DATA STRUCTURES & ALGORITHMS

COMP H3025

Lecture 9: Algorithm Complexity

Lecturer: Stephen Sheridan

ANALYSIS OF ALGORITHMS

- There are many algorithms that can solve a particular problem. They will have different characteristics that will determine how efficiently each will operate.

- When we analyse an algorithm we have to:

    - Show that it actually solves the problem.

    - That it solves the problem in an efficient manner.

ANALYSIS OF ALGORITHMS

- Algorithm analysis means determining the amount of time that an algorithm may take to execute. This is not the clock time in the sense of the number of seconds or minutes that a program takes to execute.

- It is instead an approximation of the number of operations that an algorithm performs.

- The number of operations is **related to the execution time**, so we talk about the **time** that an algorithm takes to describe an algorithm's complexity.

- The actual number of seconds it takes an algorithm to execute on a computer is not useful in our analysis because we are concerned with the relative efficiency of algorithms that solve a particular problem.

ANALYSIS OF ALGORITHMS

- The actual execution time is not a good measure of algorithm efficiency.

An algorithm does not get **better** because we move it to a **faster** computer!

An algorithm does not get **worse** because we move it to a **slower** computer!

ANALYSIS OF ALGORITHMS

- The actual number of operations for some specific size of input data set is not very interesting and will not tell us very much.

- We need to determine in our algorithm analysis an equation that relates the number of operations that a particular algorithm does to the size of the input,

- That is, where the number of operations is a function of the size of the input data set.

- We can then compare two algorithms that solve the same problem by comparing the rate at which their equations grow.

ANALYSIS OF ALGORITHMS

The growth rate is critical because there are instances where algorithm A may take fewer operations than algorithm B when the input size is small, but many more when the input size gets large.

ANALYSIS OF ALGORITHMS

- In an analysis, we come up with an estimate of how long the algorithm will take to solve a problem that has a set of N input values.

- We might, for example, determine how many comparisons a sorting algorithm does to put a list of N values into ascending order.

- We might determine how many operations it takes to multiply two matrices of size N * N.

- Studying the analysis of algorithms gives us the tools to choose between algorithms. Consider the following two algorithms.

ANALYSIS OF ALGORITHMS

Algorithm 1

```
largest = a
if b > largest
        largest = b
endif

if c > largest
        largest = c
endif

if d > largest
        largest = d
endif
return largest
```

Algorithm 2

```
if a > b
        if a > c
                if a > d
                        return a
                else
                        return d
                endif
        else
                if c > d
                        return c
                else
                        return d
                endif
        endif
else
        if b > c
                if b > d
                        return b
                else
                        return d
                endif
        else
                if c > d
                        return c
                else
                        return d
                endif
        endif
endif
```

ANALYSIS OF ALGORITHMS- TIME & SPACE

- In terms of time, these two algorithms are the same.
- In terms of space, algorithm 1 needs more because of the need to hold the temporary variable largest.
- The space requirement is not significant in this example, but it may be for other examples with different data types, i.e. large objects.
- In our analysis of algorithms we will generally be concerned with time complexity.

ANALYSIS OF ALGORITHMS- TIME & SPACE

- Time Complexity - Traversal of a linked list

- We can display the contents of a linked list by using the following Java statements.

```
Node curr = head;                          //1 assignment
while (curr != null)                       //n + 1 comparisons
{
   system.out.println(curr.getItem());  //n writes
   curr = curr.getNext();                  //n assignments
}  //end while
```

ANALYSIS OF ALGORITHMS- TIME & SPACE

- Assuming a linked list of **n nodes**, these statements require **n+ 1** assignments, **n+ 1** comparisons and **n** write operations.

- If each assignment comparison, and write operation requires, respectively, a, c and w time units, the statements require:

- **(n + 1) * (a + c) + n * w time units**.

- The time required to write n nodes is therefore proportional to n.

- This conclusion makes sense intuitively

  It takes longer to display, or traverse, a linked list of 100 items than it does a linked list of 10 items.

ANALYSIS OF ALGORITHMS- TIME & SPACE

- Assuming a linked list of **n nodes**, these statements require **n+ 1** assignments, **n+ 1** comparisons and **n** write operations.

- If each assignment comparison, and write operation requires, respectively, a, c and w time units, the statements require:

- **(n + 1) * (a + c) + n * w time units**.

- The time required to write n nodes is therefore proportional to n.

- This conclusion makes sense intuitively

  It takes longer to display, or traverse, a linked list of 100 items than it does a linked list of 10 items.

ANALYSIS OF ALGORITHMS- TIME & SPACE

- Let us look at another example of an algorithm that counts the number of different characters in a file.

- The algorithm might look like the following:

```
for all 256 characters do
   assign zero to each character counter
end for

while there are more characters in the file do
   get the next character
   increment the counter for this character by one
end while
```

ANALYSIS OF ALGORITHMS- TIME & SPACE

- We can see that there are **256** passes for the initialisation loop.
- IF there are **N** characters in the input file, THEN there are **N** passes for the second loop.
- So, …. what do we count?

```
for (expression 1;expression 2;expression 3)
{
//Java code statements
//Java code statements
//Java code statements
}
```

ANALYSIS OF ALGORITHMS- TIME & SPACE

- This means that the initialisation loop does
  - 257 assignments: 1 for the loop variable and 256 for the counters
  - 256 increments of the loop variable
  - 257 checks that this variable is within the loop bounds (the "extra" one is when the loop stops)

- For the second loop we will need to
  - do a check on the condition N + 1 times (the + 1 is for the last check when the files is empty)
  - increment N counters

ANALYSIS OF ALGORITHMS- TIME & SPACE

- The total number of operations for this example is:
  - Increments **N + 256**
  - Assignments **257**
  - Conditional checks **N + 257**

- If we have a file with **500** characters, the algorithm will do a total of **1770** operations, of which **770** are associated with initialisation.

Where

N * 2 = 500 * 2 = 1000

256 + 257 + 257 = 770

Total = 1770

ANALYSIS OF ALGORITHMS- TIME & SPACE

- What happens when N gets larger?
- If we had a file with 50,000 characters the algorithm will do a total of 100,770 operations
- The same 770 operations are associated with initialisation.
- The number of initialisations operations has not changed but their overhead takes a much smaller percentage of the total operations as N increases.
- **Therefore, the cost of initialisation becomes meaningless as N gets larger.**
- In this example the equation is:
    - **2N + initialisation operations**
- We can now see how this algorithm grows and we know the algorithm growth rate.

ALGORITHM GROWTH RATES

- We derive an algorithm's time requirements as a function of the problem size.

