

ANALYSIS OF ALGORITHMS REVISION

Exam Paper Analysis	2
Basics	3
Space Complexity	3
Considerations.....	3
Maths.....	3
Logarithms	3
Summations.....	3
Growth	3
Complexity.....	3
Exam Questions	4
Searching Algorithms	4
Sequential Search.....	4
Binary Search.....	5
Sorting Algorithms.....	6
Insertion Sort.....	6
Bubble Sort	7
Selection Sort	8
Shell Sort.....	9
Heap Sort	9
Merge Sort	10
Quick Sort	11
Graphs	12
Adjacency Matrix.....	12
Adjacency List	12
Bi-Connectors & Articulation Points.....	13
Weighted Graphs	14
Greedy Algorithms.....	15
Dijkstra's Algorithm	15
Prim's Algorithm	15
Convex Hulls	16
Basics	16
Graham's Scan Algorithm	16

EXAM PAPER ANALYSIS

Sorting

- Divide & Conquer strategies
 - Insert
 - Bubblesort
 - Shellsort
 - Selectionsort
 - Quicksort
 - Mergesort
 - Heapsort

Will be given an array of numbers and have to sort it using one of the methods opposite

Searching

- Sequential search
- Binary search

Graphs

- Adjacency Matrix / lists
- DFT depth first traversal
- Articulation points
- Bi-connections

Greedy Algorithms

- Dijkstra's algorithm (Pascal? distance)
- Prim's algorithm (spanning tree)
 - Cost matrix

Convex hulls / polygons

- Grahams Algorithm

Divide & Conquer

This algorithm will take a task; divide it into sub-sections so that different functions can process the data, and then return with a result. This often encompasses recursion and it is not usually obvious how many times it will run.

BASICS

Space Complexity

With old computers it was essential that algorithms did not use too much space whilst executing. Modern day example would be software designed for PDA's.

Considerations

- Best case scenario where the algorithm executes at its fastest
- Worst case scenario where the algorithm executes at its slowest
- Average case scenario is to be ignored

Maths

Logarithms

$$\text{Log}_B 1 = 0$$

$$\text{Log}_B B = 1$$

$$\text{Log}_B (X * Y) = \text{Log}_B X + \text{Log}_B Y$$

$$\text{Log}_B X^Y = Y * \text{Log}_B X$$

$$\text{Log}_A X = \frac{\text{Log}_B X}{\text{Log}_B A}$$

Summations

$$\sum_{i=1}^N i$$

$i = 1$ is the starting value.
 N is the end value for i .

Growth

- **Big Omega** $\Omega(f)$ is the lower bound of a function. It represents the class of functions that grow at least as the function (f) . Hence $\Omega(n^2)$ include all functions such as n^3 and 2^n .
- **Big Oh** $O(f)$ represents the upper bound of the function (f) . This is the more looked at of the classifications of growth. It represents the class of functions that grow no faster than (f) .
- **Big Theta** $\Theta(f)$ represents functions that grow at the same rate as (f) .

Complexity

Linear algorithms would a complexity of $O(N)$

Quadratic algorithms would a complexity of $O(N^2)$

Logarithmic algorithms would a complexity of $O(\lg N)$

EXAM QUESTIONS

Searching Algorithms

Sequential Search

This type of search algorithm is very simple. It will basically look at each element in a list until a match is found. It is obvious that the further down the list you go, the longer it will take to find the match you want. The following code represents how the search would run, it will return 0 if the item is not found.

```

// list           the elements to be searched
// target        the value being searched for
// N             the number of elements in the list

SequentialSearch(list, target, N)
    for i = 1 to N do
        if (target = list[i])
            return i
        end if
    end for
    return 0
end sub
    
```

Worst Case Analysis

There are two worst-case scenarios:

- This is simply if the item was in the lists last position N. It would take N searches to find the item you were looking for.
- If the item is not found in the list then it will run N times and return 0.

Best Case Analysis

The best-case scenario is if the item you were search for was at position 1 in the list. You have to assume the item will be in the list.

Example Run

list = 7 3 9 2 1 6 4 8 0
 Target = 4

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

Binary Search

Using a sorted list, you take the value you are searching for and compare it to the middle of the sorted list (binary chop). If the value is greater, then it is in the right hand side of the list, and if it is less than the value in the middle, then it is in the left half. By using “chopping” the list into halves you will eventually find the value you are looking for.

