)

arrA[8]	10	12	25	3	7	11	6	17
	0	1	2	3	4	5	6	7

EXAMPLE: (using 1st position for pivot)

arrA[8]

10	12	25	3	7	11	6	17
0	1	2	3	4	5	6	7

			10				
0	1	2	3	4	5	6	7

6	3	7	10	12	25	11	17
0	1	2	3	4	5	6	7

Now have one value placed in correct position and all other values in correct portion of array

Now consider sub-array portions:

0 to 2:

6	3	7
0	1	2

10
3

4 to 7:

12	25	11	17
4	5	6	7

3	6	7
0	1	2

10
3

11	12	25	17
4	5	6	7

Now consider sub-array portion 6 to 7:

3	6	7	10	11	12
0	1	2	3	4	5

17	25
6	7

WORK DONE

arrA

<=pivot	pivot	>pivot

For each pivot chosen and partitioning of array:

- One value (pivot) can be placed in its correct position and will not have to be moved again (we do not know this location at the start though)
- The amount of comparisons/work done for subsequent calls is reduced – ideally halved … as the values have been moved to <= pivot or to > pivot and will not have to ever be compared to each other

PARTITIONING

The main work is to:

- Get pivot value
- Partition the array in to the 2 subparts:
  - values <=pivot on LHS of pivot
  - values > pivot on RHS of pivot
- Place pivot value in correct position (we will call it position k)

Repeat this for smaller and smaller sub-arrays until there is nothing left to partition

INPUT AND OUTPUT

Inputs: Array `arrA[]` of integers in unsorted order with:

- lower bound `startval`
- upper bound `endval`

Outputs: Array `arrA[]` of integers in ascending sorted order.

Partial function:

```
void quickSort(int arrA[], int startval, int endval)
{
    int k;
    if ((endval - startval) < 1) {
        return;
    }
    else {
        //partition
        quickSort(arrA, startval, k - 1);  //left partition
        quickSort(arrA, k + 1, endval);   //right partition
    }
}
```

Base case

One value left in the array:

```
if ((endval - startval) < 1){
        return;
}
```

Reduce

If not at base case then should call function with smaller arrays (left and right partitions)

If pivot is at location k then calls should be:

```
//now sort the two sub-arrays
quickSort(arrA, startval, k - 1);  //left partition
quickSort(arrA, k + 1, endval);   //right partition
```

Choosing pivot value?

Any value can be chosen between `startval` and `endval` as the pivot.

For convenience, we will initially take the value at `startval` to be the pivot*

When partitioning is complete, pivot must be moved to the correct (and final) position in the array, which is k.

* we will revisit this decision later.

EXAMPLE:

Consider the following data in an integer array arrA[]

```
int arrA[7] = {10, 17, 2, 7, 13, 6, 11};
```

Pick pivot at index = 0, so pivot is 10

arrA[7]	**10**	**17**	**2**	**7**	**13**	**6**	**11**
	0	1	2	3	4	5	6

How to move 10 to its correct position and move all other values to left or right of this?

PARTITION … one idea …
assuming pivot is stored at `arrA[0]`

- Use two extra arrays:

  `lessThEq[]` to hold values <= pivot

  `grThan[]` to hold values > pivot

- Loop through `arrA[]` from `startval` to `endval` checking each value against the pivot value and moving to correct array
- When finished write back the values to `arrA[]` in the following order:
  - all values in `lessThEq[]`, starting at `startval` in `arrA[]`
  - the pivot value (and let k = this location)
  - all values in `grThan[]`, ending at `endval` in `arrA[]`

EXAMPLE:

Consider the following data in an integer array `arrA[]` of size 7

arrA[7]

10	17	2	7	13	6	11
0	1	2	3	4	5	6

Pick pivot at index = 0, so pivot is 10

lessThEq[7]

2	7	6				
0	1	2	3	4	5	6

grThan[7]

17	13	11				
0	1	2	3	4	5	6

arrA[7]

2	7	6	10	17	13	11
0	1	2	3	4	5	6

PARTITION … better idea … "in place partition" - using the same array rather than creating temporary arrays

One approach:

- If pivot not at `startval`, move it to `startval`
- From LHS (`startval + 1`), start comparing values to the pivot value … keep going as long as values are `<= pivot`.
- From RHS of array (`endval`), start comparing elements to pivot, keep going as long as values are `> pivot`.
- If there are still values to check and LHS value and RHS value are out of order, swap these and keep going with comparison.
- If all values have been compared, swap pivot value with the value on RHS that is `<= pivot`

EXAMPLE:

Consider the following data in an integer array `arrA[]` of size 7

pivot = 10;

arrA[7] — Initial array

10	17	2	7	13	6	11
0	1	2	3	4	5	6
	i				k	

arrA[7] — After first swap

10	6	2	7	13	17	11
0	1	2	3	4	5	6
			k	i		

arrA[7] — Final array (after 1 partition)

7	6	2	10	13	17	11
0	1	2	3	4	5	6
			k	i		

STEPS:

1. Put pivot value in first location (*)
2. Set up left and right traversals of array
   - Index i to traverse array from left; set to *startval*+1initially
   - Index k to traverse array from right; set to *endval* initially
3. With variable i, start comparing values to pivot value at each stage, moving on to next location (`i++`), if value is `<= pivot and i <= k`
4. With variable k, start comparing values to pivot value at each stage, moving on to next location (`k--`), if current value is `> pivot and k >= i`

arrA[7]	10	17	2	7	13	6	11
	0	1	2	3	4	5	6
		i					k

C CODE:

```
i = startval + 1;
k = endval;
while(k >= i) {
    while (arrA[i] <= pivot && i <= k) {
        i++;
    }
    while (arrA[k] > pivot && k >= i) {
        k--;
    }
    //swap needed?
}
```

YOU TRY ...

```
L1 i = startval + 1;
L2 k = endval;
L3 while(k >= i) {
L4    while (arrA[i] <= pivot && i <= k) {
L5        i++;
L6    }
L7    while (arrA[k] > pivot && k >= i) {
L8        k--;
L9    }
L10    //k still > i?
L11 }
```

arrA[10]

10	7	2	17	13	6	21	9	14	20
0	1	2	3	4	5	6	7	8	9
startval									

Go to www.menti.com and use the code 7869 6363

Pick pivot at index = 0, so pivot is 10

`startval = 0` and `endval = 9`

What is the value of **i** when get to line 10 (L10)?

What is the value of **k** when get to line 10 (L10)?

YOU TRY ...

```
L1 i = startval + 1;
L2 k = endval;
L3 while(k >= i) {
L4     while (arrA[i] <= pivot && i <= k) {
L5         i++;
L6     }
L7     while (arrA[k] > pivot && k >= i) {
L8         k--;
L9     }
L10     //k still > i
L11 }
```

arrA[10]	10	7	2	17	13	6	21	9	14	20
	0	1	2	3	4	5	6	7	8	9

arrA[10]	10	7	2	9	13	6	21	17	14	20
	0	1	2	3	4	5	6	7	8	9

After this swap when will we next get to L10?

Now, what is the value of **i** when get to line 10 (L10)?

Now, what is the value of **k** when get to line 10 (L10)?

YOU TRY ...

```
L1 i = startval + 1;
L2 k = endval;
L3 while(k >= i) {
L4    while (arrA[i] <= pivot && i <= k) {
L5        i++;
L6    }
L7    while (arrA[k] > pivot && k >= i) {
L8        k--;
L9    }
L10    //k still > i?
L11 }
```

arrA[10]	10	7	2	9	13	6	21	17	14	20
	0	1	2	3	4	5	6	7	8	9
					i	k				

arrA[10]	10	7	2	9	6	13	21	17	14	20
	0	1	2	3	4	5	6	7	8	9
					i	k				

arrA[10]	10	7	2	9	6	13	21	17	14	20
	0	1	2	3	4	5	6	7	8	9
					k	i				

AT SOME STAGE ....
one of two possibilities:

```
L1 i = startval + 1;
L2 k = endval;
L3 while (k >= i) {
L4     while (arrA[i] <= pivot && i <= k) {
L5         i++;
L6     }
L7     while (arrA[k] > pivot && k >= i) {
L8         k--;
L9     }
L10     //k still > i?
L11 }
```