- How do we state this?

  - Algorithm **A** requires $n^2/5$ time units to solve a problem of size n.

  - Algorithm **B** requires $5*n$ time units to solve a problem of size n.

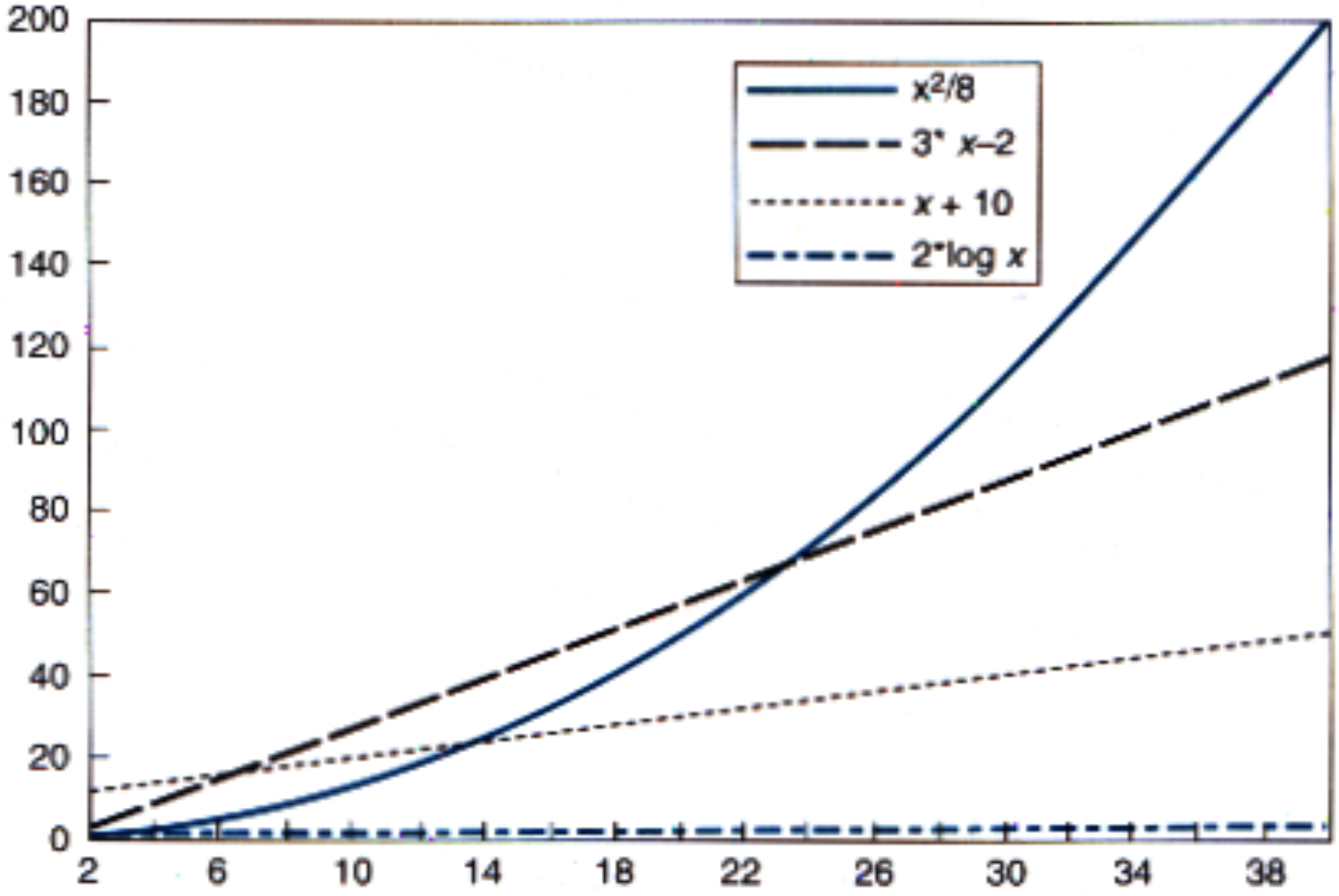
ALGORITHM GROWTH RATES

- The function based on **x2** increases slowly at first, but as the problem size gets large, it begins to grow at a rapid rate.

- The functions based on **x** grow at a steady rate as the problem size gets large.

- The function based on **log x** seems to not grow at all because it is growing at a very slow rate

ALGORITHM GROWTH RATES



Graph of four functions

ORDER OF MAGNITUDE ANALYSIS AND GROWTH RATES

- If an algorithm's growth rate function is a combination of fast-growing and slow-growing terms we may ignore all but the fastest growing term when stating its complexity because this is the dominant term in the equation.

- When we do this we are left with the order of the growth rate function, ***f(n)***, which is denoted as ***O(f(n))***.

- Because the notation used the capital letter ***O*** to denote order, it is called the Big ***O*** or Big ***Oh*** notation.

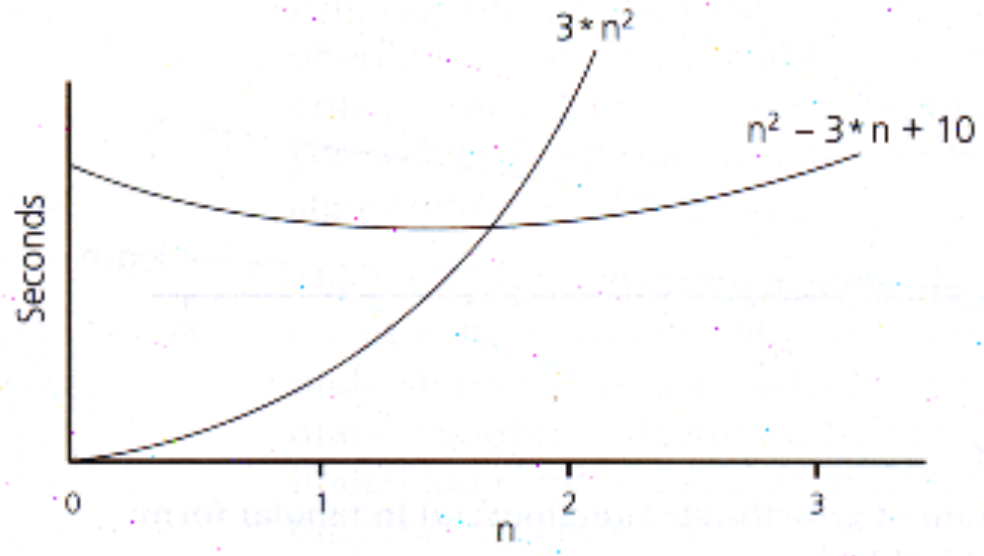
- If a problem of size n requires time that is directly proportional to n, the problem is ***O(n)***, that is, of order n.