Iterative Code

```

// list      the elements to be searched
// target    the value being searched for
// N         the number of elements in the list

BinarySearch (list, target, N)
  start = 1
  end = N
  while start <= end do
    middle = (start+end)/2
    select (Compare(list[middle], target)) from
      case -1: start = middle +1
      case 0: return = middle
      case 1: end = middle -1
    end select
  end while
  return 0
end sub

```

Worst Case Analysis

The larger the list then the more times the system has to split the list into halves. No matter what size the list is when you start, if you keep dividing by two (2) (and through away the fractional part) – you will eventually end up with a list of one element.

So the number of comparisons k is always going to be the worst case scenario.

Another worst case is if the item is not in the list. The algorithm then does N +1 comparisons.

Best Case Analysis

Have a list of size 1, that is one element – which is found on the 1st pass.

Example Run

```

list      = A B C H P S W
Target    = P

```

A	B	C	H	P	S	W
			^			
A	B	C	H	P	S	W
					^	
A	B	C	H	P	S	W
				^		

```

return 6 //index of element

```

Sorting Algorithms

Insertion Sort

The main principle of an insertion sort is that you have an already sorted list. You then want to add an item to the list but you want to **insert** the item at the correct position so as to not waste time re-sorting the list again.

```

// list    the elements to be put into order
// N       the number of elements in list

InertionSort (list, N)
  for i = 2 to N do
    newElement = list[i]
    location = i-1
    while (location > 1) and (list[location] > newElement) do
      // move any larger elements out of the way
      list[location +1] = list[location]
      location = location-1
    end while
    list[location+1] = newElement
  end for
end sub
    
```

Considerations

- First element is already sorted and of size 1.
- The position `location +1` is available for the “new” element.

Worst Case Scenario

From looking at the code you will notice that the algorithm does the most work during the “*while*” loop. The worst case would be one that creates more iterations through this loop – hence if you keep inserting elements that are smaller than the first element (the while loop moves the other elements down-a-notch).

Best Case Scenario

The value being added to the list is greater than the last item in the list. The algorithm then only needs to add the item to position `location +1`.

Bold = fixed sorted value

13	17	16	15	19	12	-> do nothing
	^					
13	17	16	15	19	12	
		^				
13	16	17	15	19	12	
			^			
13	15	16	17	19	12	-> do nothing
			^			
13	15	16	17	19	12	
				^		
12	13	15	16	17	19	-> complete

Bubble Sort

This type of algorithm simply takes adjacent (next to each other) pairs of values in a list, compares them and then swaps if they are out of order. Smallest value numbers to the left of the list and the largest to the right.

```
// list    the elements to be put into order
// N      the number of elements in list

BubbleSort(list, N)

    numberOfPairs = N
    swappedElements = false

    while swappedElements do
        for i = 1 to numberOfPairs do
            if list[i] > list[i+1] then
                Swap(list[i], list[i+1])
                swappedElements = true
            end if
        end for
    end while
end sub
```

Worst Case Scenario

If the list is in reverse order – this means that the while loop is executed $O(N^2)$ times.

Best Case Scenario

There are no swaps on the first pass ($N - 1$ comparisons).

Selection Sort

1. Initially you need to determine the smallest item in the list.
2. perform a swap so that this item is now in the correct place
3. reduce the search set (move inwards) and repeat

```

// list    the elements to be put into order
// N       the number of elements in list

SelectionSort(list, N)
  for i = 1 to N-1
    minj = i
    for j = i+1 to N do
      if (list[j] < list[minj])
        minj = j
      end if
    end for
    swap (list, i, minj)
  end for
end sub

```

Worst Case Scenario

The selection sort algorithm uses nested loops $O(N^2)$.

Worst initial order would be one of :

- all same values
- 8 1 2 3 4 5 6 7 – shifting the 8

Example Run

list = 13 17 16 15 19 12

Italic Bold = item being looked at unsorted min value
Bold = fixed sorted value

13	17	16	15	19	<i>12</i>	
12	17	16	15	19	<i>13</i>	1 st pass
12	13	16	<i>15</i>	19	17	2 nd pass
12	13	15	<i>16</i>	19	17	3 rd pass
12	13	15	16	19	<i>17</i>	4 th pass
12	13	15	16	17	19	5 th pass

Shell Sort

This algorithm takes the complete list of elements and makes several passes of the list putting values into sublists.

