OCR A Level Computer Science

Your notes

8.1 Algorithms

Contents

Your notes

Suitability of Algorithms

Why do we Need to Check the Suitability of Algorithms?

- **Algorithms** must be compared to check the suitability for a specific set of data
- Algorithms can be **compared** to determine **how long they take** to solve a particular problem and **how much memory** they require to solve problems
- Generally, algorithms that **finish quicker** and **use less memory** are preferred
- The measurement of time for an algorithm to complete is called **"time complexity"** whereas the measurement of memory is called **"space complexity"**

Time Complexity

- To measure time complexity, the execution time in terms of **operations or steps performed** is compared, **not the numerical time** taken in seconds and minutes
- It is important to understand that an algorithm's **time complexity is independent of the hardware and CPU** used to run it. This means a faster CPU with higher clock speed will still take the same number of steps or instructions to execute the algorithm compared to a slower CPU with lower clock speed
    - An analogy would be running a marathon. The marathon distance and course is the algorithm, whereas the runners are different CPUs with different speeds. The runners will complete the marathon with different times but the marathon is still the same distance
    - Alternatively, students completing several exam papers. Each exam has a different number of questions (like the number of instructions in an algorithm). Each student acts like a CPU. Some students will finish the exams faster than others. The number of questions are the same however for each student
- The following algorithm calculates the sum of the first n integers:

```
function sumNumbers1(n)
    sum = 0
    for i = 1 to n
        sum = sum + n
    next i
    return sum
endfunction
```

- Compare this to the algorithm below which also calculates the sum of the first n integers

```
function sumNumbers2(n)
    sum = n * (n+1)/2
```

```
    return sum
endfunction
```


Your notes

- Both algorithms above complete the same task, however the latter algorithm, sumNumbers2, is far more efficient in terms of the number of instructions executed
- In sumNumbers1, the number of instructions executed are as follows:
  - +1 for (sum = 0)
  - +n for (for i = 1 to n)
- The total number of instructions is therefore n+1. If n, the quantity of integers, is 10, then the total number of instructions executed is 11. If n is 1,000, then the total number is 1001
- As n increases, the (sum = 0) line becomes more and more **insignificant** in contributing to the overall time complexity. **As n increases, the algorithm becomes more inefficient**
- As n increases the time complexity of the algorithm becomes dependent on n
  - For further context, compare this to numerical time. If each instruction took one second to execute and if n is 1,000, then the algorithm would take 1,001 seconds to complete, where as if n is 10, it would take 11 seconds to complete
- In sumNumbers2, the number of instructions executed are as follows:
  - +1 for (sum = n * (n+1)/2)
- The total number of instructions is therefore 1. If n, the quantity of integers, is 10, then the total number of instructions executed is 1. If n is 1,000, then the total number is still 1
- No matter how big n becomes, sumNumbers2 always takes the same amount of time. Its time complexity is constant
- It is important to recognise that both algorithms share the same goal but complete them in very different ways

Space Complexity

- Space complexity is defined as how much **memory** an algorithm requires to complete
- If additional space is required to duplicate the data for example, this will contribute to the overall space complexity

Big O Notation

Your notes

Big O Notation

What is Big O Notation?

- Big O Notation is used to express the **complexity** of an algorithm
- Describes the **scalability** of an algorithm as it's **order** (**O**), referred to in Computer Science as **Big O Notation**
- Named after the German mathematician **Edmund Landau**
- Has two simple rules
  - **Remove** all terms except the one with the **largest factor** or **exponent**
  - **Remove** any **constants**
- There are several **classifications** used in **Big O Notation**, ranging from f**ast and efficient** to **slow and inefficient**. These are all covered below

Constant

- O(1), read as **Big-O One**, describes an algorithm that takes **constant time**, that is, requires the **same number of instructions to be executed, regardless of the size of the input data**
- For example, to calculate the length of a string 's' the following code could be used:
  - totalLength = len(s)
- No matter how long the string is, it will always take the same number of instructions to find the length

Linear

- O(n), read as **Big-O n**, describes an algorithm that takes **linear time**, that is, as the input size increases, the **number of instructions to be executed grows proportionally**
- For example, a linear search of 1000 items will take 10 times longer to search than an array of 100 items
- Implementing linear time requires use of **iteration** i.e. for loops and while loops
- In the example of sumNumbers1, a basic (for i = 1 to n) loop is used
- Below is an example of an algorithm that uses multiple for loops

```
function loop
```

```
x = input
for i = 1 to x
    print(x)
next i

a = 0
for j = 1 to x
    a = a + 1
next j
print(a)

b = 1
for k = 1 to x
    b = b + 1
next k
print(b)

endfunction
```

- When the user inputs a value of x, each loop will run x times, performing varied instructions in each loop
- The total number of instructions executed in this algorithm are as follows:
  - +1 for (x = input)
  - +x for the first for loop
  - +1 for (a = 0)
  - +x for the second for loop
  - +1 for (output a)
  - +1 for (b =1)
  - +x for the third for loop
  - +1 for (output b)
- The total number of instructions executed would therefore be: $3x + 5$
- As x increases the total number of instructions executed also increases as shown in the table below

Linear Time: Number of Instructions Executed

x	3x	5	3x + 5
1	3	5	8

10	30	5	35
100	300	5	305
10,000	30,000	5	30,005
1,000,000	3,000,000	5	3,000,005


Your notes

- The constant term (5) contributes less and less to the overall time complexity as x increases. 3x is the **significant contributing term** and causes the overall time complexity to grow linearly in a straight line as x increases

Polynomial

- $O(n^2)$, read as **Big-O n-squared**, describes an algorithm that takes **squared time to complete**. This means the performance is **proportional to the square of the number of data inputs**
- For example, a bubble sort is a polynomial time complexity algorithm. n passes are made, and for each pass n comparisons are made hence n*n or $O(n^2)$
- Implementing polynomial time requires the use of **nested iteration** i.e. for loops or while loops, where each loop executes n times
- An example of a polynomial algorithm is shown below:

```
function timesTable

x = 12
for i = 1 to x
    for j = 1 to x
        print(i, " x ", j, " = ", i*j)
    next j
next i

for k = 1 to x
    print(k)
next k
print("Hello World!")

endfunction
```

- The above algorithm outputs the times tables from the 1 times table to the 12 times table. An arbitrary for loop (for k = 1 to 10) and output (print("Hello World!")) has been added for demonstration purposes as shown below
- The total number of instructions executed by this algorithm are as follows:


Your notes

- +1 for (x = 12)
- +x * x for the nested for loop
- +x for the non-nested for loop
- +1 for (print("Hello World!"))

- The total number of instructions executed would therefore be: x*x + x + 2 or, $x^2 + x + 2$
- As x increases the total number of instructions executed also increases as shown in the table below

Polynomial Time: Number of Instructions Executed

x	x^2	2	$x^2 + x + 2$
1	1	2	4
10	100	2	112
100	10,000	2	10,102
10,000	100,000,000	2	100,010,002
1,000,000	1,000,000,000,000	2	1,000,001,000,002

- The constant term (2) and linear term (x) **contribute very little as x increases, making x^2 the significant contributing term**
  - To put this algorithm into perspective, assuming it took one second per instruction, printing up to the 1,000,000 times table would take roughly 32,000 years
- Loops can be nested any number of times. In the example above, there are two loops nested. Each loop nested increases the polynomial time as shown below. **Regardless of the number of loops nested, all are still referred to as polynomial time!**

Big-O of Nested Loops

Number of loops nested	Big-O time
2	$O(n^2)$
3	$O(n^3)$

4	$O(n^4)$
5	$O(n^5)$
6	$O(n^6)$


Your notes

- Below is an example of a six nested algorithm. Note that the algorithm does nothing useful except demonstrate this nesting

```
for a = 1 to n
    for b = 1 to n
        for c = 1 to n
            for d = 1 to n
                for e = 1 to n
                    for f = 1 to n
                    next f
                next e
            next d
        next c
    next b
next a
```

Exponential

- Exponential time, read as **Big-O 2-to-the-n**, describes an algorithm that takes **double the time to execute for each additional data input value**. The number of instructions executed becomes very large, very quickly for each input value
- Few everyday algorithms use exponential time due to its sheer inefficiency. Some problems however can only be solved in exponential time

Exponential time: Number of Instructions Executed

x	2^x
1	2
10	1024
100	1,267,650,600,228,229,401,496,703,205,376

Examiner Tips and Tricks

You do not need to know any specific examples of exponential time complexity algorithms, but you do need to know how these algorithms compare to other time complexities.

Your notes

Logarithmic

- Logarithmic time, read as **Big-O log-n**, describes an algorithm that takes **very little time as the size of the input grows larger**
- Logarithmic time in computer science is usually measured in **base 2**. A **logarithm** is the value the base must be raised to to equal another number, for example:
  - $2^3 = 8$, $\log_2(8) = 3$, i.e. to get to the value 8, 2 must be raised to the power of 3
- A binary search is an example of logarithmic time complexity. Doubling the size of the input data has very little effect on the overall number of instructions executed by the algorithm, as shown below:

Logarithmic Time: Number of Instructions Executed

x	$\log_2(x)$
1 (2^0)	0
8 (2^3)	3
1,024 (2^{10})	10
1,048,576 (2^{20})	20
1,073,741,824 (2^{30})	30

- As shown above, increasing the number of input values to over one billion did very little to the number of instructions taken to complete the algorithm

Comparison of Complexity

- All algorithms have a **best case**, **average case** and **worst case complexities**.
- These are defined as the best possible outcome, the average outcome and the worst possible outcome in terms of the number of instructions required to complete the algorithm
- **Best case complexity** is where the algorithm performs **most efficiently** such as a linear search or binary search finding the correct item on the first element O(1), or a bubble sort being supplied with an already

sorted list O(n). A bubble sort must always make at least one pass, hence O(n)

- **Average case complexity** is where the algorithm performs **neither at its best or worst** on any given data. A linear search may find the item in the middle of the list and therefore have a Big-O notation of $O(n/2)$, or a bubble sort sorting only half a list $O(n^2 / 2)$
- **Worst case complexity** is where an algorithm is at its **least efficient** such as a linear search finding the item in the last position, or not at all O(n), or a bubble sort sorting a reversed list $O(n2)$, i.e. a list whose items are ordered highest to lowest
- **Big-O notation is almost always measured by the worst case performance of an algorithm**. This allows programmers to accurately select appropriate algorithms for problems and have a guaranteed minimum performance level
- The graphical representation of constant, linear, polynomial, exponential and logarithmic times is shown below

Your notes

Big-O Time Complexity Graph

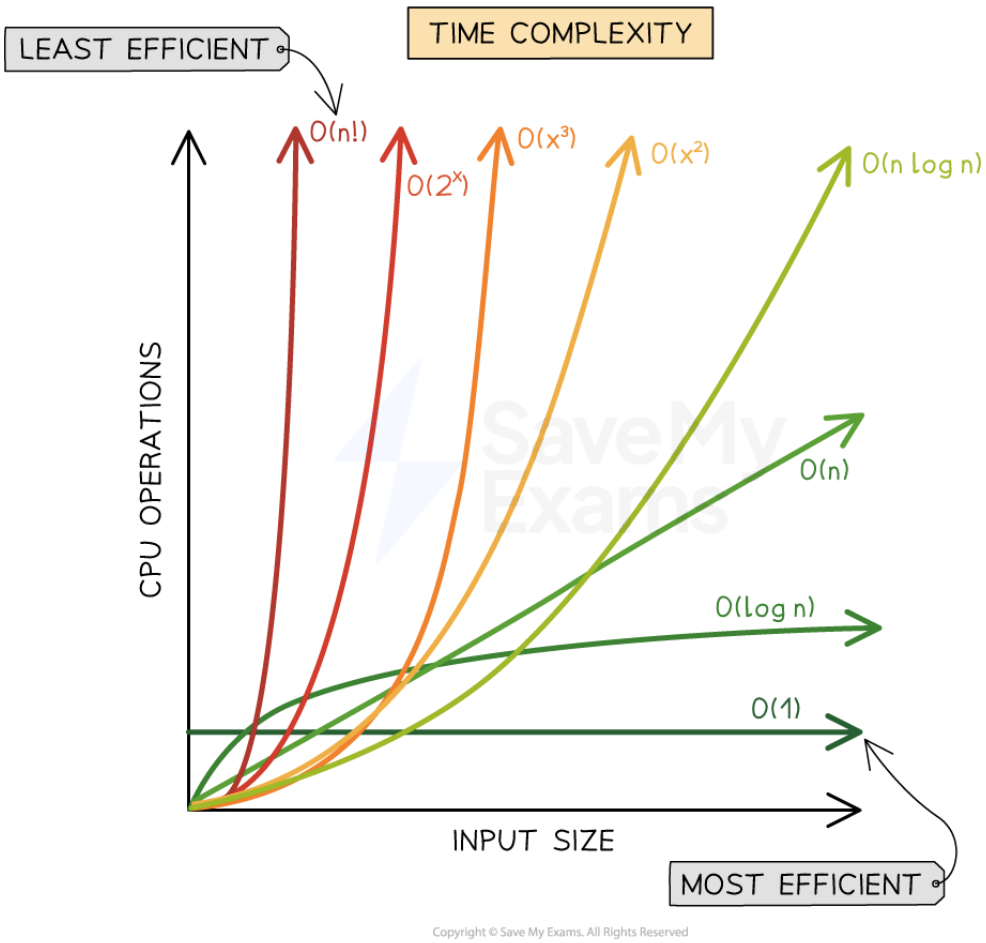


Figure 1: Graph of Time Complexity for Big-O Notation Functions

Your notes

- From the graph, **exponential time (2^n)** grows the **quickest** and therefore is the most **inefficient** form of algorithm, followed by **polynomial time (n^2)**, **linear time (n)**, **logarithmic time (logn)** and finally **constant time (1)**
- Further time complexities exist such as **factorial time (n!)** which performs **worse than exponential time** and **linear-logarithmic time (nlogn)** which sits **between polynomial and linear time**. Merge sort is an example of an algorithm that uses O(nlogn) time complexity
- The Big-O time complexity of an algorithm can be derived by inspecting its component parts
- It is important to know that when determining an algorithm's Big-O that **only the dominant factor is used**.
- This means the time complexity that **contributes the most** to the algorithm is used to describe the algorithm. For example:
  - $3x^2 + 4x + 5$, which describes three nested loops, four standard loops and five statements is described as $O(n^2)$
  - x^2 is the dominating factor. As the input size grows x^2 contributes proportionally more than the other factors
  - Coefficients are also removed from the Big-O notation. The 3 in $3x^2$ acts as a multiplicative value whose contribution becomes smaller and smaller as the size of the input increases.
    - For an arbitrarily large value such as 10^{100}, which is 10 with one hundred 0's after it, multiplying this number by 3 will have little contribution to the overall time complexity. 3 is therefore dropped from the overall time complexity

