Solver: Shao Jie

Email Address: yews0012@e.ntu.edu.sg

1. (a)
   (i) False
   (ii) False
   (iii) True
   (iv) True
   (v) False

   (b)(i)

   Best Case:
   Complexity = **1**
   When A[0][0] ==0

   Worst Case:
   Complexity = **m*n**
   When A[m-1][n-1]==0 or When none of the A[i][j] are 0

   (ii)

Case	Probability
A[0][0]==0	P
A[0][1]==0	(P-1)(P)
A[0][2]==0	(P-1)(P-1)(P)
...	...
A[m-1][n-1]==0	$(P-1)^{(m*n-1)}(P)$
None of them are 0	$(P-1)^{(m*n)}$

Average Case:

$(P + (P-1)(P) + (P-1)^2(P) + (P-1)^3(P) + (P-1)^4(P) + \ldots + (P-1)^{(m*n-1)}(P) + (P-1)^{(m*n)}) / (m*n)$

(c) (i)

$W(i) = 2 + W(i-1) + W(i-2) + W(i-3)$, $i>=4$

$W(i) = 0$, $i<4$

(ii)

$W(i) = 2 + W(i-1) + W(i-2) + W(i-3)$ ($i>=4$)

$= 4 + 2W(i-2) + 2W(i-3) + W(i-4)$ $=2*W(4)-W(1) + 2W(i-2) + 2W(i-3) + W(i-4)$

$= 8 + 4W(i-3) + 3W(i-4) + 2W(i-5)$ $=2*W(5)-W(2) + 4W(i-3) + 3W(i-4) + 2W(i-5)$

$= 16 + 7W(i-4) + 6W(i-5) + 4W(i-6)$ $=2*W(6)-W(3) + 7W(i-4) + 6W(i-5) + 4W(i-6)$

$= 30 + 13W(i-5) + 11W(i-6) + 7W(i-7)$ $=2*W(7)-W(4) + 13W(i-5) + 11W(i-6) + 7W(i-7)$

$= 56 + 24W(i-6) + 20W(i-7) + 13W(i-8)$ $=2*W(8)-W(5) + 24W(i-6) + 20W(i-7) + 13W(i-8)$

$= 104 + 44W(i-7) + 37W(i-8) + 24W(i-9)$ $=2*W(9)- W(6) + 44W(i-7) + 37W(i-8) + 24W(i-9)$

$= 192 + 81W(i-8) + 68W(i-9) + 44W(i-10)$ $=2*W(10)-W(7) + 81W(i-8) + 68W(i-9) + 44W(i-10)$

$= 354 + 81W(i-8) + 68W(i-9) + 44W(i-10)$ $=2*W(11)-W(8) + 81W(i-8) + 68W(i-9) + 44W(i-10)$

....

$W(k) = 2*W(k-1)-W(k-4)$

Since $W(k) > W(k-1)$, and $W(C)$ is always positive

Thus, $2*W(k) > 2*W(k-1)$ for any k.

$2*W(k) > 2*W(k-1) - W(k-4)$ for any k.

Therefore, the growth will always be bounded by 2^k

And since the time complexity of addition is $O(2^n)$ then it must be $O(3^n)$.

(iii)

```
arr[]
Int Tribonacci(int n){
        If(n<=2){
                arr[n]=0;
                return 0;
        }
        If(n<=4){
                arr[n]=1;
                return 1;
        }
        return Tribonacci(n-1) * 2 – arr[n-4];  // this is a simplified equation from (ii)
}
```

Foreach input parameter used in the Tribonacci will be recorded in arr. Therefore, the current input parameter can make use of the previously computed value from the recurrence equation. Thus lesser time complexity needed to compute, but extra space is needed to perform the action.

2.(a)

(i) both f(n) and g(n) are quadratic $O(n^2)$

f is in Ω (g), true

f is in Θ(g), true $\lim_{n\to\infty} \frac{5n^2 + \log(n)}{\sqrt{n^4 + n^2 + 1}} = 5$

f is in O(g), true

(ii)

f is in Ω (g), true $\lim_{n\to\infty} \frac{3^n}{3^{n/2}} = \infty$

f is in Θ(g), false

f is in O(g), false

(b)

Given, $f(n) \in \Theta(h(n))$, $g(n) \in \Theta(h(n))$

$f(n) + g(n) \in \Theta(h(n)) + \Theta(h(n))$

$\Theta(h(n)) + \Theta(h(n)) \in \Theta(h(n))$

$f(n) + g(n) \in \Theta(h(n))$

(c)(i)

0	33		
1			
2	24	57	
3			
4			
5			
6	17	39	6
7			
8			
9			
10			

(ii)

Number of Probes to search 57: 2
Number of Probes to search 60: 1 (searched index 5 and found nothing)

(d)

The advantage is that closed addressing hash table's only get linearly slower as the load factor (the ratio of elements in the hash table to the length of the bucket array) increases, even if it rises above 1.
And on the other hand open-addressing gets very, very slow if the load factor approaches 1, because you end up usually having to search through many of the slots in the bucket array before you find either the key that you were looking for or an empty slot.

The disadvantage of closed addressing hashing is having to follow pointers in order to search linked lists. While in open-addressing only need to search through the indexes which had collision when the value was inserted.

3(a)

Quicksort is not stable. For example, we have array of 4, 3a, 3b, 5, 1. Assuming (3a=3b) and 3b is chosen to be the pivot. Let the checking condition be

"if (item>=pivot), move to right side of pivot."