1. on the first pass it will match pairs and swap them around depending in its size
2. the second pass will match four elements
3. the third will match eight and so on....

The plan is to increase the number of elements per sublist so eventually there is only one list.

Heap Sort

Heapsort is based on a special type of binary tree where for every subtree the value at the root is larger than all the values in the two children.

You initially create the heap and then run a `FixHeap` algorithm to arrange the values in the correct sorted manor.

Merge Sort

Principles

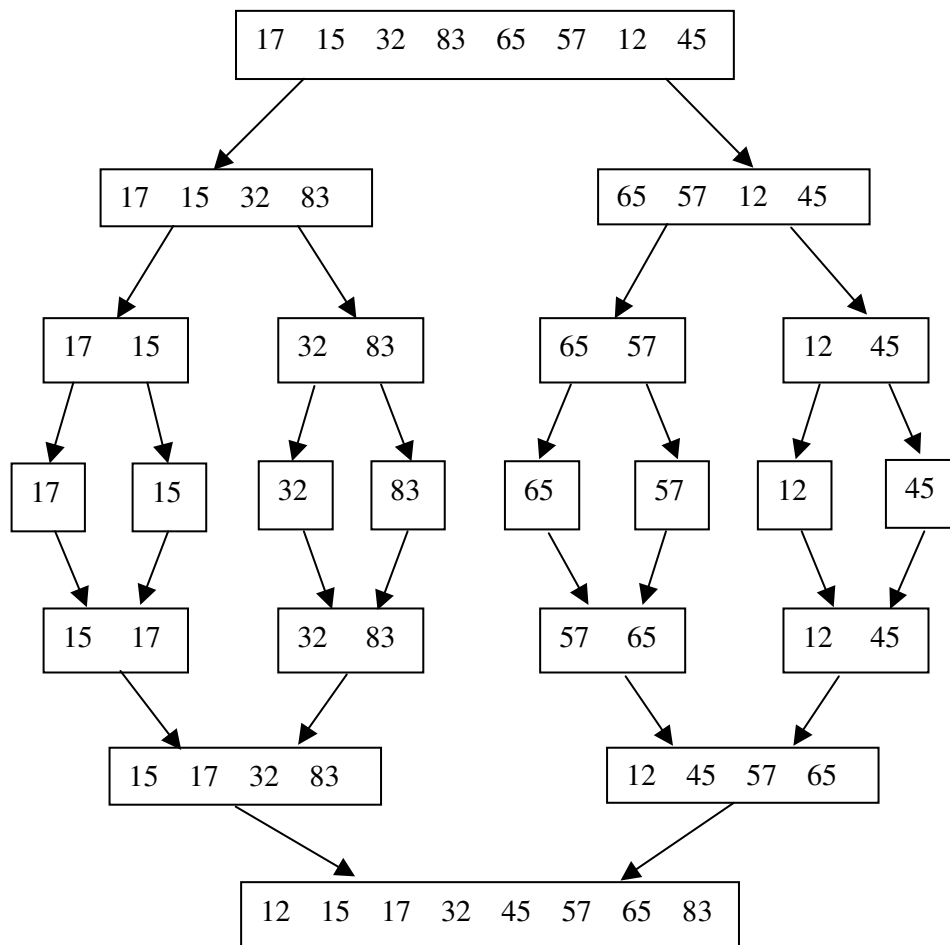
- divide the array into two sub-arrays with equal number of items
- sort separately each sub-array (using recursion if necessary)
- combine arrays

The system would use a function called `MergeLists(list, first, middle, last)` – this simply combines the split arrays into one.

```

// list    the elements to be sorted
// first   index of first of elements in sublist to be sorted
// last    index of last of elements in sublist to be sorted

MergeSort( list, first, last)
  if first < last then
    middle = ( first + last ) / 2
    MergeSort( list, first, middle )
    MergeSort( list, middle + 1, last )
    MergeLists ( list, first, middle, middle+1, last )
  end if
end sub
    
```



Best & Worst Case Scenario

The comparison code within the recursive call of the subroutine are the major time considerations within the algorithm.

Therefore, the worst-case mergesort complexity is $O(N \lg N)$ and so is the best case.

Quick Sort

This is a recursive-based divide-and-conquer sorting algorithm. It has the time complexity of $O(N \lg N)$ which means it is very quick.