Examiner Tips and Tricks

- The most likely type of Big-O calculation question you will face in an exam is deriving a constant, linear or polynomial time complexity
- A very challenging question may ask to derive a logarithmic time complexity which will require familiarity with logarithmic algorithms from your course to recognise them

- The simplest way of deriving time complexity is to **count the number of loops and nested loops** in an algorithm.

```
function sumNumbers1(n)
    sum = 0
    for i = 1 to n
        sum = sum + n
    next i
```

```
    return sum
endfunction
```

- In the example above for sumNumbers1, a single for loop is present. As loops are O(n) and single statements are O(1), the overall Big-O notation is O(n), linear time, as the loop **dominates** constant time

```
function sumNumbers2(n)
    sum = n * (n+1)/2
    return sum
endfunction
```

- In the example above for sumNumbers2, a single statement is present with no loops. Statements are O(1) therefore making the overall Big-O notation O(1), constant time

```
function loop

x = input
for i = 1 to x
    print(x)
next i

a = 0
for j = 1 to x
    a = a + 1
next j
print(a)

b = 1
for k = 1 to x
    b = b + 1
next k
print(b)

endfunction
```

- In the example above for function loop, there are three loops present 3*O(n) and five statements 5*O(1). As the input size grows, the quantity of loops or statements becomes negligible, meaning coefficients eventually become factored out or removed. This leaves O(n) and O(1). As O(n) **dominates** O(1), the overall Big-O notation of this algorithm becomes O(n), linear time

```
function timesTable

x = 12
for i = 1 to x
    for j = 1 to x
        print(i, " x ", j, " = ", i*j)
    next j
next i

for k = 1 to x
    print(k)
```


Your notes

```
next k
print("Hello World!")

endfunction
```

- In the example above for function timesTable, there is a single nesting of two loops O(n^2), a single loop O(n) and a single statement O(1). As polynomial time **dominates** linear and constant time and that no coefficients are involved, the overall Big-O notation is O(n^2)

Worked Example

Dexter is leading a programming team who are creating a computer program that will simulate an accident and emergency room to train hospital staff.

Two of Dexter's programmers have developed different solutions to one part of the problem. Table 3.1 shows the Big O time complexity for each solution, where *n* = the number of data items.

	Solution A	Solution B
Time	O(n)	O(n)
Space	O(k^n) (where k > 1)	O(log n)

Table 3.1

1. Explain, with reference to the Big O complexities of each solution, which solution you would suggest Dexter chooses.
[4]

I would recommend solution B **[1]**. As both have the same time complexity we would need to look at both algorithms' space complexity **[1]**. A's space does not scale well **[1]** when the number of items is increased whereas B's space does scale well **[1]**.

Remember to relate your answer to the context of the question to achieve marks.

Your notes

Your notes

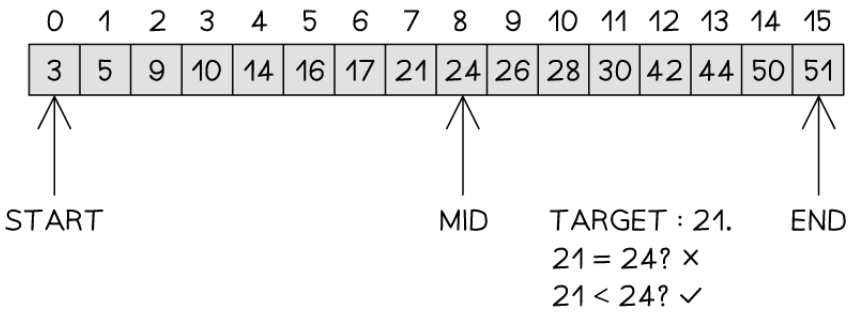
Binary Search

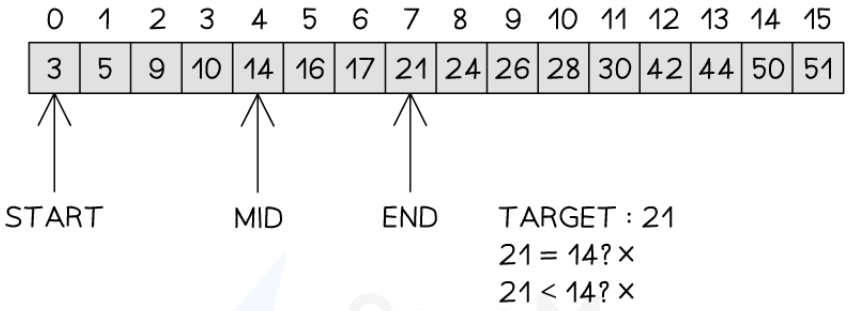
What is a Binary Search?

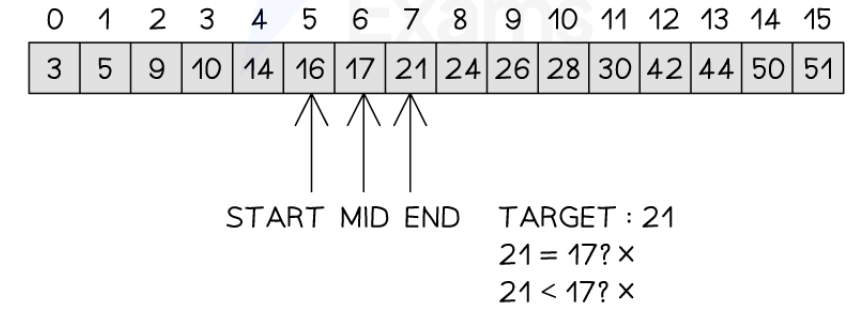
- The binary search is a more efficient search method than linear search
- The binary search **compares the middle item to the searched for target item**. If the **values match** then the **index is returned**. If not then the list is adjusted such that:
  - the **top half is ignored** if the **target item is less** than the middle value
  - or the **bottom half is ignored** if the **target item is larger** than the middle value
- Once the list has been adjusted, the search continues until the item is found or the item is not in the list
- It is important to note that the **list must be sorted** for the binary search to work correctly
- The implementation of the binary search is shown below

Performing the Binary Search

Your notes







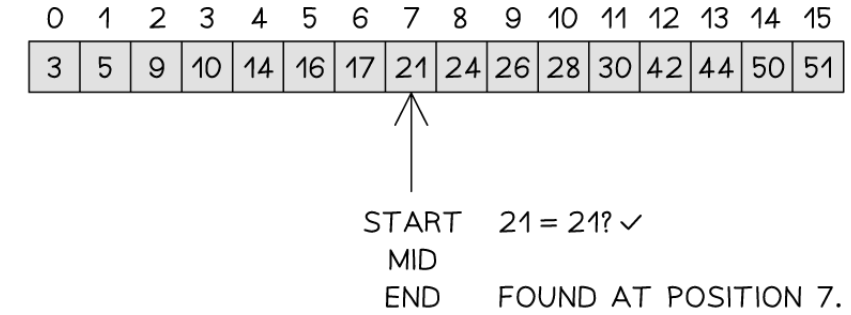


Figure 2: Performing the Binary Search

Your notes

Examiner Tips and Tricks

- When determining the rounding of the midpoint, rounding up or down are both valid provided consistent use of the same rounding is used throughout the algorithm
- It is important to note that the binary search does not discard, delete or remove parts of the list, it only adjusts the start, end and mid pointers. This only gives the appearance that items have been discarded or deleted

Time Complexity of a Binary Search

- The binary search is an example of a logarithmic time complexity O(logn) as it halves the search space with each iteration of the while loop. This approach is known as **divide and conquer**, where a problem is progressively reduced in size such that when each problem is at its smallest it is easiest to solve
- The algorithm starts with n items before the first iteration. After the first there are n/2 items, after the second there are n/4 items, after the third there are n/8 items and so on, leading to $n/2^i$ after i iterations
- The **worst case scenario** or maximum number of iterations to find one item is where $n/2^i = 1$
  - Multiply both sides by 2^i to get: $n = 2^i$
  - Take the logarithm of both sides: $\log n = \log 2^i$
  - Rewrite with i at the front (using laws of logarithms): $\log n = i \log 2$
  - Log2 (assuming a base of 2 is used) becomes 1, giving: $\log n = i$
  - The binary search is therefore **O(logn)**
- The **best case scenario would be O(1)** where the item is found on the first comparison, while the **average case** would be O(logn/2) which would find the item somewhere in the middle of the search. Removing coefficients means the average case would therefore still be **O(logn)**

Space Complexity of a Binary Search

- As the binary search requires no additional space, it is space complexity O(n), where n is the number of items in the list

Worked Example

Your notes

The list of positive even numbers up to and including 1000 is

2, 4, 6,... 500, 502,... 998, 1000

An attempt is to be made to find the number 607 in this list.

Use the values given to show the first three steps for:

i. a binary search

[3]

ii. a serial search

[3]

i) Compare 607 with 500 (mid) **[1]** then discard the lower half. Compare 607 with 750 (mid) **[1]** then discard the upper half. Compare 607 with 625 **[1]** and discard the upper half

ii) Compare 607 with 2 **[1]** then move to the next number. Compare 607 with 4 **[1]** then move to the next number. Compare 607 with 6 **[1]** then move to the next number

It is acceptable and useful to use a diagram in your answer showing how the binary search and linear search functions. Use the illustrations above as an example of how to show this

Tracing a Binary Search

- Given the following list [3, 5, 9, 10, 14, 16, 17, 21, 24, 26, 28, 30, 42, 44, 50, 51], a trace table for the binary search is shown below

Trace Table for the Binary Search

item	index	found	start	end	mid	list[mid]
21	-1	False	0	15	8	24
			0	7	4	14
			5	7	6	17
21	7	True	7	7	7	21

Implementing a Binary Search Pseudocode


Your notes

```
function binarySearch(list, item)
 found = False
 index = -1
 start = 0
 end = len(list - 1)
 while start <= last and found = False
  mid = round(start + end) / 2
  if list[mid] = item then
   found = True
   index = mid
  else
   if list[mid] < item then
    start = mid + 1
   else
    last = mid - 1
   endif
  endif
 endwhile
 return index
endfunction
```

- In the above algorithm, a list of n ordered items is passed to the function and three pointer values are set to the start element list[0], the mid element list[n/2], and the last element list[n-1]
- Each pointer is an integer value, with the **midpoint being rounded up to the next whole number**. These pointers are used to index the list
- With each iteration of the while loop, the mid value is compared against the item being looked for. If it is the same then the item is found and the index returned
- If it is not the same then the algorithm checks to see if it is smaller or bigger than the item being looked for. If it is **bigger then the start pointer is adjusted and moved to just after the mid pointer, otherwise the end pointer is adjusted and moved to just under the mid pointer**
- This gives the illusion of the list shrinking in size and allows the algorithm to narrow in on the true location of the item with each iteration. If the item is not found, -1 is returned

Python

```
def binary_search(list, item):
 found = False
 index = -1
```

```
start = 0
end = len(list) - 1

while start <= end and not found:
  mid = (start + end) // 2
if list[mid] == item:
  found = True
  index = mid
elif list[mid] < item:
  start = mid + 1
else:
  end = mid - 1

return index
```

Java

```
public class BinarySearch {

  public static int binarySearch(int[] list, int item) {
    boolean found = false;
    int index = -1;
    int start = 0;
    int end = list.length - 1;

    while (start <= end && !found) {  // Corrected found condition
      int mid = (start + end) / 2;

      if (list[mid] == item) {
        found = true;
        index = mid;
      } else if (list[mid] < item) {
        start = mid + 1;
      } else {
        end = mid - 1;
      }
    }

    return index;
  }
```

Your notes

Linear Search

Your notes

Linear search

What is a Linear Search?