1. Not all of the array has been traversed:
 i at value which is > pivot (L4 false) and
 k at value which is <= pivot (L7 false) and
 k >= i (L3 true)

2. All of the array has been traversed (i.e. all values have been compared to the current pivot value): (L3 false)

1. Not all of the array has been traversed:

In this case swap the values at i and k, and continue with i and k loop:

```
if (k > i) {
  swap(&arrA[i], &arrA[k]);
}
```

arrA[10]	10	7	2	17	13	6	21	9	14	20
	0	1	2	3	4	5	6	7	8	9
				i				k		

arrA[10]	10	7	2	9	13	6	21	17	14	20
	0	1	2	3	4	5	6	7	8	9
				i				k		

2. i and k have passed each other

Finished work for this partition so put pivot in its correct location and return this location (k): swap pivot value with value at k

```
swap(&arrA[startval], &arrA[k]);
```

arrA[10]

10	7	2	9	6	13	21	17	14	20
0	1	2	3	4	5	6	7	8	9
				k	i				

arrA[10]

10	7	2	9	6	13	21	17	14	20
0	1	2	3	4	5	6	7	8	9
start val				k	i				

arrA[10]

6	7	2	9	10	13	21	17	14	20
0	1	2	3	4	5	6	7	8	9

Defining swap()

```
void swap (int *, int *);
// call with  ... swap(&arrA[i], &arrA[j]); to swap
// values in array arrA[] at locations i and j
void swap(int* a, int* b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

partition() code in full:

```
269
270 int partition (int arrA[], int startval, int endval)
271 {
272     int i = startval + 1;
273     int k = endval;
274     int pivot = arrA[startval];
275
276     while (k >= i) {
277         while (arrA[i] <= pivot && i <= k) {
278             i++;
279         }
280         while (arrA[k] > pivot && k >= i) {
281             k--;
282         }
283         if (k > i){ //swap values at k and i
284             swap(&arrA[i], &arrA[k]);
285         }
286     }
287     //out of this loop when k >= i not true
288     swap(&arrA[startval], &arrA[k]);
289     return(k);
290 }
```

Note:

The partition function (the main work) is iterative – using loops for control, not recursion

BETTER VERSION?

We would like to re-write the `partition()` function without the nested while loop.

Idea is:

- Keep pivot at `startval`.
- k is used to compare values and also increments to the `endval`.
- Location of i is at the “last small” value found.
- When some k location finds a value <= pivot, i is incremented and the value at i and k is swapped.
- When finished, swap values at locations i and `startval`

```
269
270 int partition (int arrA[], int startval, int endval)
271 {
272     int i = startval + 1;
273     int k = endval;
274     int pivot = arrA[startval];
275
276     while (k >= i) {
277         while (arrA[i] <= pivot && i <= k) {
278             i++;
279         }
280         while (arrA[k] > pivot && k >= i) {
281             k--;
282         }
283         if (k > i){ //swap values at k and i
284             swap(&arrA[i], &arrA[k]);
285         }
286     }
287     //out of this loop when k >= i not true
288     swap(&arrA[startval], &arrA[k]);
289     return(k);
290 }
```

ALTERNATIVE PARTITION

(still using same `swap()` function)

```
220
221 // better version of partition .. no nested loop
222 // pivot at startval as before
223 int partition2 (int arrA[], int startval, int endval)
224 {
225     int k;
226     int pivot = arrA[startval];
227     int i = startval;
228
229     for (k = startval + 1; k <= endval; k++) { // k keeps incrementing to the end
230         if (arrA[k] <= pivot) {
231             i++;   // i only increments when there is a new value to add to the <= portion
232             if (i != k) {
233                 swap (&arrA[i], &arrA[k]);
234             }
235         }
236     }
237     swap (&arrA[i], &arrA[startval]);  // put pivot in correct location i
238     return(i);
239 }
240
```

EXAMPLE:

pivot = 10;