The algorithm picks a “*pivot*” element and then determines the elements smaller or larger than it by calling itself. When the recursion reaches a single element it will terminate and the elements will now be in the correct order.

```
// list      the elements to be put in order
// first     index of first element in the part of list to be sorted
// last      index of last element in the part of list to be sorted

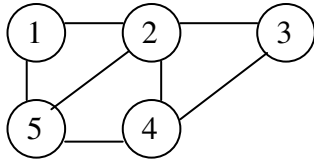
quickSort ( list, first, last )
  if first < last then
    pivot = pivotList ( list, first, last)
    quickSort( list, first, pivot-1)
    quickSort( list, pivot+1, last)
  end if
end sub
```

Worst Case Scenario

- relies on evenly balanced numbers so that it will correctly pick a pivot element.
- The worst case scenario occurs when the pivot value is one that is greater (or lower) than all of the other numbers. This would make one side of the partition execute $N-1$ times
- An already sorted list would cause this.

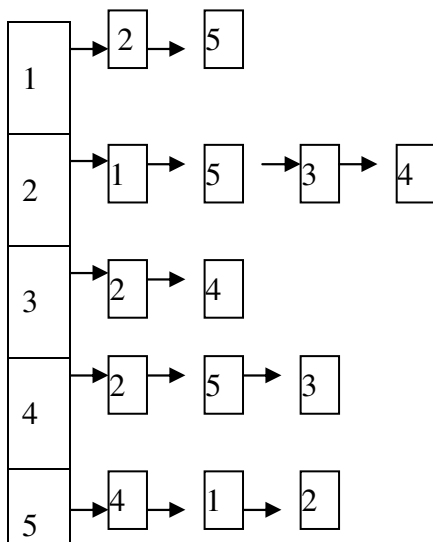
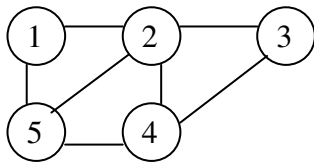
Graphs

Adjacency Matrix

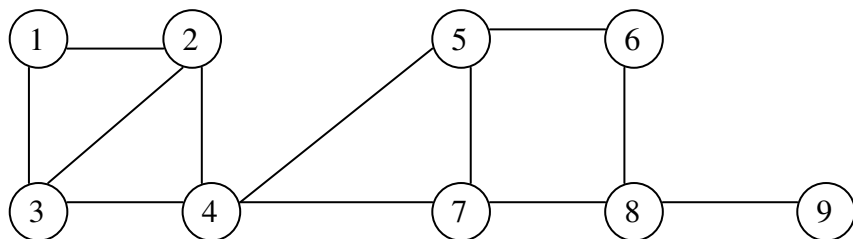


$$M = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 & 5 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix} & \begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \end{bmatrix} \end{matrix}$$

Adjacency List



Bi-Connectors & Articulation Points

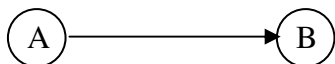


A **path** is a sequence of edges connecting two vertices.

A graph is **connected** if there is a path between any two vertices.

An **articulation** point is where two graphs meet (4&8) in this case and if deleted (along with its incident edges) would separate a graph into two or more disconnected components.

A directed has arrows indicating the adjacency:

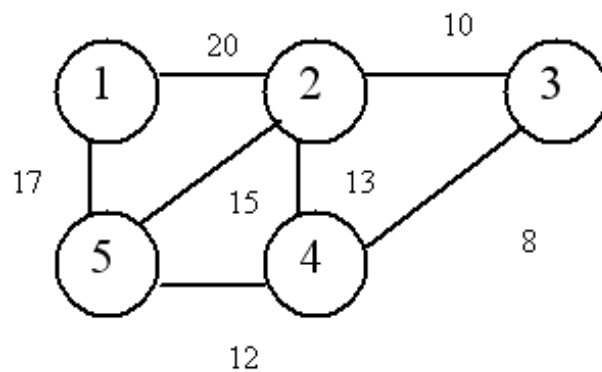


AB are adjacent; BA are not.

Weighted Graphs

A weighted graph (or weighted digraph) is one where each edge has a value, called the *weight*, associated with it. In working with graphs or digraphs, we consider the weight to be the *cost* for traversing the edge (or arc). A path through a weighted graph has a cost that is the sum of weights of each edge in the path. In a weighted graph, the shortest path between two vertices is the path with the smallest cost.