- The linear search is a standard algorithm used to find elements in an **unordered list**. The list is searched **sequentially** and **systematically** from the start to the end one element at a time, **comparing each element** to the **value being searched** for
  - If the **value is found** the algorithm **outputs where it was found in the list**
  - If the **value is not found** it **outputs a message stating it is not in the list**
- An example of using a linear search would be looking for a specific student name in a list or searching for a supermarket item in a shopping list

Performing the linear Search

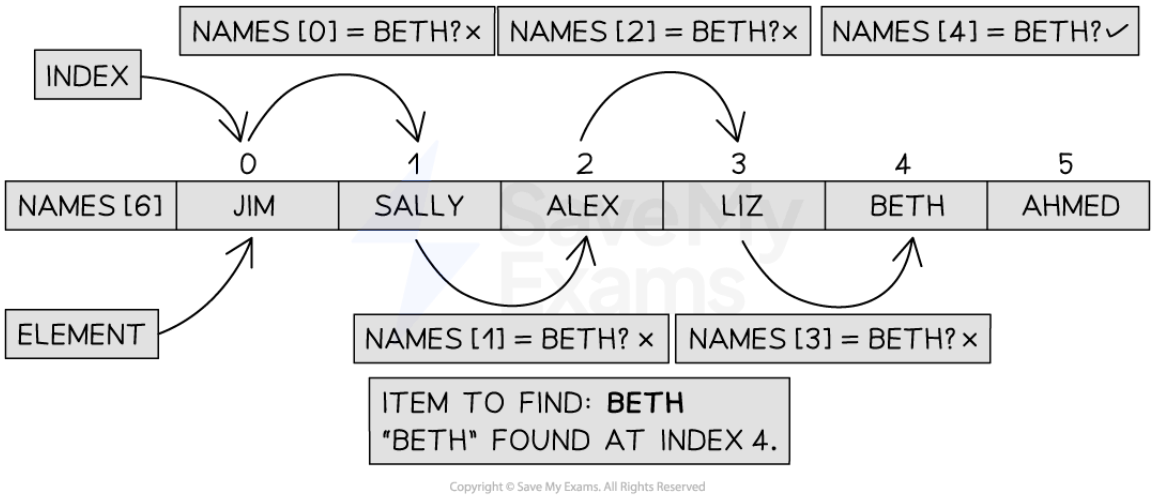


Figure 1: Performing the Linear Search

Time Complexity of a Linear Search

- To determine the algorithm's execution time as measured by the number of instructions or steps carried out, Big-O notation must be used

- The loop is executed n times, once for each item. There are two instructions within the loop, the if statement and an assignment, hence the Big-O for the loop is O(2n)
- There are three initial statements which are O(1)
- The overall Big-O notation is therefore 2n + 3, which when coefficients are factored out, becomes **O(n) as linear time dominates constant time**. The **constant term and coefficients contribute significantly less** as the input size n grows larger
- It is important to remember here that **O(n) is the worst case scenario**. The **best case scenario O(1)** would find the item the first time, while the **average case** would be O(n/2) which when we remove coefficients still becomes **O(n)**


Your notes

Space Complexity of a Linear Search

- As the linear search requires no additional space, it is space complexity O(n) where n is the number of items in the list

Tracing a Linear Search

- Given the following list [5, 4, 7, 1, 3, 8, 9, 2], a trace table for the linear search is shown below

Trace Table of the Linear Search

item	index	i	list[i]	found
8	-1	0	5	False
		1	4	
		2	5	
		3	1	
		4	3	
8	5	5	8	True

Implementing a Linear Search

Pseudocode

```
function linearSearch(list, item)
    index = -1
```

```
    i = 0
    found = False
    while i < len(list) and found = False
        if list[i] = item then
            index = i
            found = True
        endif
        i = i + 1
    endwhile
    return index
endfunction
```

- In the above algorithm, a list of n ordered items is passed to the function and three variables are initialised
- `index = -1` is a default value to indicate "not found", `i = 0` is a counter to ensure the loop starts at the beginning of the list and `found = False` sets a flag to track when/if the value you are searching for is found
- The loop continues as long as the **counter hasn't reached the end of the list** and if `list[i] = item` the current index is stored as the location of the item
- The flag **found** is set to True to **signal the search can be stopped**
- `i = i + 1` increments the counter to **move to the next element** in the list if the value is not found
- After the loop completes the function **returns the final value of index**

Python

```
def linear_search(list, item):
    index = -1
    for i in range(len(list)):
        if list[i] == item:
            index = i
            break  # Stop the loop once the item is found
    return index
```

Java

```
public class LinearSearch {
    public static int linearSearch(int[] list, int item) {
        int index = -1;
        for (int i = 0; i < list.length; i++) {
            if (list[i] == item) {
                index = i;
                break;  // Stop the loop once the item is found
            }
        }
```

Your notes

```
    return index;
}
```

Your notes

Your notes

Bubble Sort

What is a Bubble Sort?

- A Bubble Sort sorts items into order, **smallest to largest**, by **comparing pairs of elements** and **swapping them if they are out of order**
- The first element is compared to the second, the second to the third, the third to the fourth and so on, until the second to last is compared to the last. Swaps occur if each comparison is out of order. This overall process is called a **pass**

Examiner Tips and Tricks

The highest value in the list eventually "bubbles" its way to the top like a fizzy drink, hence the name "Bubble sort"

- Once the **end of the list** has been reached, the **value at the top of the list is now in order** and the **sort resets back to the start of the list**. The **next largest value** is then **sorted** to the top of the list
- More passes are completed until all elements are in the correct order
- A **final pass** checks all elements and if no swaps are made then the sort is complete
- An example of using a bubble sort would be sorting an array of names into alphabetical order, or sorting an array of student marks from a test

Performing the Bubble sort

Your notes

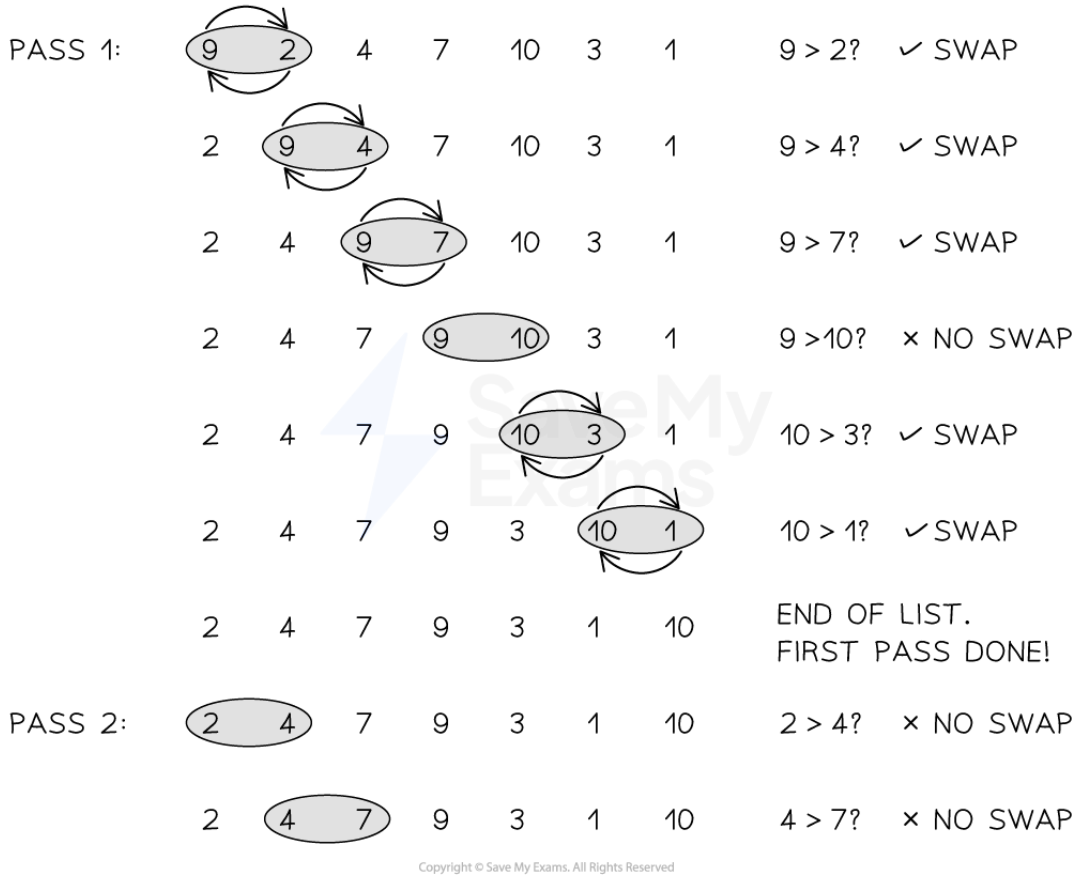


Figure 1: Performing the Bubble sort

Time Complexity of the Bubble Sort

- To determine the algorithm's execution time as measured by the number of instructions or steps carried out Big-O notation must be used
- Four statements are present in the above algorithm O(1), followed by a while loop O(n) containing two statements and a further for loop O(n). This results in the expression: $2n^2 + 4$
- As **coefficients and lower order values are not used**, this give the bubble sort **$O(n^2)$ worst case time complexity**
- The **best case scenario** would be an **already sorted list** which means a time complexity of **O(n)** as each item must still be compared to check if they are in order
- The **average case scenario** would be an **almost sorted list** leading to $O(n^2/2)$, which if coefficients are removed still gives **$O(n^2)$**

Space Complexity of the Bubble Sort


Your notes

- A bubble sort has a space complexity of O(1) because it operates in-place, requiring a fixed amount of memory
- The fixed amount of memory is for variables such as loop counters and a temporary swap variable

Tracing a Bubble Sort

- The trace table follows the algorithm through, updating values that change in the table
- Each iteration of the list reduces the overall size of the list by 1 as after each pass the previously sorted final digit is in order and does not need to be rechecked. This means that 9 is in order and doesn't need to be rechecked, followed by 9 and 7, followed by 9, 7 and 6 and so on
- When the if statement is checked a new row has been added to show the swap of list[j], list[j + 1] and Temp, followed by the subsequent change to Swap from FALSE to TRUE
- After several iterations (that are not shown) the algorithm will eventually output a sorted list

Trace Table of the Bubble Sort

LastElement	Index	Mylist[Index]	Mylist[Index + 1]	Temp	Swap	Output
10	1	5	9		FALSE	
	2	9	4			
		4	9	4	TRUE	
	3	9	2			
		2	9	9		
	4	9	6			
		6	9			
	5	9	7			
		7	9			

	6	9	1			
		1	9			
	7	9	2			
		2	9			
	8	4	9			
		9	4			
	9	9	3			
		3	9			
9	1	5	4		FALSE	
		4	5	5	TRUE	
	2	5	2			
		2	5			
1					FALSE	1, 2, 2, 3, 4, 4, 5, 6, 7, 9


Your notes

Implementing a Bubble Sort

Pseudocode

```
list = [5, 9, 4, 2, 6, 7, 1, 2, 4, 3]
last = len(list)
swap = True
i = 0
while i < (last - 1) and (swap = True)
    swap = False
    for j = 0 to last - i - 2
```

```
        if list[j] > list[j+1]
            temp = list[j]
            list[j] = list[j+1]
            list[j+1] = temp
            swap = True
        endif
    next j
    i = i + 1
endwhile
print(list)
```

- The algorithm above implements an efficient bubble sort in that with each successive iteration, the number of comparisons decreases
    - This is due to the **sorted end of the list not being rechecked** as it is already sorted and therefore inefficient to compare them
    - This is managed by the j for loop where i increments after each iteration and reduces the number of compared items by 1
    - This is because the last iteration's sorted element should already be in the correct position. The overall effect is reducing unnecessary swaps
- Furthermore, using a while loop instead of a for loop means that if a swap does not occur then the list is in order. A for loop would continue comparing even if the items are already in order

Python

```
def bubble_sort(list):
    n = len(list)
    for i in range(n - 1):
        swapped = False
        for j in range(0, n - i - 1):
            if list[j] > list[j + 1]:
                list[j], list[j + 1] = list[j + 1], list[j]
                swapped = True
        if not swapped:
            break  # Early termination if no swaps occurred
    return list
```

Java

```
public class BubbleSort {
    public static void bubbleSort(int[] list) {
        int n = list.length;
        for (int i = 0; i < n - 1; i++) {
            boolean swapped = false;
            for (int j = 0; j < n - i - 1; j++) {
```

Your notes

```
            if (list[j] > list[j + 1]) {
                int temp = list[j];
                list[j] = list[j + 1];
                list[j + 1] = temp;
                swapped = true;
            }
        }
        if (!swapped) {
            break;  // Early termination if no swaps occurred
        }
    }
}
```


Your notes

Insertion Sort

Your notes

Insertion sort

What is an Insertion Sort?