- If a problem of size n requires time that is directly proportional to ***n2***, the problem is ***O(n2)***, that is, of order ***n2***.

ORDER OF MAGNITUDE ANALYSIS AND GROWTH RATES

- If an algorithm requires $n^2 - 3 * n + 10$ time units, then the algorithm is of order n^2 or $O(n^2)$

- If an algorithm requires $(n+1) * (a + c) + (n * w)$ time units, say for displaying the first **n** elements of a linked list, then the task is $O(n)$.

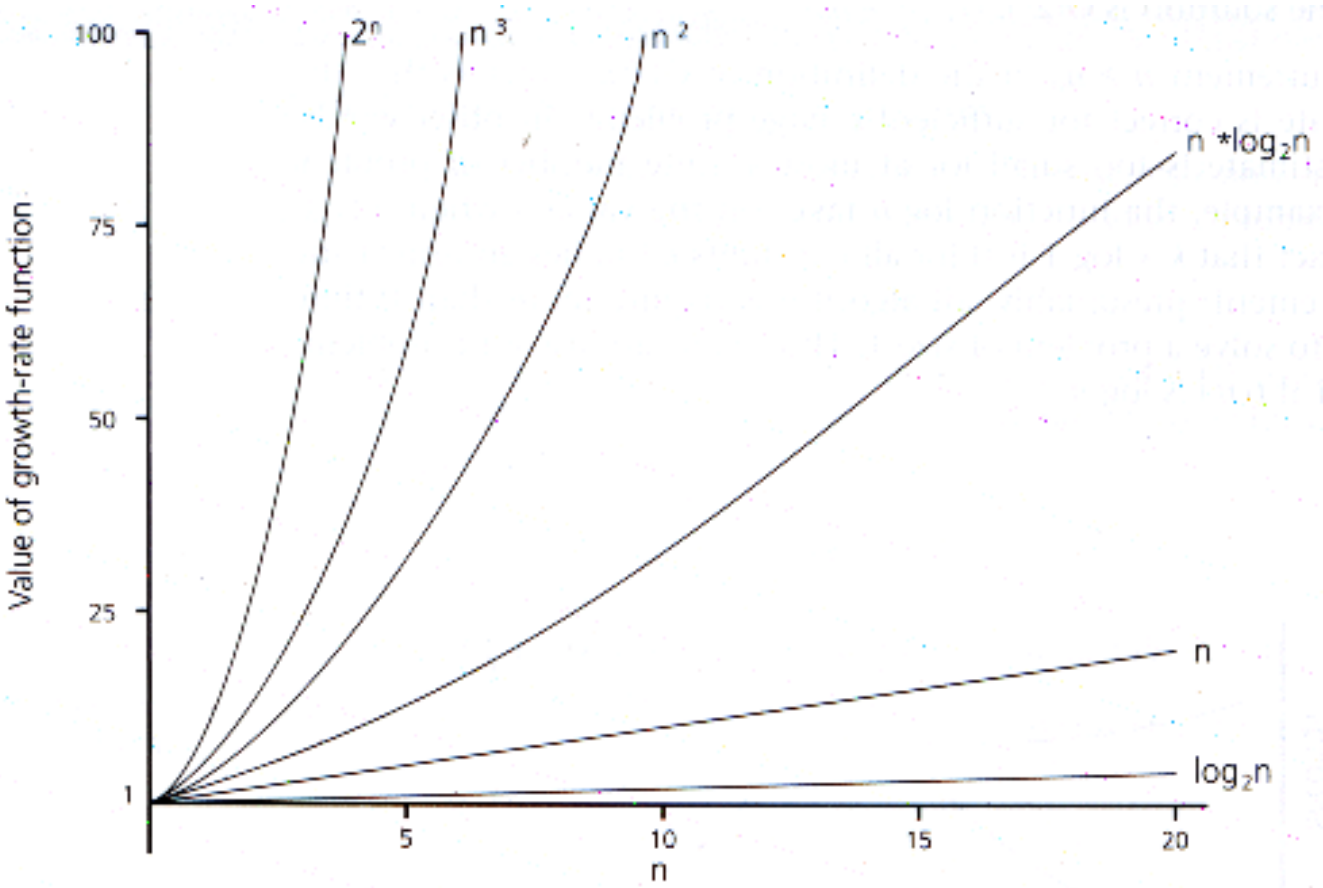


When $n \geq 2$, $3 * n^2$ exceeds $n^2 - 3 * n + 10$

ORDER OF MAGNITUDE ANALYSIS AND GROWTH RATES

- The growth rate functions are listed in order of growth

$$O(1) < O(\log_2 n) < O(n) < O(n * \log 2\ n) < O(n^2) < O(n^3) < O(2^n)$$



ORDER OF MAGNITUDE ANALYSIS AND GROWTH RATES

- What are the growth rates of the following operations:

- Calculate some number N * 100? ***O(1)***

- Search for a value in an unsorted integer array? ***O(n)***

- Search for a value in a sorted array or binary search tree? ***O(log(n))***

- Sorting an array using the **Selection** sort algorithm? ***$O(n^2)$***

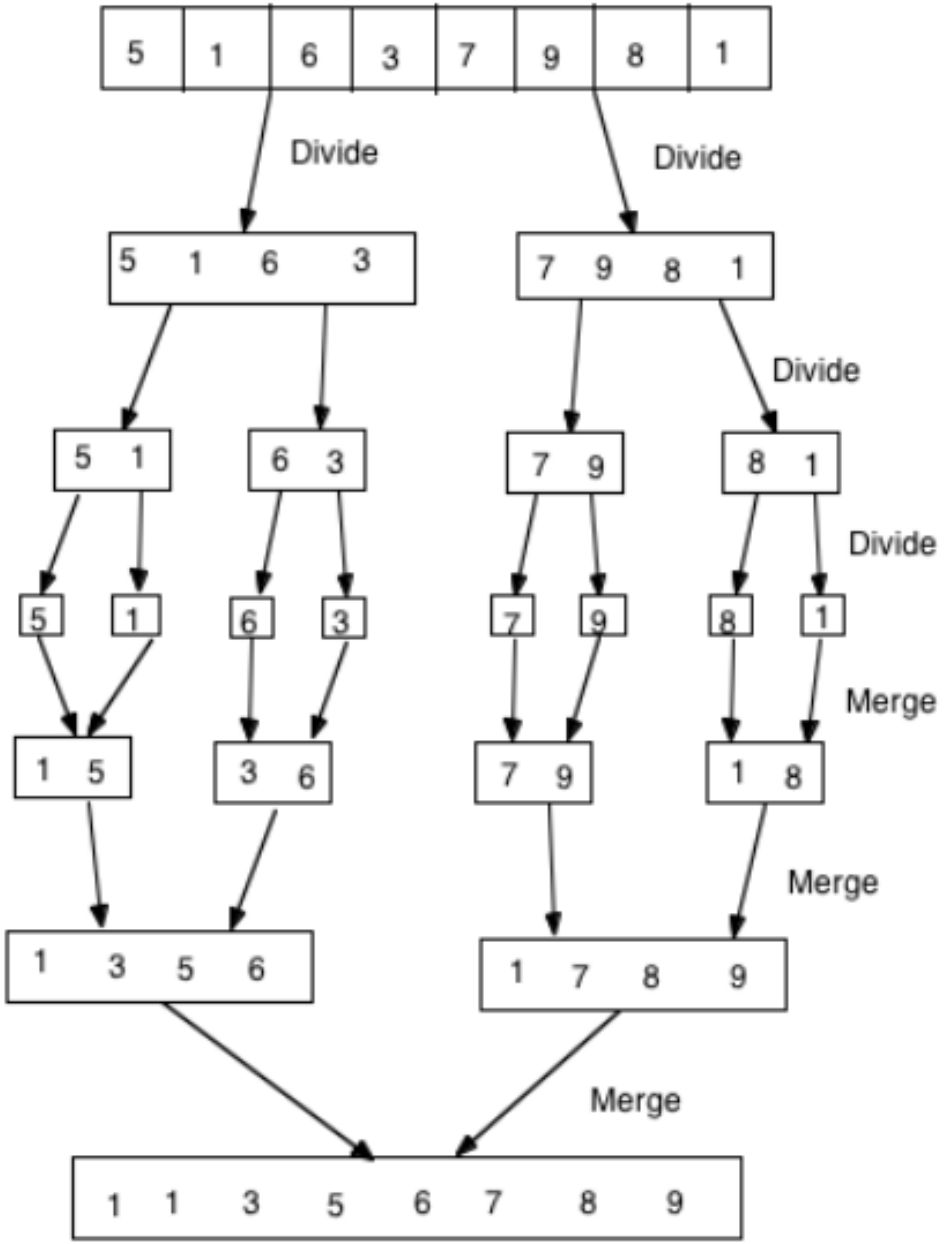
MERGESORT

- The merge sort closely follows the divide and conquer paradigm by recursively dividing a given sequence of values until the size of the sequence to be sorted is 1. It then merges the sequences.

- The division involves no comparisons. The key operation then becomes the merging of sorted subsequences to form the sorted sequence.

- The following tree like diagram illustrates the process of dividing and merging a given data set.

MERGESORT


5 1 6 3 7 9 8 1
Divide
Divide
5 1 6 3
7 9 8 1
Divide
5 1
6 3
7 9
8 1
Divide
5
1
6
3
7
9
8
1
Merge
1 5
3 6
7 9
1 8
Merge
1 3 5 6
1 7 8 9
Merge
1 1 3 5 6 7 8 9

MERGESORT

- The division continues until segments of size 1 are reached. Then the merging begins from the bottom up until the complete data set is re-assembled. The recursive function is:

```
static void mergeSort(int f[], int lb, int ub){
//termination reached when a segment of size 1 reached - lb+1 = ub
 if(lb+1 < ub){
    int mid = (lb+ub)/2;
    mergeSort(f,lb,mid);
    mergeSort(f,mid,ub);
    merge(f,lb,mid,ub);
 }
}
```

MERGESORT

- The value mid forms the upper bound on the first invocation of mergeSort and the lower bound in the second. Each division gives rise to a single merge that takes all three arguments as parameters.

- The function merge uses a temporary array to store the merging data. Its size is always r - p, where p represents the lower bound and r the upper bound. When merging is complete the merged data is copied back to the original array f.

MERGESORT

```
static void merge(int f[], int p, int q, int r)
{
  //p<=q<=r
  int i = p; int j = q;
  //use temp array to store merged sub-sequence
  int temp[] = new int[r-p]; int t = 0;
  while(i < q && j < r){
    if(f[i] <= f[j]){
      temp[t]=f[i];i++;t++;
    }
    else{
      temp[t] = f[j]; j++; t++;
    }
  }
  //tag on remaining sequence
    while(i < q){
      temp[t]=f[i];
      i++;t++;
    }
    while(j < r){
      temp[t] = f[j];
      j++; t++;
    }
    //copy temp back to f i = p; t = 0;
    while(t < temp.length){
      f[i] = temp[t];
      i++; t++;
    }
  }
}
```

MERGESORT

- A simple program to test this sorting algorithm is given below.

```
public class MergeSortTest {
  public static void main(String args[]){
    int d[] = {2,5,1,2,3,6,7,8,4,2,5,3,7,9,1};
    mergeSort(d,0,d.length);
    for(int x : d)
     System.out.print(x + " “);

    System.out.println();
}
```

MERGESORT ANALYSIS

- Merge sort always performs in $O(n * log_2 n)$ because the divide component is $O(log_2 n)$ and merging is $O(n)$. This would suggest that it should outperform QuickSort on average.

- However, this is not the case because there is a hidden overhead. Every time we merge we require a new temporary data array to store the values. This is then copied back to the original

- array. This is an expensive overhead and for large data sets QuickSort outperforms it.

DIJKSTRA'S SHORTEST PATH

- Given a weighted graph comprising a set of vertices V, a set of edges E and a set of weights C specifying weights ci,j for edges (i, j) in E.

- We are also given a starting vertex v in V. The one-to-all shortest path problem is the problem of determining the shortest path from node v to all the other nodes in the graph.

DIJKSTRA'S SHORTEST PATH

- Dijkstra's algorithm starts by assigning some initial values for the distances from vertex v and to every other vertex in the graph. It operates in steps, where at each step the shortest distance from node v to another vertex is determined.

- Each vertex has a state that consists of two features: distance value and status label. The distance value of a node is a scalar representing an estimate of its distance from vertex v.

- The status label is an attribute specifying whether the distance value of a vertex is equal to the shortest distance to vertex v or not. Its status label is Permanent (p) if its distance value is equal to the shortest distance from vertex v; otherwise, its status is Temporary (t) At each step one node is designated as current.