```
223  int partition2 (int arrA[], int startval, int endval)
224  {
225      int k;
226      int pivot = arrA[startval];
227      int i = startval;
228
229      for (k = startval + 1; k <= endval; k++) { // k keeps incrementing to the end
230          if (arrA[k] <= pivot) {
231              i++;   // i only increments when there is a new value to add to the <= portion
232              if (i != k) {
233                  swap (&arrA[i], &arrA[k]);
234              }
235          }
236      }
237      swap (&arrA[i], &arrA[startval]);  // put pivot in correct location i
238      return(i);
239  }
240
```

Initial array

arrA[7]	10	17	2	7	13	6	11
	0	1	2	3	4	5	6
	i	k					

arrA[7]	10	17	2	7	13	6	11
	0	1	2	3	4	5	6
	i		k				

Swap at i and k

arrA[7]	10	17	2	7	13	6	11
	0	1	2	3	4	5	6
		i	j				

EXAMPLE:

pivot = 10;

```
223  int partition2 (int arrA[], int startval, int endval)
224  {
225      int k;
226      int pivot = arrA[startval];
227      int i = startval;
228
229      for (k = startval + 1; k <= endval; k++) { // k keeps incrementing to the end
230          if (arrA[k] <= pivot) {
231              i++;   // i only increments when there is a new value to add to the <= portion
232              if (i != k) {
233                  swap (&arrA[i], &arrA[k]);
234              }
235          }
236      }
237      swap (&arrA[i], &arrA[startval]);  // put pivot in correct location i
238      return(i);
239  }
240
```

arrA[7]

10	2	17	7	13	6	11
0	1	2	3	4	5	6
	i		k			

After swap (L233) and k++

arrA[7]

10	2	17	7	13	6	11
0	1	2	3	4	5	6
		i	k			

arrA[7]

10	2	7	17	13	6	11
0	1	2	3	4	5	6
		i		k		

After swap (L233) and k++

EXAMPLE:

pivot = 10;

```
223 int partition2 (int arrA[], int startval, int endval)
224 {
225     int k;
226     int pivot = arrA[startval];
227     int i = startval;
228 
229     for (k = startval + 1; k <= endval; k++) { // k keeps incrementing to the end
230         if (arrA[k] <= pivot) {
231             i++;   // i only increments when there is a new value to add to the <= portion
232             if (i != k) {
233                 swap (&arrA[i], &arrA[k]);
234             }
235         }
236     }
237     swap (&arrA[i], &arrA[startval]);  // put pivot in correct location i
238     return(i);
239 }
240 
```

arrA[7]

10	2	7	17	13	6	11
0	1	2	3	4	5	6
		i			k	

arrA[7]

10	2	7	6	13	17	11
0	1	2	3	4	5	6
			i			k

After swap (L233) and k++

arrA[7]

6	2	7	10	13	17	11
0	1	2	3	4	5	6
			i			k

Final swap (L237)

Analysis of partition2() for one call with array of size N and startval at 0

```
223  int partition2 (int arrA[], int startval, int endval)
224  {
225      int k;
226      int pivot = arrA[startval];
227      int i = startval;
228
229      for (k = startval + 1; k <= endval; k++) { // k ke
230          if (arrA[k] <= pivot) {
231              i++;   // i only increments when there is
232              if (i != k) {
233                  swap (&arrA[i], &arrA[k]);
234              }
235          }
236      }
237      swap (&arrA[i], &arrA[startval]);  // put pivot in
238      return(i);
239  }
```

Line	Cost	numTimes	cost* numTimes
225-227	3	1	7
229	1	n	n
230	1	n – 1	n – 1
231 and 232 Assume half of the values will be <= pivot *not guaranteed	2	$\frac{n-1}{2}$	n - 1
233	1? or more? assume 4	$\frac{n-1}{2}$	$4(\frac{n-1}{2})$ = 2n – 2
237	4	1	4
238	1	1	1
			f(n) = 5n + 8

How about full function? Again assume that array is partitioned evenly

For 1st call ….