- The insertion sort **sorts one item at a time** by **placing it in the correct position** of an unsorted list. This process repeats until all items are in the correct position
- Specifically, the **current item being sorted is compared to each previous item**. If it is **smaller** than the previous item then the **previous item is moved to the right** and the **current item takes its place**. If the **current item is larger** than the previous item then it is in the **correct position** and the next current item is then sorted as illustrated below

Performing the Insertion sort

Your notes

★ 5 IS ALREADY SORTED AS ITS THE FIRST ITEM

5 9 4 2 7 1 2 4 3

9 < 5? × ∴ NO MOVE, NOW SORTED

5 9 4 2 7 1 2 4 3

SORTED LIST SO FAR

---

5 9 4 2 7 1 2 4 3

4 < 9? ✓ ∴ SHIFT 9 TO RIGHT

---

5 4 9 2 7 1 2 4 3

4 < 5? ✓ ∴ SHIFT 5 TO RIGHT

---

4 5 9 2 7 1 2 4 3

SORTED LIST SO FAR

---

4 5 9 2 7 1 2 4 3

2 < 9? ✓ ∴ SHIFT 9 TO RIGHT
2 < 5? ✓ ∴ SHIFT 5 TO RIGHT
2 < 4? ✓ ∴ SHIFT 4 TO RIGHT

2 4 5 9 7 1 2 4 3

SORTED LIST SO FAR

---

2 4 5 9 7 1 2 4 3

7 < 9? ✓ ∴ SHIFT 9 TO RIGHT

Figure 2: Performing the Insertion sort

Your notes

Time Complexity of an Insertion Sort

- In the above algorithm, one statement is present, a for loop with three statements and a nested while loop that contains two statements
- The time complexity for the insertion sort would be $3n*2n+1$, giving $6n^2+1$. 3n comes from the for loop having three statements inside it, not including the while. 2n comes from the while loop having two statements inside it
- Removing coefficients and dominated factors leaves **$O(n^2)$ as the worst case scenario**
- The **best case scenario** would be an **already sorted list** which means each item is checked one at a time giving a time complexity of **O(n)**

The **average case scenario** would be a **half sorted list** giving $O(n^2/2)$, which when coefficients are removed still gives **$O(n^2)$**

Space Complexity of an Insertion Sort

- As the insertion sort requires no additional space, it is space complexity O(1)

Tracing an Insertion Sort

- Tracing through the insertion sort involves starting at element 1 (not 0)
- i increments through each element one at a time and each item is compared to the one previous. If the previous item is bigger or equal it is set to list[position]
- If position equals 0 then the iteration is at the start of the list and the next iteration begins

Trace Table of the Insertion sort

list	i	item	position	list[position-1]	list[position]

[5, 9, 4, 2, 7, 1, 2, 4, 3]	1	9	1	5	9
	2	4	2	9	9
			1	5	5
			0		
[4, 5, 9, 2, 7, 1, 2, 4, 3]	3	2	3	9	9
			2	5	5
			1	4	4
			0		
[2, 4, 5, 9, 7, 1, 2, 4, 3]	4	7	4	9	9
			3	5	7
[2, 4, 5, 7, 9, 1, 2, 4, 3]	5	1	5	9	9
			4	7	7
			3	5	5
			2	4	4
			1	2	2
			0		1
[1, 2, 4, 5, 7, 9, 2, 4, 3]	6	2	6	9	9
			5	7	7


Your notes

			4	5	5
			3	4	4
			2	2	2
[1, 2, 2, 4, 5, 7, 9, 4, 3]	7	4	7	9	9
			6	7	7
			5	5	5
			4	4	4
[1, 2, 2, 4, 4, 5, 7, 9, 3]	8	3	8	9	9
			7	7	7
			6	5	5
			5	4	4
			4	4	4
			3	2	3
[1, 2, 2, 3, 4, 4, 5, 7, 9]					

Your notes

Worked Example

The following pseudocode procedure performs an insertion sort on the array parameter.

01	procedure insertionSort(dataArray:byRef)
02	for i = 1 to dataArray.Length - 1

03	temp = dataArray[i]
04	tempPos = i - 1
05	exit = false
06	while tempPos >= 0 and exit == false
07	if dataArray[tempPos] < temp then
08	dataArray[tempPos + 1] = dataArray[tempPos]
09	tempPos = tempPos - 1
10	else
11	exit = true
12	endif
13	endwhile
14	dataArray[tempPos + 1] = temp
15	next i
16	endprocedure

Describe how a bubble sort will sort an array of 10 elements.

[6]

Bubble sort compares each pair of adjacent elements [1]. If they are not in the correct order then the elements are swapped [1] and a flag is set to false [1], otherwise they are not swapped [1]. The loop is repeated while the flag is not false [1]. When the end of the array is reached, we return to the start and is the process is repeated for a total of n times [1].

Your notes

Implementing an Insertion Sort

Pseudocode

```
procedure insertsionSort(list)
    n = len(list)
    for i = 1 to n -1
        item = list[i]
        position = i
        while position > 0 and list[position - 1] > item
            list[position] = list[position -1]
            position = position - 1
```

```
            endwhile
            list[position] = item
        next i
endprocedure

list = [5, 9, 4, 2, 7, 1, 2, 4, 3]
insertionSort(list)
print(list)
```

- In the algorithm above, each item is iterated over starting with the first value (9) as the 5 is already in the correct position

Examiner Tips and Tricks

- Remember, in computer science, indexing always starts at 0

- The 9 (the current item) is compared to each previous value using the "position" pointer. While the "position" pointer is greater than the 0th index and the previous item (5) is also larger than 9 then the 5 will be moved right. 5 is less than 9 so no move happens. The 9 is therefore in the correct position
- The process repeats with 4 which is compared to the 9, which is larger and therefore moved right, then 4 is compared to 5, which is larger and is also moved right
- Moving an item right is defined using the "list[position] = list[position-1]" which sets the previous item into the current "position"
- The for loop iterates over all items in the list and each is compared until the correct position is found for each item

Python

```
def insertion_sort(data):
    for i in range(1, len(data)):
        item = data[i]
        position = i
        while position > 0 and data[position - 1] > item:
            data[position] = data[position - 1]
            position -= 1
        data[position] = item
    return data
```

Java

```
public class InsertionSort {
```

Your notes

```
public static void insertionSort(int[] data) {
    for (int i = 1; i < data.length; i++) {
        int item = data[i];
        int position = i;
        while (position > 0 && data[position - 1] > item) {
            data[position] = data[position - 1];
            position--;
        }
        data[position] = item;
    }
}
```

Your notes


Your notes

Merge sort

What is a Merge Sort?

- Merge sort is an example of a **divide and conquer** algorithm that **reduces the size of the problem into smaller problems that are more easily solvable** by an algorithm
- Specifically, the merge sort **divides the list in half**, creating **two sub lists**. Each sub list is then divided into two until each sub list contains only **one item** or element
- Groups of two sub lists are then **sorted** and **merged** together to form new, larger sub lists until all sub lists have been merged into a single sorted list
- The merge sort therefore contains two parts:
  - **Part one**: Divide the list into sub lists until each sub list contains only one element
  - **Part two**: Repeatedly merge groups of two and sort them, until all lists have been merged into a single sorted list
- The merge sort has been illustrated below using a list of ten items: [15, 5, 2, 7, 4, 9, 10, 1, 8, 3]

Performing the Merge sort

Your notes

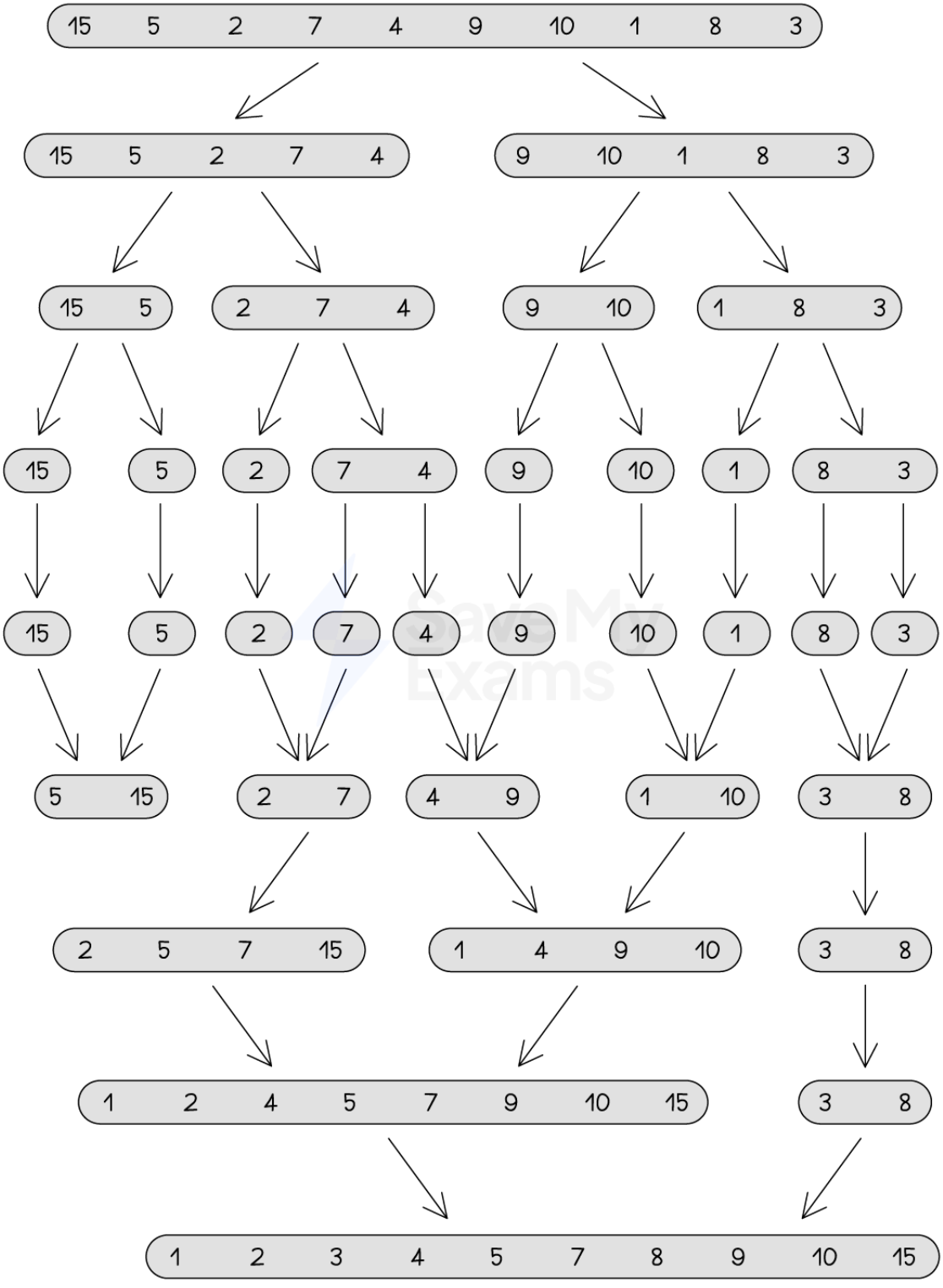


Figure 1: Performing the Merge sort

Your notes

- Notice that halving the list sometimes produces odd numbers of elements in sub lists. When dividing, all sub lists must contain a single element before merging can begin.
- When merging, **only two sub lists can be merged at a time**. Any left over sub lists are merged in the next merging iteration
- In order to merge two sub lists, two pointers are required, one for each sub list. The **pointer** keeps track of **which elements are to be compared**. Once an element has been merged into the new list, the corresponding **pointer is incremented**. The process continues until a list contains **no elements** to compare, at which point the remaining elements are **appended** to the end of the new merged sub list.

Time Complexity of a Merge Sort

- To determine the algorithm's execution time as measured by the number of instructions or steps carried out Big-O notation must be used
- The time complexity of a merge sort is dependent on its divide and conquer nature O(logn) and the fact that n sub lists must be merged. This means the overall **worst case time complexity** of the merge sort is **O(nlogn)**
- The **best case** and **average case** scenario time complexity of the merge sort is **O(nlogn)** as regardless of the number of elements or item, n must still be merged and halved

Space Complexity of a Merge Sort

- The merge sort requires twice the amount of space of other sorting algorithms due to holding copies of the left and right halves of the list. Its space complexity is therefore **O(n)** original space and an **additional O(n)** space to store the copies of the list

Tracing a Merge Sort

- In the trace table below, several calls to merge sort are made which splits the list. Each call generates a **stack frame** which is put on the **stack**. **Stack frames are removed when the current stack frame reaches the end of its code**
- The list is gradually broken down into sub lists, starting with the left side of each sub list. Sub lists are created until each sub list contains only one element which is the **base case**. When a **base case is reached** the stack frame is removed and the **program continues execution on the line after the call to merge sort**
- Each sub list is gradually merged, two at a time and passed up the stack to be merged with another sub list in a later merge sort call
- Pointers are incremented during the merge such that the next element to merge in each sub list is tracked and appropriately added to the new list. NewListPointer tracks the position of elements in the

new list

- Row 25 shows the algorithm completing the left side of the list and the beginning of the merge of the right sub list which repeats the process outlined in the trace table


Your notes

Trace Table for the Merge Sort

Row	Stack frame	len(list) > 1?	left	right	Split left, right or merge?	newList	mid	Left Pointer	Right Pointer	newListPointer
1	1	TRUE	15, 5, 2, 7, 4	9, 10, 1, 8, 3	LEFT		5			
2	2	TRUE	15, 5	2, 7, 4	LEFT		2			
3	3	TRUE	15	5	LEFT		0			
4	4	FALSE			BASE CASE					
5	3	TRUE	15	5	RIGHT		0			
6	4	FALSE			BASE CASE					
7	3	TRUE	15	5	MERGE	[5, 15]	0	0	0	0
8	3	TRUE	2	7, 4	LEFT		1			
9	4	FALSE			BASE CASE					
10	3	TRUE	7	4	LEFT		0			
11	4	FALSE			BASE CASE					

12	3	TRUE	7	4	RIGHT		0			
13	4	FALSE			BASE CASE					
14	3	TRUE	7	4	MERGE	[4, 7]	0	0	0	0
15	3	TRUE	2	4, 7	MERGE		1	0	0	0
16						[2]		1		1
17						[2, 4]			1	2
18						[2, 4, 7]			2	3
19	3	TRUE	5, 15	2, 4, 7	MERGE		2	0	0	0
20						[2]			1	1
21						[2, 4]			2	2
22						[2, 4, 5]		1	2	3
23						[2, 4, 5, 7]		1	3	4
24						[2, 4, 5, 7, 15]		1	4	5
25	2	TRUE	9, 10	1, 8 3	LEFT		2			

Your notes

Examiner Tips and Tricks

Merge sort is a complex algorithm. You will not be asked to derive its time or space complexity in detail.