DIJKSTRA'S SHORTEST PATH

- Step 1. Initialization

  Label each vertex as follows:
  (1) Starting vertex v is given zero distance and status permanent, represented by (0,p).
  (2) All other vertices have status (∞, t) Designate the vertex v as the current vertex.

- Step 2. Distance Value Update and Current Node Designation Update

  Let i be the index of the current vertex.
  Find the set J of vertices with temporary labels that can be reached from the current vertex i by a link (i, j). Update the distance values of these vertices using the formula:

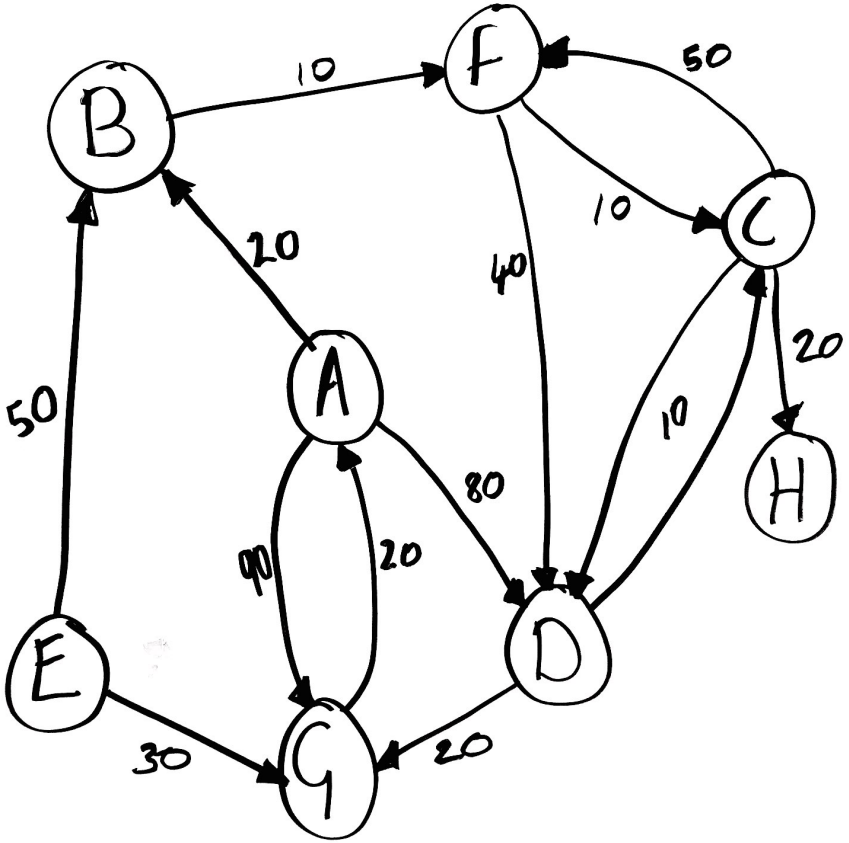
  for each j J, the distance value dj = min {dj , di + cij }, where cij is the cost of link (i, j), as given in the graph.

  Taking vertex j that has the smallest distance value dj among all nodes jJ, change its status to permanent and designate it as the current node.

- Step 3. Termination

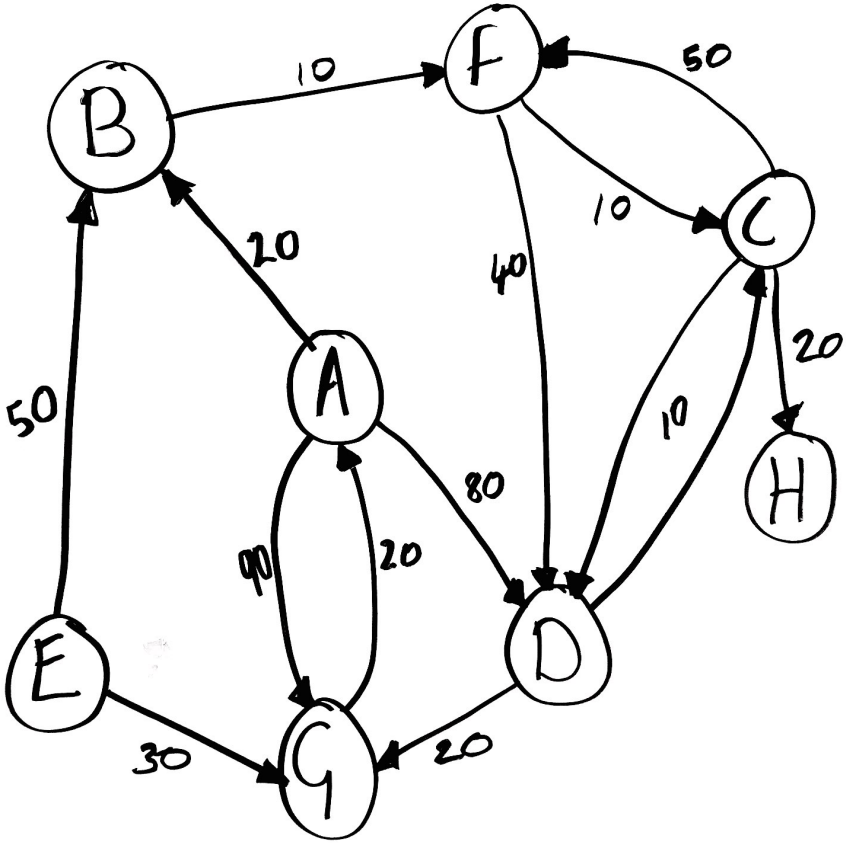
  If all nodes that can be reached from starting vertex v have been labeled permanent then stop. If any temporary labeled vertex cannot be reached from the current vertex change all temporary labels to permanent. Otherwise, go to Step 2.