```
139
140  void quickSort(int arrA[], int startval, int endval) {
141
142      if ( (endval - startval) < 1) {
143          return;
144      }
145      else {
146          int k = partition2(arrA, startval, endval);
147          //now sort the two sub-arrays
148          quickSort(arrA, startval, k - 1);  //left partition
149          quickSort(arrA, k + 1, endval);    //right partition
150      }
151  }
152
```

Line	Cost
L142	1
L143	1
L146	5n + 8
L148	$f(\frac{n}{2})$
L149	$f(\frac{n}{2})$
	$f(n) = 2\ f(\frac{n}{2}) + 5n + 8$

What happens for next 2nd call?

Substitute for *f(n)*

```
139
140 void quickSort(int arrA[], int startval, int endval) {
141
142     if ( (endval - startval) < 1) {
143         return;
144     }
145     else {
146         int k = partition2(arrA, startval, endval);
147         //now sort the two sub-arrays
148         quickSort(arrA, startval, k - 1);  //left partition
149         quickSort(arrA, k + 1, endval);   //right partition
150     }
151 }
152
```

Call:	Cost
1	$f(n) = 2\ f(\frac{n}{2}) + 5n + 8$
	We will ignore constants 5 and 8 calling them c and const
	$f(n) = 2\ f(\frac{n}{2}) + c\ n + const$
2	$f(n) = 2\ (f(\frac{n}{4}) + f(\frac{n}{4}) + \frac{cn}{2} + const) + cn + const$
	$f(n) = 2\ (2f(\frac{n}{4}) + \frac{cn}{2} + const) + cn + const$
	$f(n) = 4f(\frac{n}{4}) + \frac{2cn}{2} + const) + cn + c$
	$f(n) = 4f(\frac{n}{4}) + 2cn + const$

What happens for 3rd call

Again, substitute for *f(n)*

$$f(n) = 2\, f(\frac{n}{2}) + c\, n + const$$

```
139
140  void quickSort(int arrA[], int startval, int endval) {
141
142      if ( (endval - startval) < 1) {
143          return;
144      }
145      else {
146          int k = partition2(arrA, startval, endval);
147          //now sort the two sub-arrays
148          quickSort(arrA, startval, k - 1);  //left partition
149          quickSort(arrA, k + 1, endval);   //right partition
150      }
151  }
152
```

Call:	Cost
2	$f(n) = 4f(\frac{n}{4}) + 2cn + const$
3	$f(n) = 4\,(f(\frac{n}{8}) + f(\frac{n}{8}) + \frac{cn}{4} + const) + 2cn + const$
	$f(n) = 4\,(2f(\frac{n}{8}) + \frac{cn}{4} + const) + 2cn + const$
	$f(n) = 8f(\frac{n}{8}) + cn + const) + 2cn + const$
	$f(n) = 8f(\frac{n}{8}) + 3cn + const$
As we have seen already with merge sort …	
In general:	$f(n) = 2^k\, f(\frac{n}{2^k}) + k\, cn + const$
i.e., for 3rd call:	$f(n) = 2^3\, f(\frac{n}{2^3}) + 3cn + const$

SOLVING FOR k …

If the array is partitioned evenly, then on average, each partition produces 2 sub-array portions:

- place 1 item and produce 2 sub-array portions
- place 2 items and produce 4 sub-array portions
- place 4 items and produce 8 sub-array portions
- etc.
- until we have 1 item in each sub-array portion

Eventually, $\frac{n}{2^k}$ will be equal to 1 (at the final recursive call), i.e., $\frac{n}{2^k} = 1$

So therefore multiplying across: $n = 2^k$

As before, solve for k with $n = 2^k$