- You will however be expected to know and justify its time and space complexity


Your notes

Implementing a merge sort

Pseudocode

```
procedure mergesort(list)
    if len(list) > 1 then
        mid = len(list) div 2
        left = list[:mid]
        right = list[mid:]

        mergesort(left)
        mergesort(right)

        leftPointer = 0
        rightPointer = 0
        newListPointer = 0
        while leftPointer < len(left) and rightPointer < len(right)
            if left[leftPointer] < right[rightPointer] then
                list[newListPointer] = left[leftPointer]
                leftPointer = leftPointer + 1
            else
                list[newListPointer] = right[rightPointer]
                rightPointer = rightPointer + 1
            endif

            newListPointer = newListPointer + 1
        endwhile

        while leftPointer < len(left)
            list[newListPointer] = left[leftPointer]
            leftPointer = leftPointer + 1
            newListPointer = newListPointer + 1
        endwhile

        while rightPointer < len(right)
            list[newListPointer] = right[rightPointer]
            rightPointer = rightPointer + 1
            newListPointer = newListPointer + 1
        endwhile
    endif
endprocedure

list = [15, 5, 2, 7, 4, 9, 10, 1, 8, 3]
mergesort(list)
print(list)
```


Your notes

- It is important to note that the merge sort uses **[popover id="N-tIO9JV~O-0w97n" label="recursion"]** in order to work. To divide the list into sub lists, the merge sort algorithm must call itself, using a **[popover id="5UjdORCO7kg2tW1L" label="base case"]** of one element. Repeated calls to the algorithm are put on the **[popover id="vZKPSwu5BsgHPpV_" label="stack"]**. Once each list is composed on one element, **[popover id="eAY1NI_y6NFvH4Le" label="stack frame"]** are removed from the stack and processed
- It is furthermore important to note that **[:mid]** means **"all elements up to but not including the mid value"** and **[mid:]** means **"all elements starting from and including the mid value"**
  - This technique is known as **slicing**
- Once the list has been split, and no more recursive calls are made, three new variables are defined per stack frame:
  - **leftPointer** (keeps track of the left sub list)
  - **rightPointer** (keeps track of the right sub list)
  - **newListPointer** (keeps track of the next available space in the sorted list)
- The new list is merged in two stages:
  - Stage one: Merge all elements **until one list is empty**
    - The first while loop manages this process. The left and right pointers increment over the left and right lists, comparing elements one at a time. If one is smaller than the other then it is placed in the next available space in the new list
  - Stage two: **Append the non-empty list's** remaining elements to the new list
    - This is managed by the final two while loops. One merges if the left half has remaining elements. The other merges if the right half has remaining elements. Only one will be called, not both

Python

```
def merge_sort(list):
  if len(list) > 1:
    mid = len(list) // 2
    left = list[:mid]
    right = list[mid:]
    merge_sort(left)
    merge_sort(right)
    left_pointer, right_pointer, new_list_pointer = 0, 0, 0
    new_list = [None] * len(list)
    while left_pointer < len(left) and right_pointer < len(right):
      if left[left_pointer] < right[right_pointer]:
        new_list[new_list_pointer] = left[left_pointer]
        left_pointer = left_pointer + 1
```

```
    else:
      new_list[new_list_pointer] = right[right_pointer]
      right_pointer = right_pointer + 1
    new_list_pointer = new_list_pointer + 1
  while left_pointer < len(left):
    new_list[new_list_pointer] = left[left_pointer]
    left_pointer = left_pointer + 1
    new_list_pointer = new_list_pointer + 1
  while right_pointer < len(right):
    new_list[new_list_pointer] = right[right_pointer]
    right_pointer = right_pointer + 1
    new_list_pointer = new_list_pointer + 1
  list[:] = new_list
 return list

list = [15, 5, 2, 7, 4, 9, 10, 1, 8, 3]
merge_sort(list)
print(list)  # Output: [1, 2, 3, 4, 5, 7, 8, 9, 10, 15]
```

Java

```
public class MergeSort {

  public static void mergeSort(int[] data) {
    if (data.length > 1) {
      int mid = data.length / 2;
      int[] left = new int[mid];
      int[] right = new int[data.length - mid];
      System.arraycopy(data, 0, left, 0, mid);
      System.arraycopy(data, mid, right, 0, data.length - mid);
      mergeSort(left);
      mergeSort(right);
      merge(data, left, right);
    }
  }

  private static void merge(int[] data, int[] left, int[] right) {
    int leftPointer = 0, rightPointer = 0, newListPointer = 0;
    while (leftPointer < left.length && rightPointer < right.length) {
      if (left[leftPointer] < right[rightPointer]) {
        data[newListPointer] = left[leftPointer];
        leftPointer++;
      } else {
        data[newListPointer] = right[rightPointer];
        rightPointer++;
      }
      newListPointer++;
    }
    System.arraycopy(left, leftPointer, data, newListPointer, left.length - leftPointer);
    System.arraycopy(right, rightPointer, data, newListPointer, right.length - rightPointer);
  }
```


Your notes

```
    public static void main(String[] args) {
        int[] data = {15, 5, 2, 7, 4, 9, 10, 1, 8, 3};
        mergeSort(data);
        System.out.println(Arrays.toString(data));  // Output: [1, 2, 3, 4, 5, 7, 8, 9, 10, 15]
    }
}
```


Your notes

Quick Sort

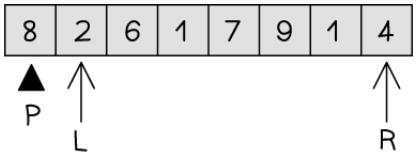
Your notes

Quick sort

What is a Quick Sort?

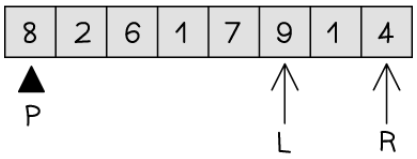
- Quick sort is another example of a divide and conquer algorithm that **reduces the size of the problem into smaller problems that are more easily solvable** by an algorithm
- Unlike the merge sort however, quick sort does not use any additional storage
- Below is an illustration of the quick sort

Your notes



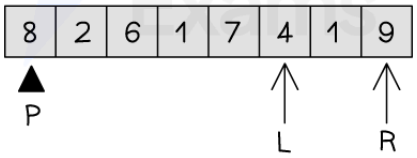
1 SET THE PIVOT TO THE FIRST VALUE. SET LEFT POINTER TO **PIVOT +1**, SET RIGHT POINTER TO END OF LIST

2 INCREMENT LEFT UNTIL ARRAY [LEFT] IS GREATER THAN THE PIVOT

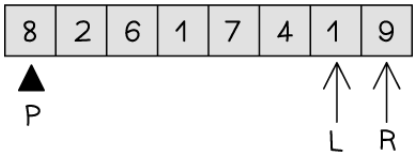


3 DECREMENT RIGHT UNTIL ARRAY [RIGHT] IS LESS THAN THE PIVOT 4 < 8 ALREADY SO RIGHT DOES NOT MOVE

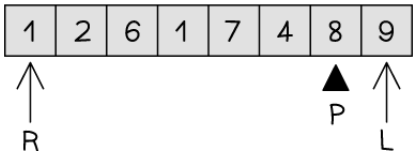
4 SWAP LEFT AND NIGHT



5 CONTINUE WITH STEPS **2** TO **4** UNTIL THE LEFT AND RIGHT POINTERS CROSS EACH OTHER



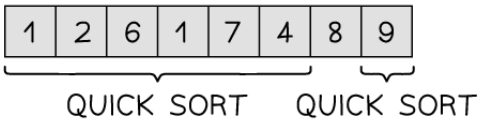
6 SWAP THE RIGHT POINTS VALUE WITH THE PIVOT. THE PIVOT IS NOW SORTED

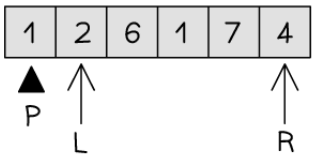


Your notes

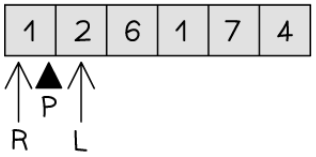
Your notes

7 RECURSIVELY CALL THE FUNCTION ON ALL ITEMS BELOW THE PIVOT AND ALL ITEMS ABOVE THE PIVOT

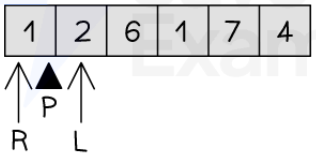




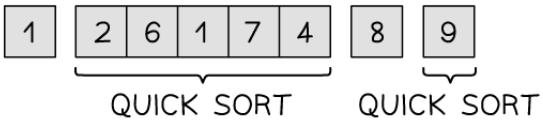
8 REPEAT STEPS **2 – 4** ON THE LIST



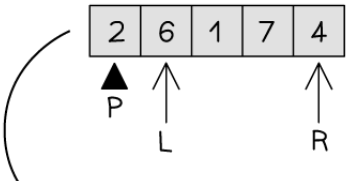
LEFT POINTER DOESN'T MOVE AS 2 IS GREATER THAN 1.
RIGHT POINTER MOVES UNTIL IT IS LESS THAN LEFT POINTER.
SWAP PIVOT AND RIGHT POINTER 1⇆1

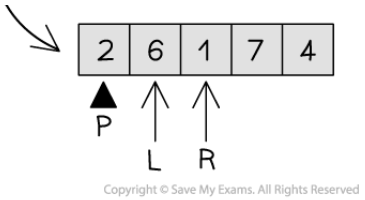


1 IS NOW IN THE CORRECT POSITION AND IS SORTED.
PERFORM STEP **1** ON THE LIST



9 REPEAT STEPS **2 –7** ON THE NEW SUBLISTS





Your notes

Your notes

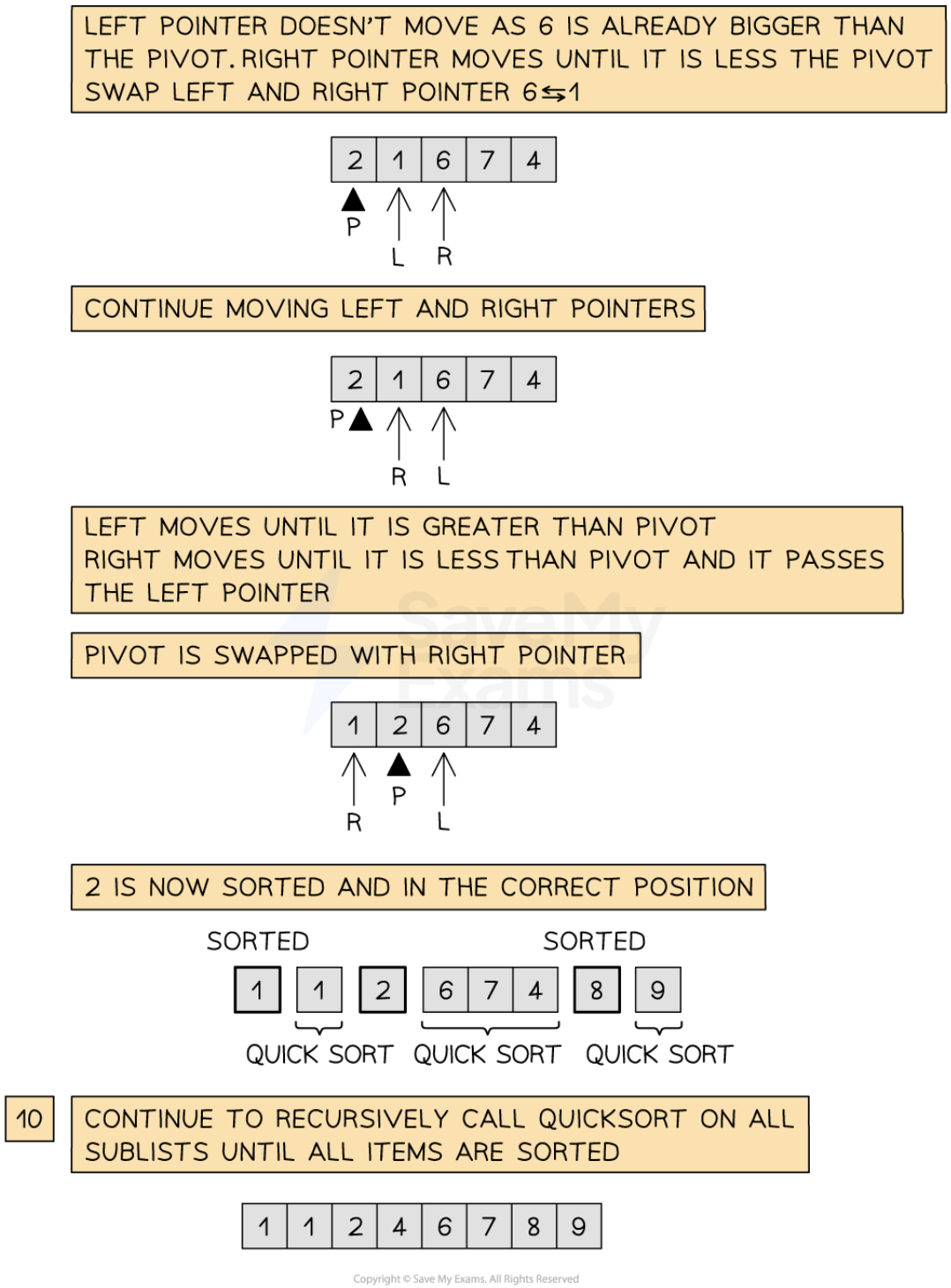


Figure 2: Performing the Quick sort

Time Complexity of a Quick Sort


Your notes

- Like merge sort, quick sort is a divide and conquer algorithm which means the problem size is reduced, creating an easier problem to solve. This gives quick sort a time complexity of **O(logn)**. Like merge sort it must sort **n lists**, giving it an overall **best case** time complexity of **O(nlogn)**
- Unlike merge sort, quick sort has a **worst case time complexity of O(n^2)**. This is due to the **potential placement of the pivot values**
  - If the pivot value is placed roughly in the middle of the list, then quick sort performs optimally
  - If the pivot is sorted at the **start** of the list, **all items are above** this value. If the pivot is sorted at the **end** of the list, **all items are below** this value
  - As the sorted pivots are at the extremes of the list this means there is only **one call to quick sort** rather than two, as **no second sublist exists past the extreme**. Each subsequent list has a size of n-1 with n operations performed, leading to n*(n-1) or **O(n^2)**
  - Quick sort also has the problem that if the **list is too large or pivots aren't optimally chosen** and recursion continues for too long, this can cause a **stack overflow** as too many stack frames have been placed on the stack