DIJKSTRA'S SHORTEST PATH



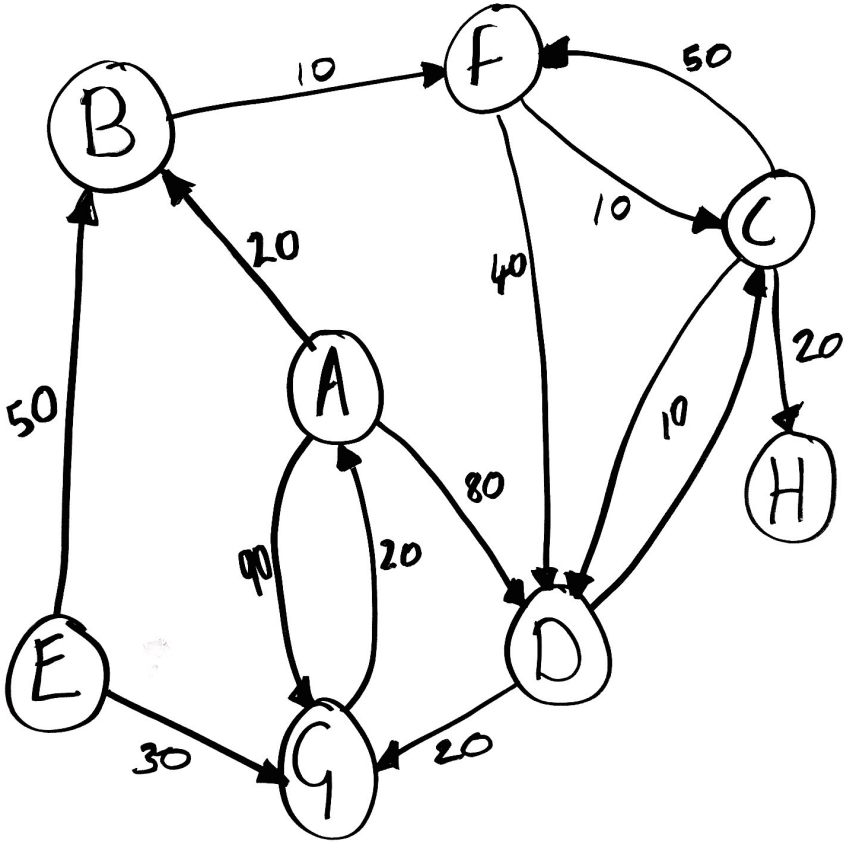
	A →	B	C	D	E	F	G	H
1								
2								
3								
4								
5								
6								
7								
8								

DIJKSTRA'S SHORTEST PATH



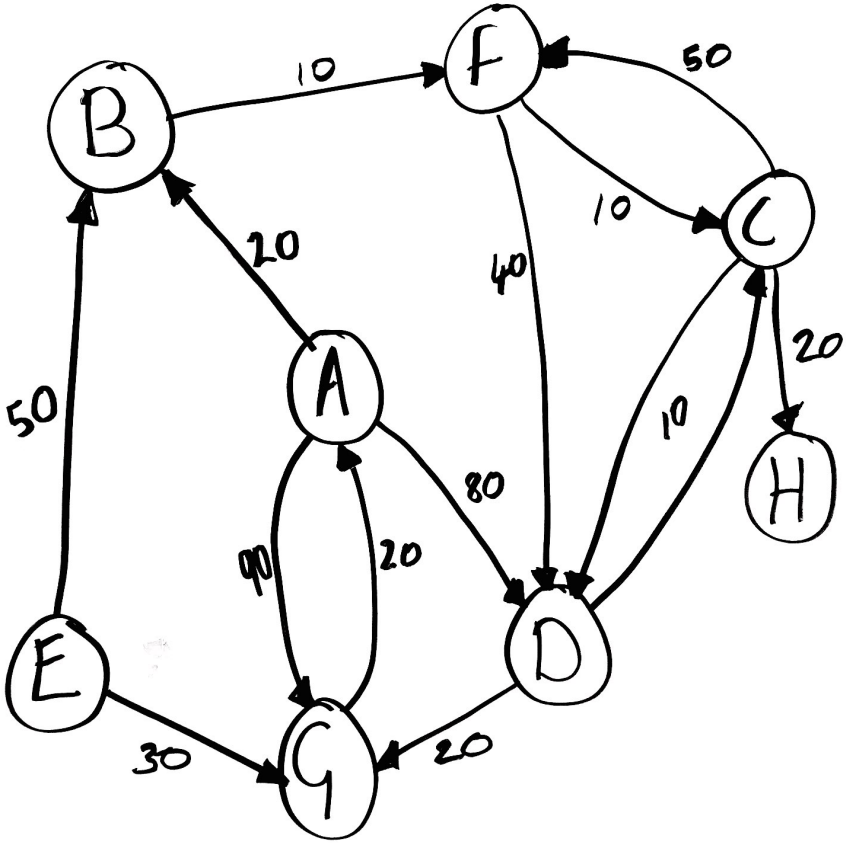
	A →	B	C	D	E	F	G	H
1	A	20 A	∞	80 A	∞	∞	90 A	∞
2								
3								
4								
5								
6								
7								
8								

DIJKSTRA'S SHORTEST PATH



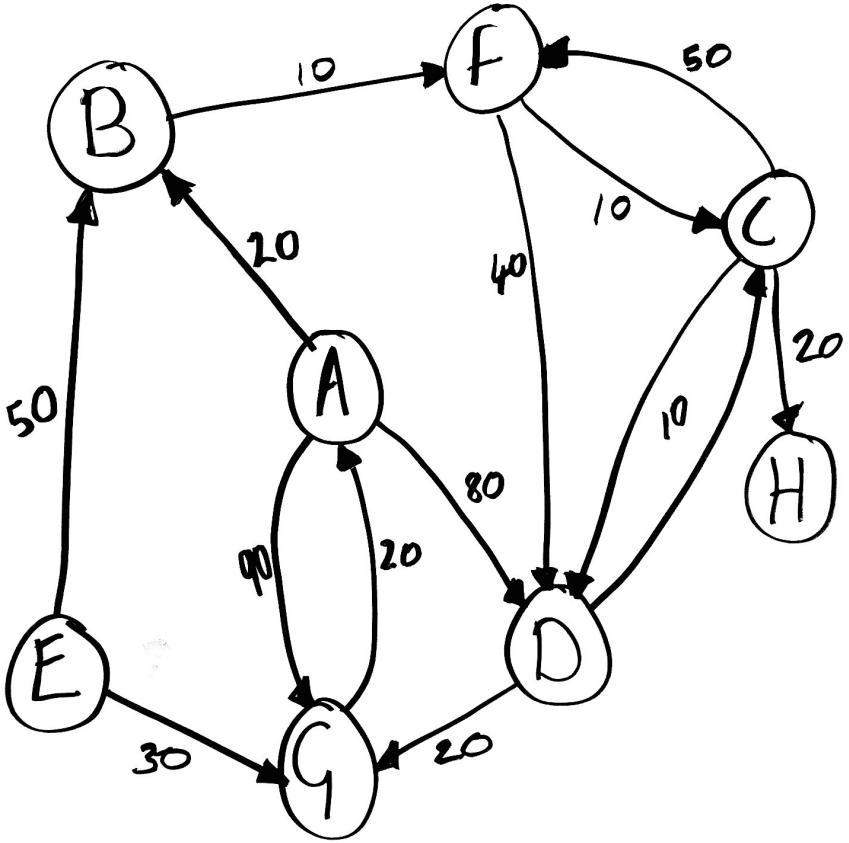
	A →	B	C	D	E	F	G	H
1	A	20 A	∞	80 A	∞	∞	90 A	∞
2	B	20 A	∞	80 A	∞	30 B	90 A	∞
3								
4								
5								
6								
7								
8								