```
139
140 void quickSort(int arrA[], int startval, int endval) {
141
142     if ( (endval - startval) < 1) {
143         return;
144     }
145     else {
146         int k = partition2(arrA, startval, endval);
147         //now sort the two sub-arrays
148         quickSort(arrA, startval, k - 1);  //left partition
149         quickSort(arrA, k + 1, endval);    //right partition
150     }
151 }
152
```

	$f(n) = 2^k f(\frac{n}{2^k}) + c\,k\,n + const$
	$f(n) = n\,f(\frac{n}{n}) + c\,k\,n + const$
	$f(n) = n\,f(1) + c\,k\,n + const$
Let f(1) = const time	$f(n) = n + c\,k\,n + const$

Now we must solve for k:

If $n = 2^k$, multiply both sides by $\log_2$ to get rid of power of 2:

$\log_2 n = k$

Can now substitute for k in $f(n) = n + c\,k\,n + const$

Giving: $n + c \log_2 n \ * n + const$

$O(n \log_2 n)$

NOTE ON PERFORMANCE

However, this performance is dependent on the value of the pivot:

- A "good" pivot will split the array very evenly in two halves thus giving the O(n log n) complexity
- A "poor" pivot will not split the array at all and give $O(n^2)$ complexity

Generally we will not see improvements in performance on small arrays

Quicksort is often modified so that when the sub-array is small (e.g., size = 10) an algorithm that performs fewer swaps and comparisons is used – think about how you might do this!

EXAMPLE: (using 1st position for pivot)

arrA[8]

10	12	25	30	35	41	44	70
0	1	2	3	4	5	6	7

After 1st partition, everything goes to right of pivot at location 0

10	12	25	30	35	41	44	70
0	1	2	3	4	5	6	7

After 2nd partition, everything goes to right of pivot at location 1

10	12	25	30	35	41	44	70
0	1	2	3	4	5	6	7

After 3rd partition, everything goes to right of pivot at location 2

etc.

NOTE: would have the exact same problem if using the last position for pivot)

arrA[8]

10	12	25	30	35	41	44	70
0	1	2	3	4	5	6	7

After 1st partition, everything goes to left of pivot at location 7

10	12	25	30	35	41	44	70
0	1	2	3	4	5	6	7

After 2nd partition, everything goes to left of pivot at location 6

10	12	25	30	35	41	44	70
0	1	2	3	4	5	6	7

After 3rd partition, everything goes to left of pivot at location 5

etc.

NOTE: a less extreme version of this problem would occur if the first portion of the array has the smallest values

arrA[8]	0	1	2	35	30	41	40	35
	0	1	2	3	4	5	6	7

After 1st partition, everything goes to right of pivot at location 0

0	1	2	35	30	41	40	70
0	1	2	3	4	5	6	7

After 2nd partition, everything goes to right of pivot at location 1

After 3rd partition, then start to make more than one sub-array per partition call

PICKING BETTER PIVOTS

As well as the modification when the sub-arrays are small, a better choice of pivot value is also used in practice to get better performance.

For example:

- pivot location at middle of the array is chosen per run.
- pivot location is chosen randomly per run (good and quick random number generator needed).
- pivot location is chosen from getting the median of the values at the first, last and middle location of the array per run.

WOULD USING MID POSITION HELP?

```
mid = int(startval + endval)/2;
```

arrA[8]

10	12	25	30	35	41	44	70
0	1	2	3	4	5	6	7

mid = 3 and get two sub-arrays:

0 to 2 and 4 to 7

10	12	25	30	35	41	44	70
0	1	2		4	5	6	7

mid = 1 and get two sub-arrays:

0 to 0 and 2 to 2

mid = 5 and get two sub-arrays:

4 to 4 and 6 to 7

etc.

IMPLEMENTING BETTER PIVOTS?

We should still move the pivot to the `startval` location, but we do not need to pick the value at this location.

Try implementing the previous approaches to get pivot value (based on current `startval` and `endval`):

```
int mid = int(startval + endval) / 2;
swap (&arrA[mid], &arrA[startval]);
pivot = arrA[startval];
```

TESTING:

- Try test the quicksort code with the larger files.
- As well as checking number of comparisons and swaps we are also interested in counting the number of function calls required for quicksort.
- In this situation, you need to declare and initialise global counters that are incremented on each entry to the relevant functions

```
17
18  int cnt_qs_calls = 0;
19  int cnt_p_calls = 0;
```

RESULTS (comparing with merge sort):

```
Calling QuickSort with array size 1000
time taken is 0.001000
#quickSort calls = 1343, #partition calls = 671 and # swaps calls = 2382
```

```
Calling MergeSort with array size 1000
time taken is 0.002000
For mergeSort:  #mergeSort calls = 1999 and #merge calls = 999
```

RESULTS
(comparing with other techniques):