Space Complexity of a Quick Sort

- Quick sort requires additional memory as copies of lists are made and put on the stack. The overall space complexity is therefore **O(n)** with **O(n) additional space** required

Tracing a Quick Sort

- The trace table below shows the process of calling the quick sort and moving the left and right pointers up and down the list to find elements to swap and then find the correct position of the pivot
- Once the pivot is in place, quick sort is recursively called on both sublists on either side of the pivot. Note that a second call to quick sort may not exist if the pivot value ends on either extreme of the list
- The process repeats, calling quick sort on progressively smaller sublists until all values are in the correct order

Trace Table for the Quick sort

list	operation	done	pivot	list[pivot]	left	list[left]	right	list[right]
[8, 2, 6, 1, 7, 9, 1, 4]	QUICKSORT	False	0	8	1	2	7	4
	INC LEFT				1	2		

					2	6		
					3	1		
					4	7		
					5	9		
	DEC RIGHT						7	4
[8, 2, 6, 1, 7, 4, 1, 9]	SWAP				5	4	7	9
	INC LEFT				6	1		
					7	9		
	DEC RIGHT						6	1
[1, 2, 6, 1, 7, 4, 8, 9]	SWAP PIVOT	True	6	8	7	9	0	1
[1, 2, 6, 1, 7, 4]	QUICKSORT	False	0	1	1	2	5	4
	INC LEFT				1	2		
	DEC RIGHT						5	4
							4	7
							3	1
							2	6
							1	2
							0	1

Your notes

	SWAP PIVOT	True	0	1	1	2	0	1
[2, 6, 1, 7, 4]	QUICK SORT							
CONTINUE UNTIL ALL ITEMS SWAPPED								
[1, 1, 2, 4, 6, 7, 8, 9]								

Your notes

Worked Example

A 1-dimensional array stores a set of numbered cards from 0 to 7. An example of this data is shown below

[2, 0, 1, 7, 4, 3, 5, 6]

A programmer is writing a computer program to sort the cards into the correct order (0 to 7).

i) Show how an insertion sort would sort the array in above into the correct order. Draw the array after each move

3 marks

ii) Describe how a quick sort algorithm works with the data in shown above.

6 marks

i)

[2, 0, 1, 7, 4, 3, 5, 6]

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

Sort the 0 into place

[0, 1, 2, 4, 7, 3, 5, 6]

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

Sort 4 and 3 into place

[0, 1, 2, 3, 4, 7, 5, 6]

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

Sort 5 and 6 into place

ii) Quick sort uses divide and conquer to split the list into sublists **[1]**. The first item becomes the pivot, so 2 is the pivot here **[1]**. Compare each item to the pivot, 0 to 2, 1 to 2, etc **[1]**. Two lists are created, one with items less than the pivot [0, 1] **[1]** and one with items more than the pivot [7, 4, 3, 5, 6] **[1]**. Quick sort is called on the new sublists and the lists are eventually recombined. **[1]**


Your notes

Implementing a Quick Sort

Pseudocode

```
function quicksort(list, start, end)

    if start <= end then
        return
    else
        pivot = list[start]
        left = start + 1
        right = end
        done = False
        while done == False
            while left <= right and list[left] <= pivot
                left = left + 1
            endwhile

            while right >= left and list[right] >= pivot
                right = right - 1
            endwhile

            if left < right then
                temp = list[left]
                list[left] = list[right]
                list[right] = temp
            else
                done = True
            endif
        endwhile

        temp = items[start]
        list[start] = list[right]
        list[right] = temp

        quicksort(list, start, right - 1)
        quicksort(list, right + 1, end)
    endif

    return list
endfunction
```

- The quick sort works in several stages

- 1) The **pivot** is set to the **0th element**, the **left pointer** is set to the **1st element** and the **right pointer** is set to the **end of the list**
- 2) The **left pointer increments** while it is **less than or equal to the right pointer** and the **value indexed** by the left pointer is **less than or equal to the pivot value**. Once the left pointer finds a **value bigger than the pivot**, or it **passes the right pointer it stops**
- 3) After the left pointer has been incremented, the right pointer is decremented. While the **right pointer** is **greater than or equal to the left pointer** and the **value indexed** by the right pointer is **greater than or equal to the pivot value**, the **right pointer is decremented**
- 4) If both **pointers have not yet passed each other** but both have **finished moving**, the **values indexed** by both pointers are **swapped**. The pointers then continue moving as per step 2 and 3
- 5) If both **pointers pass each other** then the **swapping is done**. The **pivot value** is then **swapped** with the **value indexed** by the **right pointer**. The pivot value is now in the correct position and is sorted
- 6) Both sides of the pivot value now exist as two new sublists. **Quicksort is recursively called on both sublists** and the process repeats as per step 1
- 7) Steps 1–7 repeat until all sublists have been sorted


Your notes

Python

```
def quicksort(data, start, end):
  if start <= end:
    pivot = data[start]
    left = start + 1
    right = end
    done = False
    while not done:
      while left <= right and data[left] <= pivot:
        left += 1
      while right >= left and data[right] >= pivot:
        right -= 1
      if left < right:
        temp = data[left]
        data[left] = data[right]
        data[right] = temp
      else:
        done = True
    temp = data[start]
    data[start] = data[right]
    data[right] = temp
    quicksort(data, start, right - 1)
```

```
    quicksort(data, right + 1, end)
  return data

data = [5, 2, 7, 4, 9, 10, 1, 8, 3]
quicksort(data, 0, len(data) - 1)
print(data)  # Output: [1, 2, 3, 4, 5, 7, 8, 9, 10]
```

Java

```
public class QuickSort {

    public static void quickSort(int[] data, int start, int end) {
        if (start < end) {
            int pivotIndex = partition(data, start, end);
            quickSort(data, start, pivotIndex - 1);
            quickSort(data, pivotIndex + 1, end);
        }
    }

    private static int partition(int[] data, int start, int end) {
        int pivot = data[start];
        int left = start + 1;
        int right = end;
        while (true) {
            while (left <= right && data[left] <= pivot) {
                left++;
            }
            while (right >= left && data[right] >= pivot) {
                right--;
            }
            if (left < right) {
                swap(data, left, right);
            } else {
                break;
            }
        }
        swap(data, start, right);
        return right;
    }

    private static void swap(int[] data, int i, int j) {
        int temp = data[i];
        data[i] = data[j];
        data[j] = temp;
    }
```

Your notes

```
    public static void main(String[] args) {
        int[] data = {5, 2, 7, 4, 9, 10, 1, 8, 3};
        quickSort(data, 0, data.length - 1);
        System.out.println(Arrays.toString(data));  // Output: [1, 2, 3, 4, 5, 7, 8, 9, 10]
    }
}
```


Your notes

Your notes

Dijkstra's Shortest Path Algorithm

What is Dijkstra's Shortest Path Algorithm?

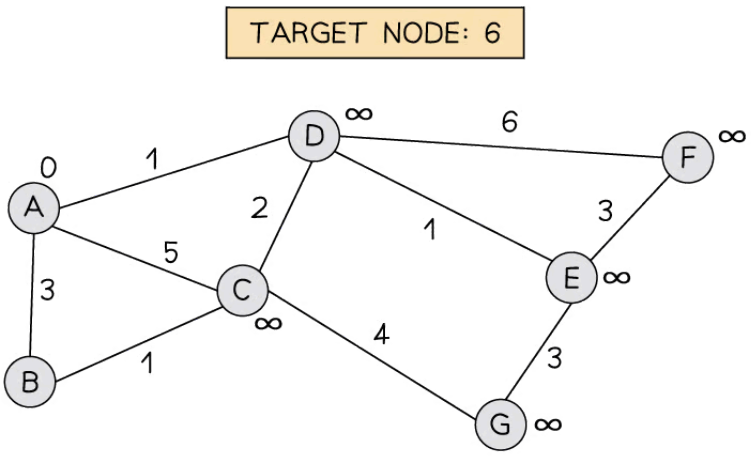
- Dijsktra's shortest path algorithm is an example of an optimisation algorithm that calculates the shortest path from a starting location to all other locations
- Dijkstra's uses a **weighted graph** in a similar manner to a breadth first search to exhaust all possible routes to find the optimal route to the destination node
- To revise Graphs you can follow this link
- Each route is represented as an **arc/edge** from a **node/vertex**. Each edge carries a **weighted value** which represents some **measurement, for example time, distance or cost**
- An illustrated example of Dijsktras is shown below

What is an Optimisation Problem?

- Optimisation problems involve **maximising the efficiency** of a solution to solve the stated problem. Examples of optimisation problems can include:
  - Finding the shortest route between two points. This is used for planning journeys, creating networks, building circuit boards, etc.
  - Minimising resource usage in manufacturing or games
  - Timetabling lessons in school or office meetings
  - Scheduling staff breaks and work hours
- The most common optimisation problem encountered in daily life is finding the shortest route from A to B. Various apps exist to calculate these distances, such as Google Maps or road trip planners

Performing Dijkstra's Shortest Path Algorithm

Your notes

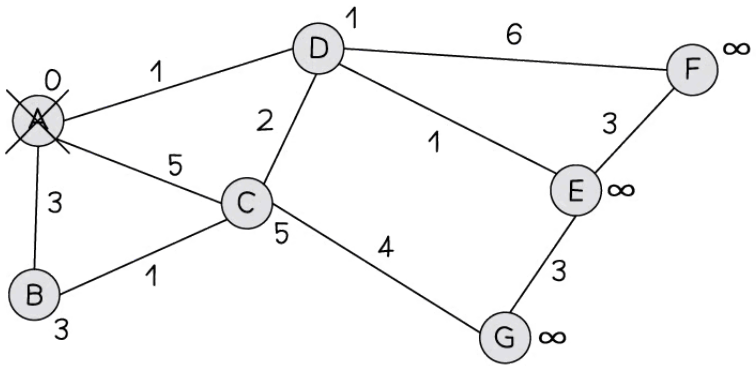


VISITED

Node	Weight + path	Previuos
A	0	–
B	∞	–
C	∞	–
D	∞	–
E	∞	–
F	∞	–
G	∞	–

1) Set A path to 0 and all other path weights to infinity

Your notes

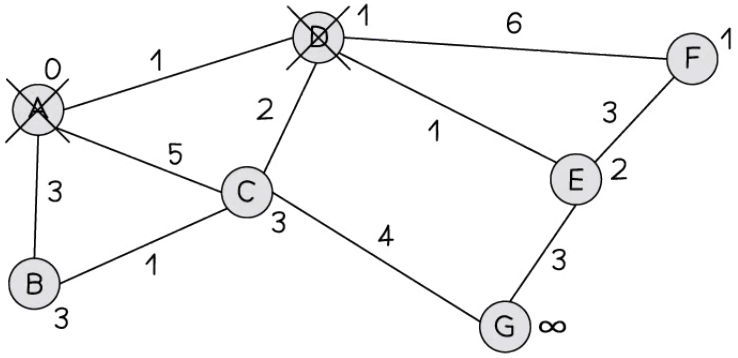


VISITED
A

Node	Weight + path	Previuos
~~A~~	~~0~~	~~–~~
B	3	–
C	5	–
D	1	–
E	∞	–
F	∞	–
G	∞	–

2) Visit A. Update all neighbouring nodes path weights to the distance from A + A's path weight (0)

Your notes

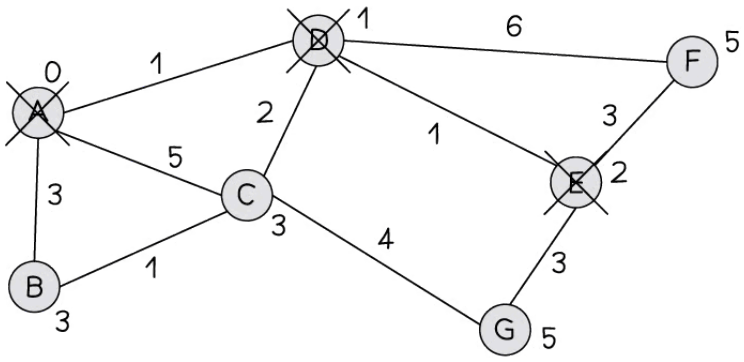


VISITED
AD

Node	Weight + path	Previuos
~~A~~	~~0~~	~~–~~
B	3	–
C	3	–
~~D~~	~~1~~	~~A~~
E	2	–
F	7	–
G	∞	–