Weighted Graph



Cost Matrix C for this weighted graph:

$$C = \begin{bmatrix} \infty & 20 & \infty & \infty & 17 \\ 20 & \infty & 10 & 13 & 15 \\ \infty & 10 & \infty & 8 & \infty \\ \infty & 13 & 8 & \infty & 12 \\ 17 & 15 & \infty & 12 & \infty \end{bmatrix}$$

Note: it is usual to make cost infinity where there is no edge

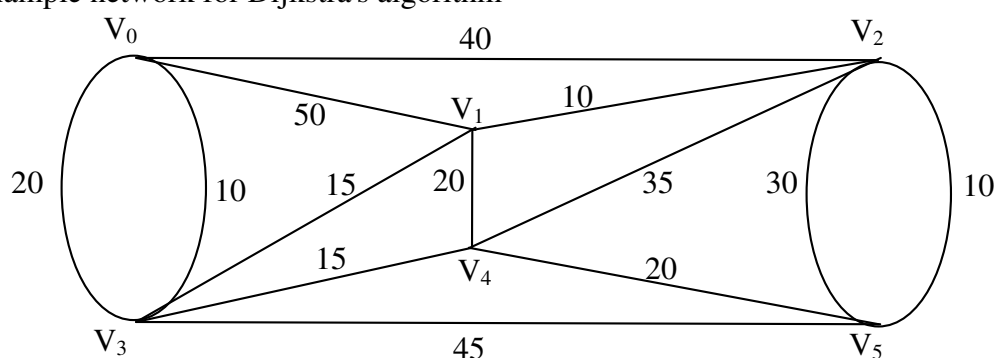
Greedy Algorithms

Dijkstra's Algorithm

The purpose of **Dijkstra's** algorithm is to find the lengths of the shortest paths from a source node to each of the other nodes, in an increasing order of the lengths of these paths.

Note that whereas Dijkstra's algorithm finds shortest paths from a single source to the rest of the nodes in a graph, **Prim's** algorithm finds successive shortest distances from any point on the minimum spanning tree as it is being built, to the rest of the nodes in the graph. The MST does not necessarily contain the edges of the shortest path between any two vertices of the graph.

Example network for Dijkstra's algorithm



The adjacency matrix for this example is below

	0	1	2	3	4	5
0	∞	50	40	10	∞	∞
1	∞	∞	10	15	∞	∞
2	∞	∞	∞	∞	∞	10
3	20	∞	∞	∞	15	45
4	∞	20	35	∞	∞	∞
5	∞	∞	30	∞	20	∞

Prim's Algorithm

// Prim's algorithm outline

```

let T be a single vertex x
while (T has fewer than N vertices)
    find the smallest edge connecting T to G-T
    add it to T
end while
    
```

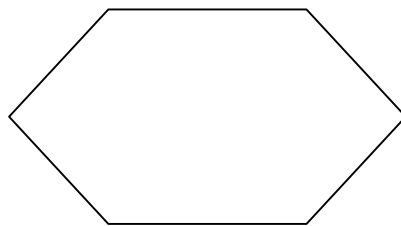
Convex Hulls

Basics

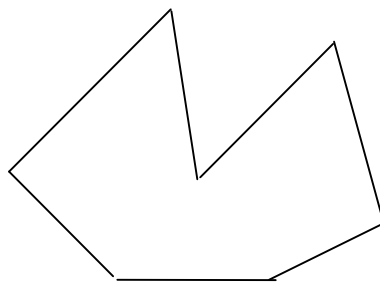
A convex hull is an object that has all of its edges sticking out. Otherwise it is known as a non-convex hull.

Like when wrapping a present, the paper goes across each point and does not stick-in. This would therefore be classified as a convex hull.

With a polygon you determine whether a point is within a polygon by drawing a line to anywhere that outside the object. An odd number of intersections would mean that you are inside the object:



is convex : all points "stick out" of the polygon



is non - convex

In general, the number of points on a convex hull of an N point polygon need to lie between 3 and N (three being a triangle).

Graham's Scan Algorithm

- denote the point with smallest y
- coordinate by $p[0]$. This is the *anchor* point.
- sort the points anticlockwise by radial angle from the anchor point so that $p[0]$, $p[1]$, ..., $p[n-1]$, $p[0]$ form a closed path.
- now try to place each point on the convex hull
- previous point ok if anticlockwise turn to new point
- **eliminate** previously placed **indented point(s)** when **clockwise** turn to new point (indented so cannot lie on hull)