```
Calling QuickSort with array size 1000
time taken is 0.001000
#quickSort calls = 1343, #partition calls = 671 and # swaps calls = 2382
```

```
1000 items have been read
In Insertion Sort:
time taken is: 0.002000 seconds;
number of swaps is: 257335;
number of comparisons is: 258334_
```

```
1000 items have been read
In Selection Sort:
time taken is 0.003000;
number of swaps is 999;
number of comparisons is 499500_
```

```
1000 items have been read
In Bubble Sort:
time taken is 0.007000;
number of swaps is 257335;
number of comparisons is 499500
```

HYBRID QUICK SORT:

Quicksort is often modified so that when the sub-array is small (e.g., size = 10) an algorithm that performs fewer swaps and comparisons is used

What needs to change in function quicksort to allow this modification?

```
void quickSort(int arrA[], int startval, int endval) {
    if((endval - startval) < 1){
        return;
    }
    else {
        int k = partition(arrA, startval, endval);
        quickSort(arrA, startval, k - 1);  //left partition
        quickSort(arrA, k + 1, endval);   //right partition
    }
}
```

HYBRID QUICK SORT:

```
void quickSortHybrid(int arrA[], int startval, int endval)
{
    if((endval - startval) < 1) {
            return;
    }
    else if (endval - startval + 1 < 10) { //subarrays of size < 10
        // call Insertion Sort or Selection Sort
    }
    else {
        int k = partition1(arrA, startval, endval);
        quickSort(arrA, startval, k - 1);  //left partition
        quickSort(arrA, k + 1, endval);   //right partition
    }
}
```

MODIFICATIONS TO INSERTION SORT?

what lines need to change and how?

```
void insertionSort(int[], int);
```

```
28
29  // Insertion Sort: integer array arrA [] of size
30  void insertionSort(int arrA[], int size)
31  {
32      int i, j, curr;
33
34      for(i = 1; i < size; i++) {
35          curr  = arrA[i];
36
37          for(j = i - 1; j >= 0 && curr < arrA[j]; j--) {
38              //make room ...
39              arrA[j + 1] = arrA[j];
40          }
41
42          if (i != j + 1) // if not at the correct position
43              arrA[j + 1] = curr;
44
45      } // end outer i for
46
47  } //return
48
```

MODIFICATIONS TO INSERTION SORT?

what lines need to change and how?

```
void insertionSort(int[], int);
```

```
//Insertion Sort: a sub-portion of the array
void insSort(int arrA[], int startval, int endval) {
      int i, j, curr;
      for (i = startval + 1; i <= endval; i++) {
            curr  = arrA[i];
            for (j = i - 1; j >= startval && curr < arrA[j]; j--) {
                  arrA[j+1] = arrA[j];
            }
            if (i != j + 1) {
                  arrA[j + 1] = curr;
            }
      } // end i for
}
```

```
29  // Insertion Sort: integer array arrA [] of size
30  void insertionSort(int arrA[], int size)
31  {
32      int i, j, curr;
33
34      for(i = 1; i < size; i++) {
35          curr  = arrA[i];
36
37          for(j = i - 1; j >= 0 && curr < arrA[j]; j--) {
38              //make room ...
39              arrA[j + 1] = arrA[j];
40          }
41
42          if (i != j + 1) // if not at the correct position
43              arrA[j + 1] = curr;
44
45      } // end outer i for
46
47  } //return
```

ITERATIVE QUICK SORT

How to modify the quicksort function to remove the recursive calls? (Next lecture)

```
void quickSort(int arrA[], int startval, int endval) {
    if((endval - startval) < 1) {
        return;
    }
    else {
        int k = partition(arrA, startval, endval);
        quickSort(arrA, startval, k - 1);  //left partition
        quickSort(arrA, k + 1, endval);    //right partition
    }
}
```

SUMMARY

- Quicksort along with Merge Sort give the best performance on average ($O(n\ log_2 n)$) when sorting general purpose data.
- Quicksort is the default sorting program used in all applications – however a fully iterative, and often hybrid, version tends to be used (see next).
- Important to understand the partition aspect as this is where the main work ("conquering") is done, as well as the difference between the recursive and iterative approach to dividing the array.