3) Choose the next node to visit that has the lowest path weight (D). Visit D. Update all neighbouring, non-visited nodes with a new path weight if the old path weight is bigger than the new path weight. C is updated from 5 to 3 as the new path is shorted (A > D > C)

Your notes

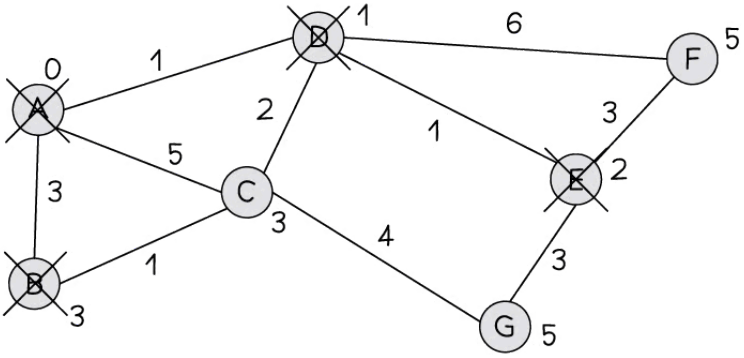


VISITED
ADE

Node	Weight + path	Previuos
~~A~~	~~0~~	~~–~~
B	3	–
C	3	–
~~D~~	~~1~~	~~A~~
~~E~~	~~2~~	~~D~~
F	5	–
G	5	–

4) Choose the next node to visit that has the lowest path weight (E). Visit E. Update all neighbouring, non-visited nodes from E with a new path weight. F is updated from 7 to 5 as the new path is shorter (A > D > E > F)

Your notes

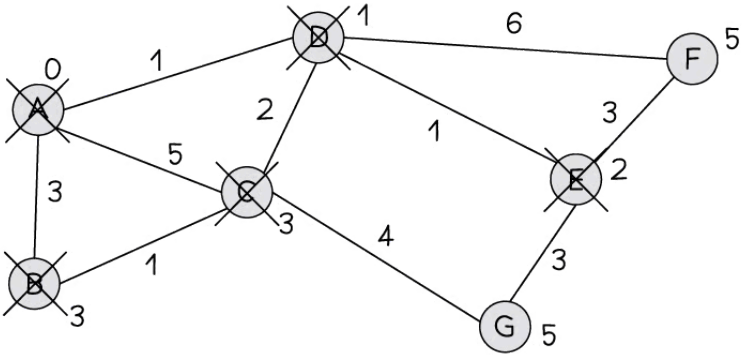


VISITED
ADEB

Node	Weight + path	Previuos
~~A~~	~~0~~	~~–~~
~~B~~	~~3~~	~~A~~
C	3	–
~~D~~	~~1~~	~~A~~
~~E~~	~~2~~	~~D~~
F	5	–
G	5	–

5) Choose the next node to visit, alphabetically (B). Visit B. Update all neighbouring non-visited nodes from B with a new weight if the path is less than the current path. C is not updated as A > B > C is 4

Your notes

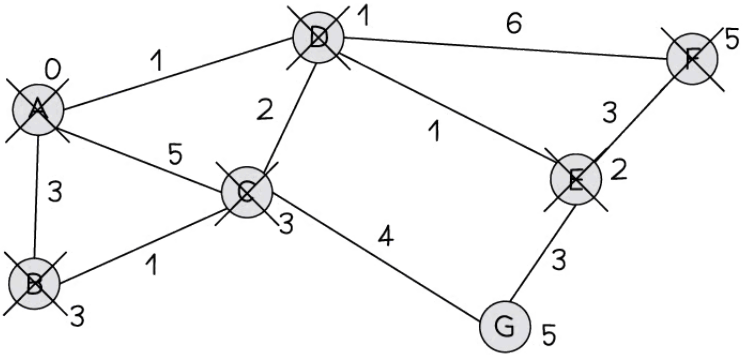


VISITED
ADEBC

Node	Weight + path	Previuos
~~A~~	~~0~~	~~–~~
~~B~~	~~3~~	~~A~~
~~C~~	~~3~~	~~D~~
~~D~~	~~1~~	~~A~~
~~E~~	~~2~~	~~D~~
F	5	–
G	5	–

6) Choose the next lowest node and visit (C). Update C's neighbours if the new path is shorter than the old path. A > D > C > G is 8, so G is not updated

Your notes

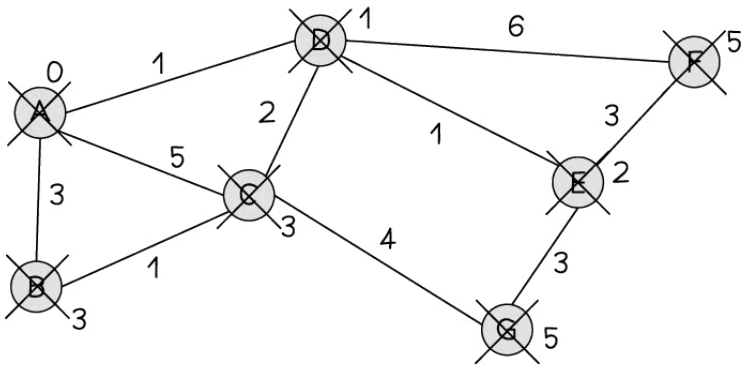


VISITED
ADEBCF

Node	Weight + path	Previuos
~~A~~	~~0~~	~~–~~
~~B~~	~~3~~	~~A~~
~~C~~	~~3~~	~~D~~
~~D~~	~~1~~	~~A~~
~~E~~	~~2~~	~~D~~
~~F~~	~~5~~	~~E~~
G	5	–

7) Choose the next lowest node and visit (F). Update F's neighbours if the new path is shorter than the old path. F has no non-visited neighbours so no updates are made

Your notes



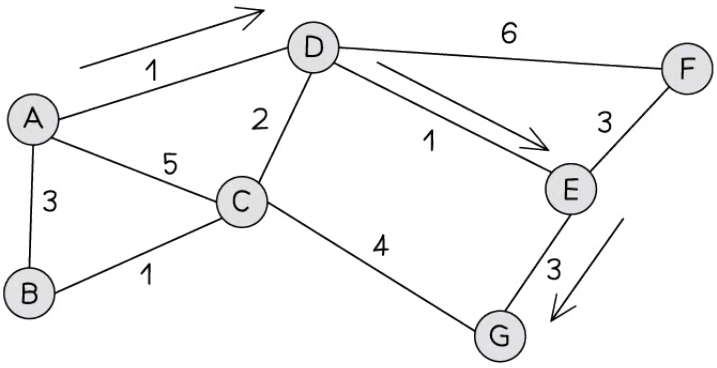
VISITED
ADEBCFG

Node	Weight + path	Previuos
A	0	–
B	3	A
C	3	D
D	1	A
E	2	D
F	5	E
G	5	E

8) Choose the next lowest node and visit (G). G has no non-visited neighbours so no updates are made

9) All nodes have been visited. Back tracking from G gives the following order and total distance A-D-E-G (5)

Your notes



VISITED
ADEBCFG

Node	Weight + path	Previous
A	0	–
B	3	A
C	3	D
D	1	A
E	2	D
F	5	E
G	5	E

Figure 1: Performing Dijkstra's Shortest path Algorithm

- The algorithm for Diskstras is shown below, both in structured english and a pseudocode format

Dijkstra's Shortest path Algorithm (Structured English)

```
assign a distance value of 0 to the initial node and infinity for every other node
add all nodes to a priority queue, sorted by current distance
while the queue is not empty
    remove node x from the front of the queue and add to a visited list
    for each unvisited neighbour y of the current node x
        newDistance = distanceAtX + distanceFromXtoY
        if newDistance < distanceAtY then
            distanceAtY = newDistance
            reorder the priority queue by changing Y's position based on the new distance
        endif
    next X
endwhile
```

Dijkstra's Shortest path Algorithm (Pseudocode)

```
function Dijkstra(graph, start, goal)
    //set all distances to infinity and first distance to 0
for node in graph
        distance[node] = infinity
    next node
    distance[start] = 0


    while unvisitedNodes in graph
        //find the shortest distance from all the nodes in order to select the next node to visit and expand
        min = null
        for node in graph
            if min == null then
                min = node
            elseif distance[node] < distance[min] then
                min = node
            endif
        next node

        //for each neighbour, update the cost if the current cost and distance are less than the previously stored distance
        for neighbour in graph
            if cost + distance[min] < distance[neighbour] then
                distance[neighbour] = cost + distance[min]
                previousNode[neighbour] = min
            endif
        next neighbour

        visited.append(node)
    endwhile

    //backtrack through the previous nodes and build up the final path until the start is reached
    node = goal
while node != start
    path.append(node)
    node = previousNode[node]
endwhile
```


Your notes

```
    return path
endfunction
```

Your notes

Examiner Tips and Tricks

You will not be asked to program Diksjtras in the exam as the algorithm is too complex to implement. You may be asked about parts of the algorithm however

- In the first above algorithm, a **priority queue** is used to order the nodes in terms of distance The shortest routes are prioritized to be explored next. In the second algorithm above, a simple loop finds the next shortest node in the list. Both are valid methods
- Once a node has been **removed from the queue** it is **added to a visited list** and is not rechecked
- It is important to note that the algorithm is **exhaustive** and **will check every available node and path**, calculating the shortest distance from the initial node to every other node
- As Dijkstra's is a graph traversal algorithm, a **dictionary** storing the nodes and neighbours will need to be passed to the algorithm
- A corresponding dictionary of distances from the starting node to each node will also need to be kept during the algorithm

Examiner Tips and Tricks

You do not need to know the time or space complexity for Dijkstra's shortest path algorithm

Tracing Dijkstra's Shortest path Algorithm

- Tracing Dijkstra's shortest path algorithm can be done in a manner similar to Figure 1 above
- Remember to **initialize the starting node to 0 and all other nodes to infinity**
- Update each nodes neighbours by **adding the previous path and the distance from the current node to the neighbour**
- **Keep track of each nodes previous node** that offered the shortest route

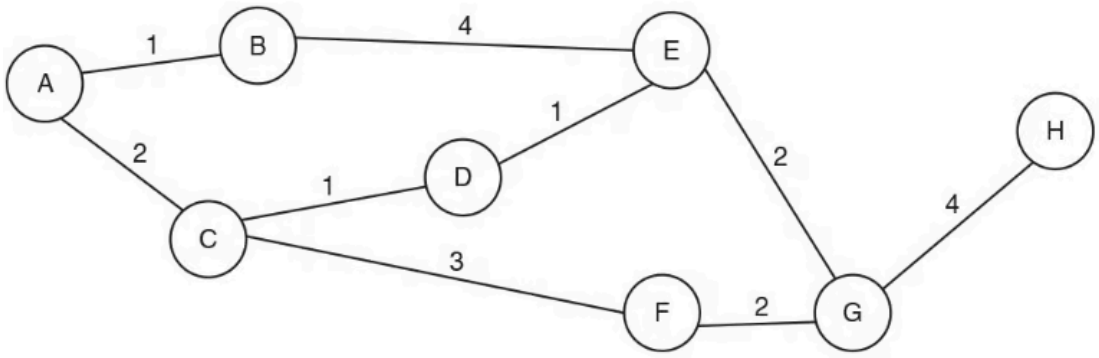
Worked Example

Your notes

Mabel is a software engineer. She is writing a computer game for a client. In the game the main character has to avoid their enemies. This becomes more difficult as the levels of the game increase.

The game's 'challenging' level has intelligent enemies that hunt down the character in an attempt to stop the user from winning. The program plans the enemies' moves in advance to identify the most efficient way to stop the user from winning the game.

The possible moves are shown in a graph. Each node represents a different state in the game. The lines represent the number of moves it will take to get to that state.



Show how Dijkstra's algorithm would find the shortest path from A to H.

6 marks

Node	Visited	From A	Previous Node	
A	✓	0	-	**[1]**
B	✓	1	A	**[1]**
C	✓	2	A	
D	✓	3	C	**[1]**
F	✓	5	C	
E	✓	5 4	B D	**[1]**
G	✓	6	E	**[1]**
H		10	G	

Final route: ACDEGH (length 10) **[1]**

Only one mark is available for the final route. Remember to show your working. You won't get given the table in the exam, just lots of lines but you can draw a table like the one above

Your notes

A* Algorithm

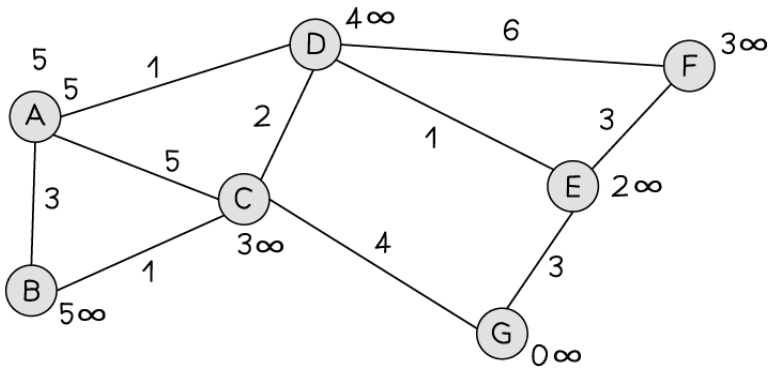
Your notes

A* Search Algorithm

What is A* Search Algorithm?