DIJKSTRA'S SHORTEST PATH



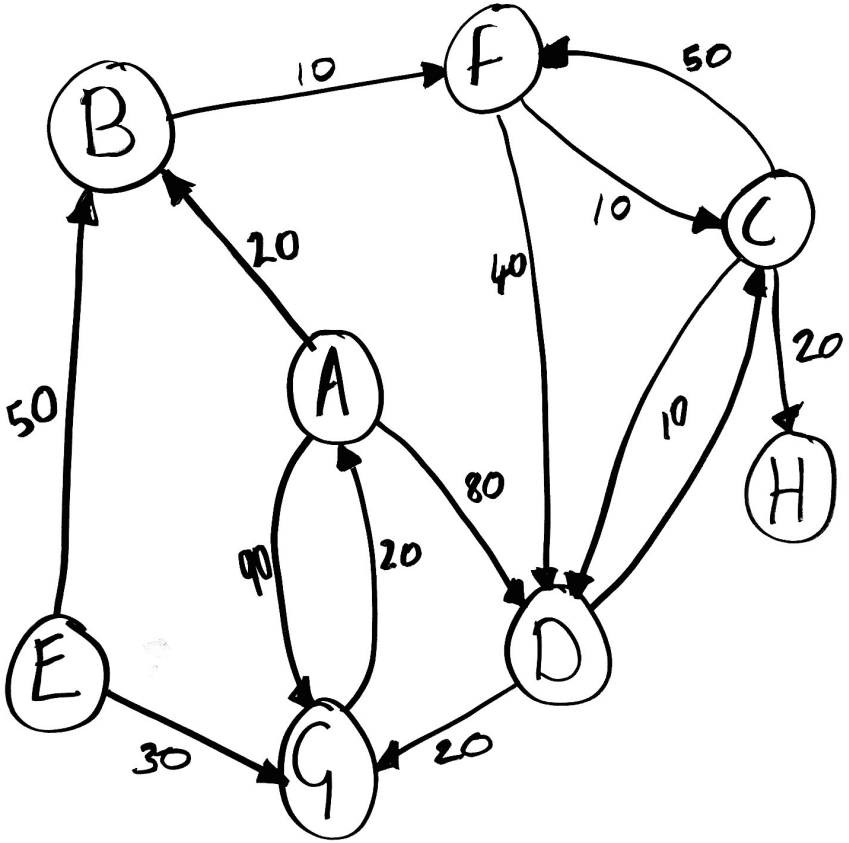
	A →	B	C	D	E	F	G	H
1	A	20 A	∞	80 A	∞	∞	90 A	∞
2	B	20 A	∞	80 A	∞	30 B	90 A	∞
3	F	20 A	40 F	70 F	∞	30 B	90 A	∞
4								
5								
6								
7								
8								

DIJKSTRA'S SHORTEST PATH



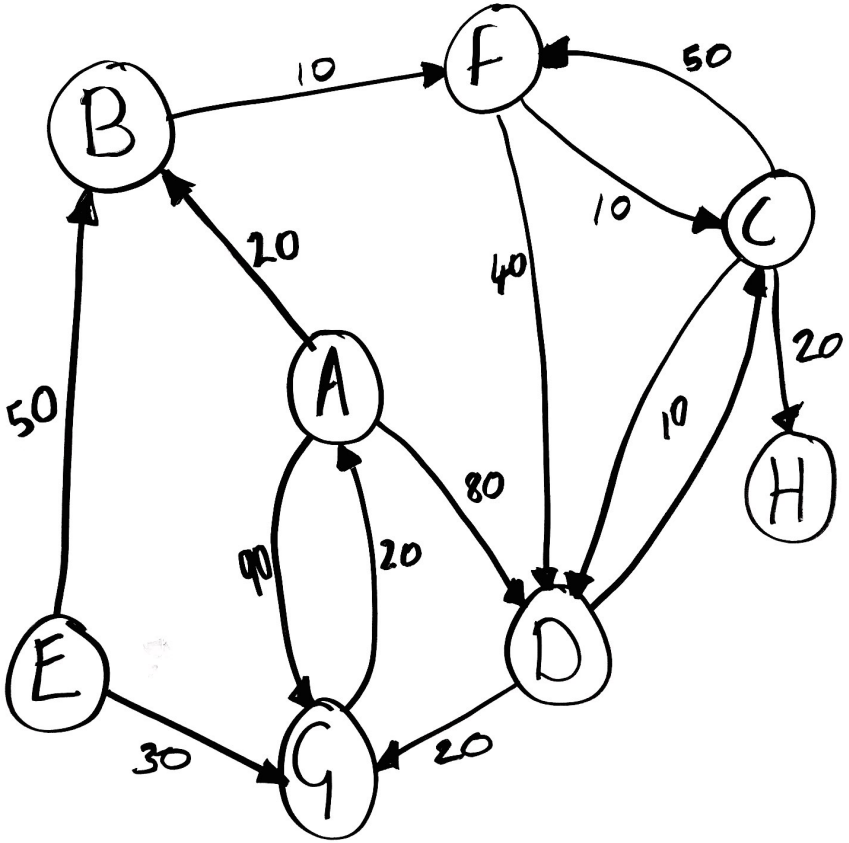
	A →	B	C	D	E	F	G	H
1	A	20 A	∞	80 A	∞	∞	90 A	∞
2	B	20 A	∞	80 A	∞	30 B	90 A	∞
3	F	20 A	40 F	70 F	∞	30 B	90 A	∞
4	C	20 A	40 F	50 C	∞	30 B	90 A	60 C
5								
6								
7								
8								