- A* search builds on Dijkstra's by using a **heuristic**. A heuristic is a **rule of thumb, best guess or estimate** that helps the algorithm accurately approximate the shortest path to each node
- Dijkstra's shortest path algorithm is an example of a path-finding algorithm. Other path finding algorithms exist that are more efficient. One example is the A* search
- Dijkstra's uses a single cost function i.e. the weight of the current path plus the total weight of the path so far. This is the **real distance** from the initial node to every other node
- To revise Graphs you can follow this link
- The **cost function is unreliable**, for example, going from node A to B may be a large distance such as 117, whereas going from A to C may be a distance of 5. Dijkstra's will follow node C even if it doesn't lead to the goal node as A to C is simply the shortest path
- In order to prevent going in the wrong direction, another function is necessary: a **heuristic function**
- A* search uses **g(x) as the real distance cost function and h(x) as the heuristic function**
- Every node will have a heuristic value. The purpose of the **heuristic function** is that it will **approximate the Euclidean distance i.e. the straight line distance from the current node to the goal node**. By following nodes based on shorter heuristic distances, the algorithm can be sure it is getting closer to the goal node
- As the heuristic is an estimate, it is **not necessarily optimum**. The A* search operates under the assumption that the **heuristic function should never overestimate the real cost** from the initial node to the goal node. The real cost should always be greater or equal to the heuristic function
- The **total cost f(x)** of each node rather than just g(x) (the distance from node X to Y plus the previous route total distance) **is g(x) + h(x)** (the estimate of how far the goal node is away from the current node)
  - The calculation for each node is therefore $f(x) = g(x) + h(x)$
  - **If h(x) increases** compared to the previous node, it means the algorithm is **traveling further away** from the goal node
- A* search **focuses on reaching the goal node** unlike Dijkstra's which calculates the distance from the initial node to every other node
- Below is an illustration of the A* search

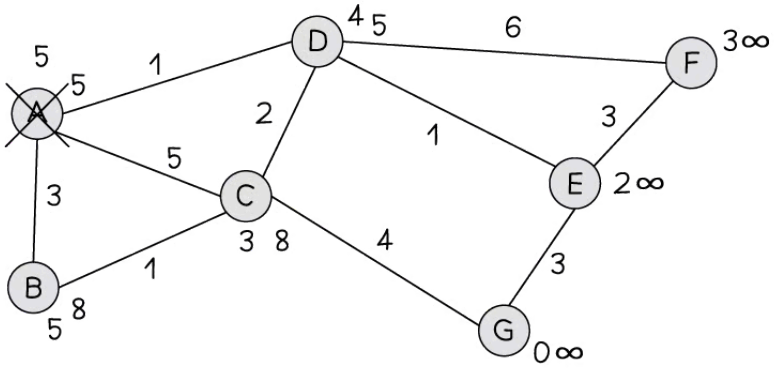
Your notes



VISITED

Node	g(x)	h(x)	f(x)	Previuos
A	0	5	5	–
B	∞	5	∞	–
C	∞	3	∞	–
D	∞	4	∞	–
E	∞	2	∞	–
F	∞	3	∞	–
G	∞	0	∞	–

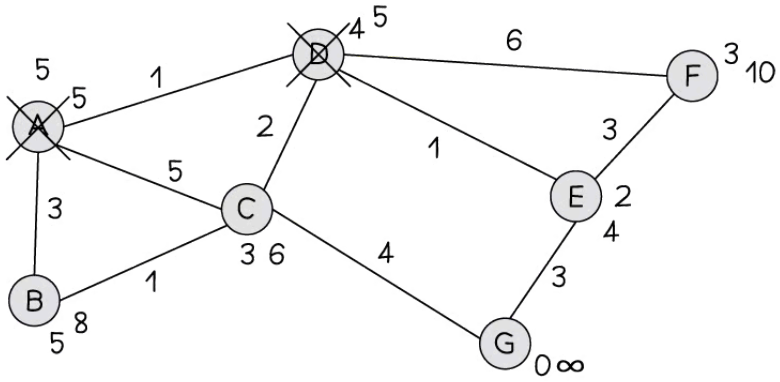
Your notes



VISITED
A

Node	g(x)	h(x)	f(x)	Previous
~~A~~	~~0~~	~~5~~	~~5~~	~~–~~
B	3	5	8	–
C	5	3	8	–
D	1	4	5	–
E	∞	2	∞	–
F	∞	3	∞	–
G	∞	0	∞	–

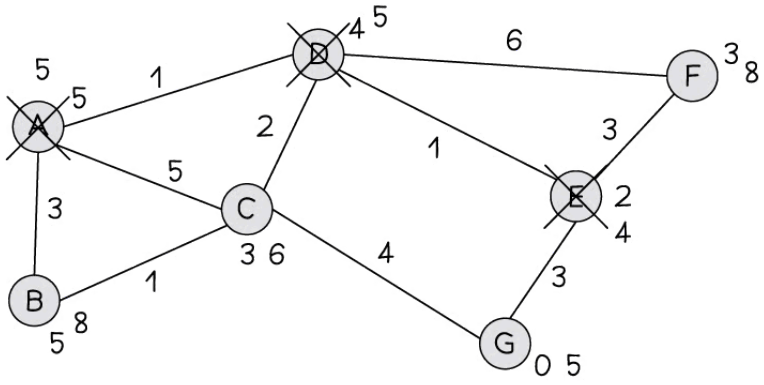
Your notes



VISITED
AD

Node	g(x)	h(x)	f(x)	Previous
~~A~~	~~0~~	~~5~~	~~5~~	~~–~~
B	3	5	8	–
C	3	3	6	–
~~D~~	~~1~~	~~4~~	~~5~~	~~A~~
E	2	2	4	–
F	7	3	10	–
G	∞	0	∞	–

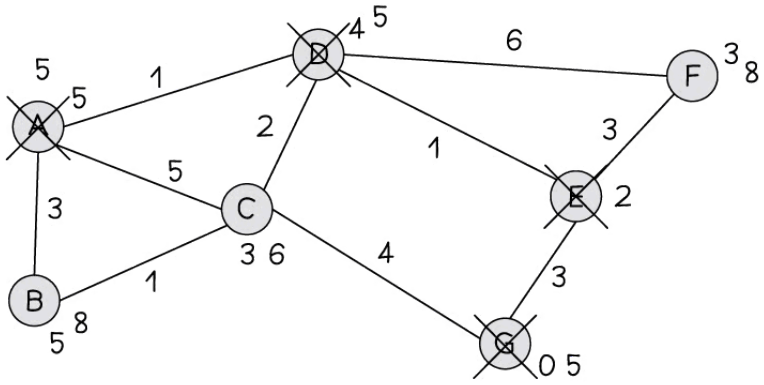
Your notes



VISITED
ADE

Node	g(x)	h(x)	f(x)	Previous
~~A~~	~~0~~	~~5~~	~~5~~	~~–~~
B	3	5	8	–
C	3	3	6	–
~~D~~	~~1~~	~~4~~	~~5~~	~~A~~
~~E~~	~~2~~	~~2~~	~~4~~	~~D~~
F	5	3	8	–
G	5	0	5	–

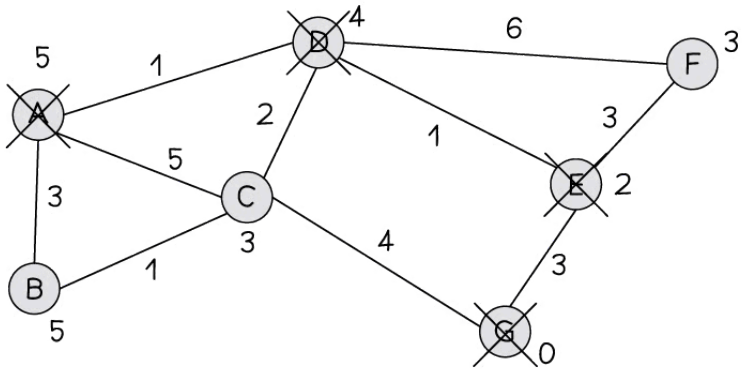
Your notes



VISITED
ADEG

Node	g(x)	h(x)	f(x)	Previous
~~A~~	~~0~~	~~5~~	~~5~~	~~–~~
B	3	5	8	–
C	3	3	6	–
~~D~~	~~1~~	~~4~~	~~5~~	~~A~~
~~E~~	~~2~~	~~2~~	~~4~~	~~D~~
F	5	3	8	–
~~G~~	~~5~~	~~0~~	~~5~~	~~E~~

Your notes



VISITED
ADEG

Node	g(x)	h(x)	f(x)	Previous
A	0	5	5	–
B	3	5	8	–
C	3	3	6	–
D	1	4	5	A
E	2	2	4	D
F	5	3	8	–
G	5	0	5	E

Figure 2: Performing the A* Search

- The algorithm for A* search is shown below, both in structured english and a pseudocode format

A* Search (Structured English)

```
assign g(x) to 0 and f(x) to h(x) for the initial node and g(x) and f(x) to infinity for every other node
add all nodes to a priority queue, sorted by f(x)
while the queue is not empty
    remove node x from the front of the queue and add to a visited list
    If node x is the goal node,
end the while loop
    else
    for each unvisited neighbour y of the current node x
        g(x) = g(x)AtX + distanceFromXtoY
        if g(x) < g(x)AtY then
            g(x)AtY = g(x)
            f(x)AtY = g(x)AtY + h(x)AtY
            reorder the priority queue by changing Y's position based on the new f(x)
        endif
    next X
endwhile
```

A* Search (Pseudocode)

```
function AStarSearch(graph, start, goal)
    //set all distances to infinity and first distance to 0
for node in graph
        g[node] = infinity
        f[node ] infinity
    next node
    g[start] = 0
    f[start] = h[start]

    //A* ends when the goal node has been reached
    while goal in graph
        //find the shortest distance f(x) from all the nodes in order to select the next node to visit and expand
        min = null
        for node in graph
            If min == null then
                min = node
            elseif f[node] < f[min] then
                min = node
            endif
        next node

        //for each neighbour, update the cost if the current cost and distance are less than the previously stored distance
        for neighbour in graph
            if cost + distance[min] < distance[neighbour] then
                distance[neighbour] = cost + distance[min]
                F[neighbour] = cost + g[min] + h[neighbour]
                previousNode[neighbour] = min
            endif
        next neighbour

        graph.remove(min)
    endwhile
```


Your notes

```
    //backtrack through the previous nodes and build up the final path until the start is reached
    node = goal
while node != start
    path.append(node)
    node = previousNode[node]
endwhile

    return path

endfunction
```

- A* search behaves almost exactly as Dijsktra's except that when the goal node is reached, the algorithm stops and the old cost function is replaced with a new heuristic based function

Examiner Tips and Tricks

You do not need to know the time or space complexity for the A* search algorithm

Tracing the A* Search Algorithm

- Tracing the A* search algorithm can be done in a manner similar to the Figure 2 above
- Remember to initialise the starting node g(x) to 0 and f(x) to h(x)
- Update each nodes neighbours by adding the previous path and the distance from the current node to the neighbour to the heuristic value for the neighbour node
- Keep track of each nodes previous node that offered the shortest route

Worked Example

Some of the characters in a game will move and interact independently. Taylor is going to use graphs to plan the movements that each character can take within the game.

DancerGold is one character. The graph shown in Fig. 1 shows the possible movements that DancerGold can make.

Your notes

Your notes

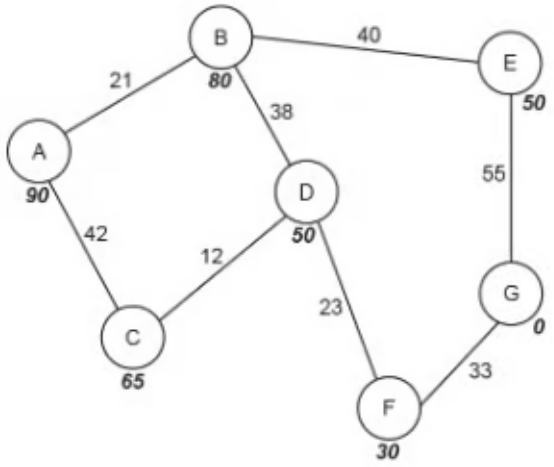


Fig. 1

DancerGold's starting state is represented by node A. DancerGold can take any of the paths to reach the end state represented by node G.

The number on each path represents the number of seconds each movement takes.

The number in bold below each node is the heuristic value from A.

i) Define the term heuristic in relation to the A* algorithm.

2 marks

ii) Perform an A* algorithm on the graph shown in Fig. 1 to find the shortest path from the starting node to the end node. Show your working, the nodes visited and the distance. You may choose to use the table below to give your answer.

Node	Distance travelled	Heuristic	Distance travelled + Heuristic	Previous node

Final path:

Distance:

8 marks

Answer:

i) An heuristic is a rule of thumb or guess **[1]** that estimates the distance or cost from one node to the destination node **[1]**.

ii)

Node	Distance travelled	Heuristic	Distance travelled + Heuristic	Previous node	MARKs
A (✓)	0	90	90		**[1]**
B (✓)	∞ 21	80	101	A	**[1]**
C (✓)	∞ 42	65	107	A	**[1]**
D (✓)	∞ 42+12=54	50	104	C	**[1]**
E	∞ 21+40=61	50	111	B	**[1]**
F (✓)	∞ 42+12+23=77	30	107	D	**[1]**
G	∞ 42+12+23+33=110	0	110	F	**[1]**

Final path = A,C,D,F,G and Distance = 110 **[1]**

Remember that the distance travelled is based on the edge weights between two nodes. This is added to the heuristic to find the total distance travelled to a node.

Your notes