DIJKSTRA'S SHORTEST PATH



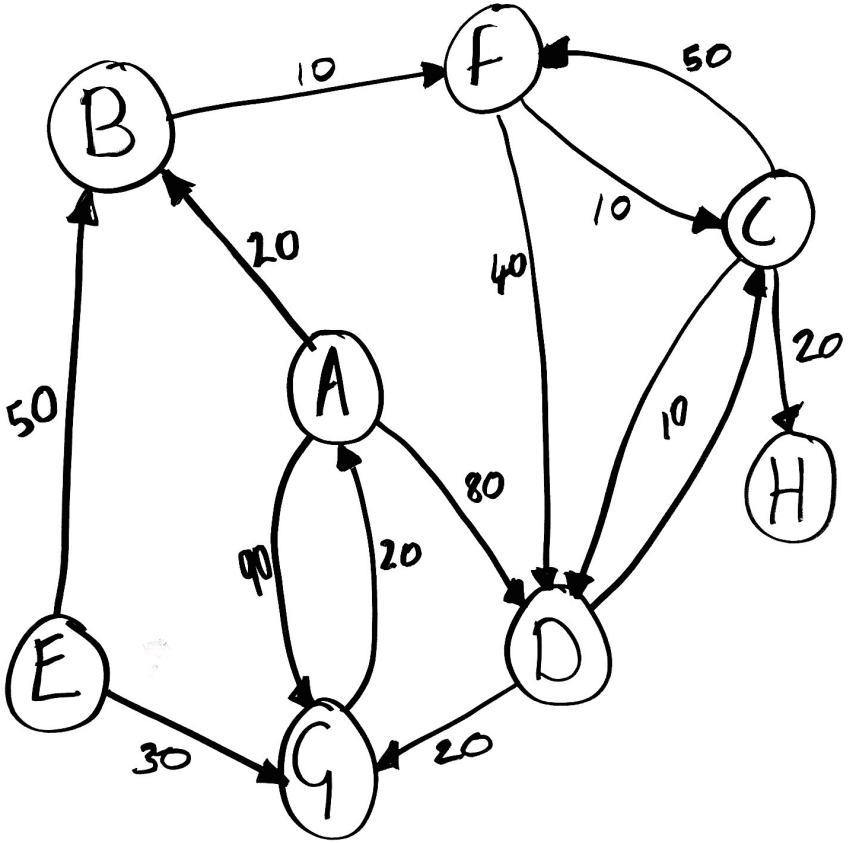
	A →	B	C	D	E	F	G	H
1	A	20 A	∞	80 A	∞	∞	90 A	∞
2	B	20 A	∞	80 A	∞	30 B	90 A	∞
3	F	20 A	40 F	70 F	∞	30 B	90 A	∞
4	C	20 A	40 F	50 C	∞	30 B	90 A	60 C
5	D	20 A	40 F	50 C	∞	30 B	70 D	60 C
6								
7								
8								

DIJKSTRA'S SHORTEST PATH



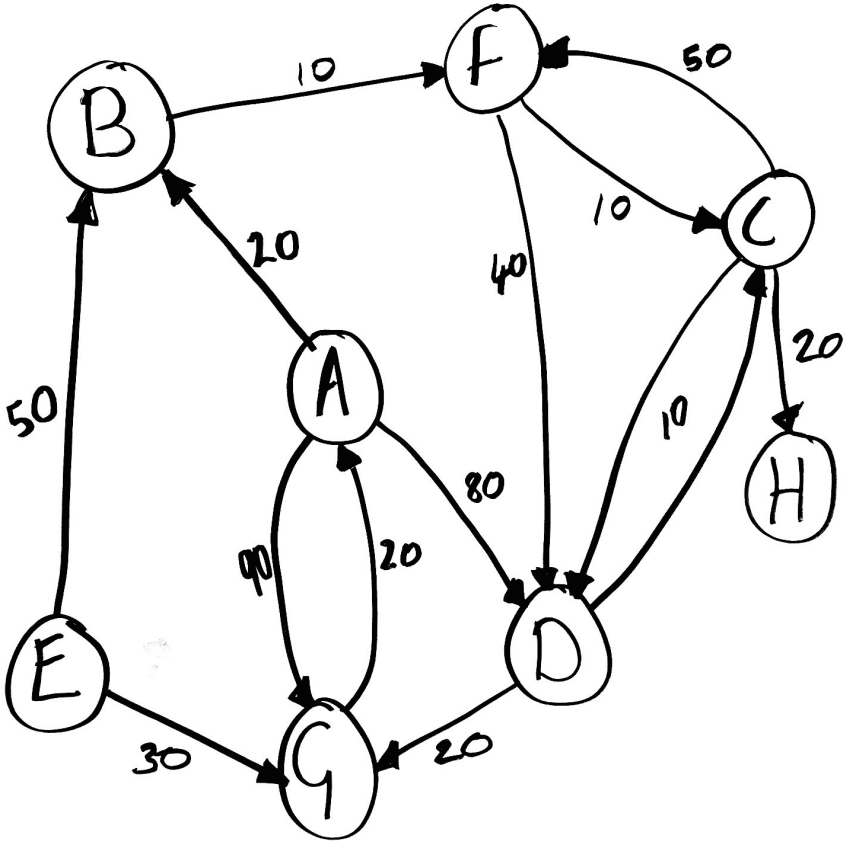
	A →	B	C	D	E	F	G	H
1	A	20 A	∞	80 A	∞	∞	90 A	∞
2	B	20 A	∞	80 A	∞	30 B	90 A	∞
3	F	20 A	40 F	70 F	∞	30 B	90 A	∞
4	C	20 A	40 F	50 C	∞	30 B	90 A	60 C
5	D	20 A	40 F	50 C	∞	30 B	70 D	60 C
6	H	20 A	40 F	50 C	∞	30 B	70 D	60 C
7								
8								

DIJKSTRA'S SHORTEST PATH



	A →	B	C	D	E	F	G	H
1	A	20 A	∞	80 A	∞	∞	90 A	∞
2	B	20 A	∞	80 A	∞	30 B	90 A	∞
3	F	20 A	40 F	70 F	∞	30 B	90 A	∞
4	C	20 A	40 F	50 C	∞	30 B	90 A	60 C
5	D	20 A	40 F	50 C	∞	30 B	70 D	60 C
6	H	20 A	40 F	50 C	∞	30 B	70 D	60 C
7	G	20 A	40 F	50 C	∞	30 B	70 D	60 C
8								

DIJKSTRA'S SHORTEST PATH



	A →	B	C	D	E	F	G	H
1	A	20 A	∞	80 A	∞	∞	90 A	∞
2	B	20 A	∞	80 A	∞	30 B	90 A	∞
3	F	20 A	40 F	70 F	∞	30 B	90 A	∞
4	C	20 A	40 F	50 C	∞	30 B	90 A	60 C
5	D	20 A	40 F	50 C	∞	30 B	70 D	60 C
6	H	20 A	40 F	50 C	∞	30 B	70 D	60 C
7	G	20 A	40 F	50 C	∞	30 B	70 D	60 C
8								

TODO - WEEK 9

Study for your test on Trees and Graphs