Thus in this case, the sorted array will 1, 3b, 3a, 4, 5. Which means that the original order of 3a and 3b is not preserved.

Merge sort is stable if we ensure that the merging of list starts from the 1st half then the 2nd half when two element are equal. For example, using the same array 4, 3a, 3b, 5, 1.

[4, 3a, 3b, 5, 1]

[4, 3a, 3b]		[5, 1]	
[4, 3a]	[3b]	[5]	[1]
[4] [3a]	[3b]	[5]	[1]
[3a, 4]	[3b]	[1, 5]	
[3a, 3b, 4] ← 3a is moved in 1st then 3b		[1, 5]	

[1, 3a, 3b, 4, 5]

However, if we move the 1st element of the 2nd half of the list to the end of merged list instead of the 1st element of the 1st half during merging, the outcome will be different. And the original order of two equal elements will not be preserved.

[4] [3a]	[3b]	[5]	[1]
[3a, 4]	[3b]	[1, 5]	
[3b, 3a, 4]		[1, 5]	

[1, 3b, 3a, 4, 5]

(b)

[A, L, G, O, R, I, T, H]

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

[A, L, G, O]		[R, I, T, H]		
[A, L]	[G, O]	[R, I]	[T, H]	
[A][L]	[G][O]	[R][I]	[T][H]	
[A,L]	[G,O]	[I,R]	[H,T]	4 comparisons
[A, G, L, O]		[H, I, R, T]		6 comparisons
[A, G, H, I, L, O, R, T]				6 comparisons

(c)

Insertion Sort

Best Case: n-1 comparison
Worst case: 1+2+3+4+...+n-1
Average case: The i-th iteration may have 1 to i comparison each with 1/i chance.

$1+(1/2)(1+2)+(1/3)(1+2+3)+\ldots+(1/(n-1))(1+\ldots+n-1)$

$$=\sum_{i=1}^{n-1}\left(\frac{1}{i}\frac{i(i+1)}{2}\right)=\frac{1}{2}\sum_{i=1}^{n-1}(i+1)=\frac{1}{2}\left(\sum_{i=1}^{n}i-1\right)=\frac{1}{2}\left(\frac{n(n+1)}{2}-1\right)$$

$= (n^2+n)/4 - 1/2 = \Theta(n^2/4)$

(d)

(A) It is a maximising heap, as each node has a greater or equal key value than each of its child nodes

(B) It is a maximising heap, as each node has a greater or equal key value than each of its child nodes

(C) Not a maximising heap, as a node with value 5 is a parent of a child node with value 7 which is smaller.

(e)

Assume the heap has n elements, k levels,

$2^{k-1}-1 < n$

$2^{k-1} <= n$

$2^{k-1} <= n <= 2^k-1$

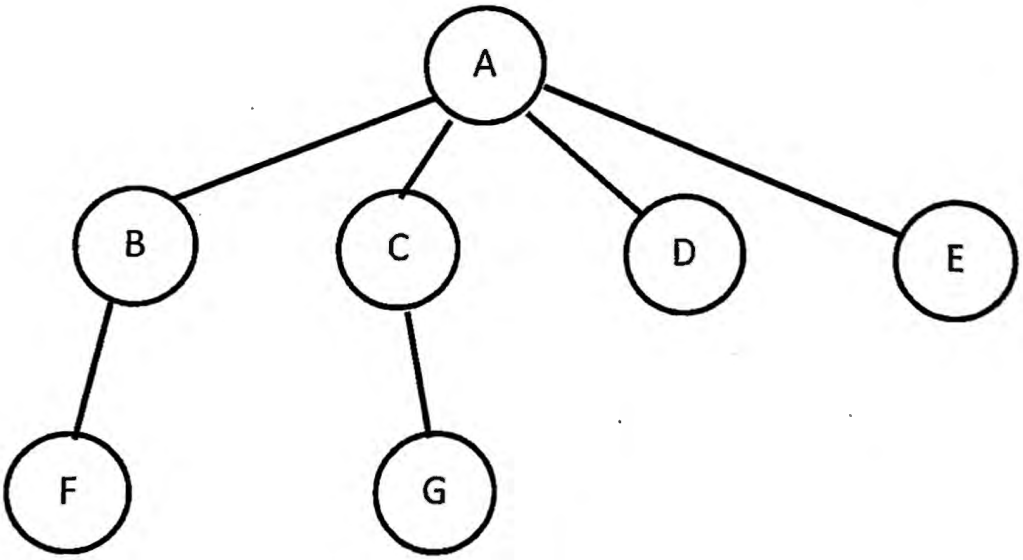
$k-1 <= \lg n < k$

$k-1 = \lfloor \lg n \in \Theta(\lg n)$

4. (a)

Adjacency lists is more efficient than adjacency matrix. This is because |E| is upper bounded by |V|. If we use adjacency matrix, the amount of space used to store the maximum number edge between each vertex will be relative to V^2. However, since |E| is bounded by |V| the amount of space used will be $(|V|/V^2)$ which will not be very efficient as V increase. Take for example, if we have 4 vertexes, only (4/16) of the space is utilized.

(b)

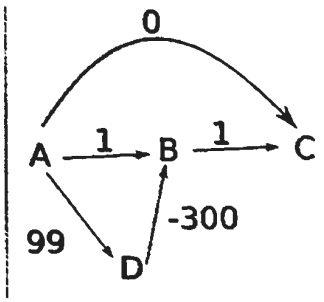


(c)

Order of vertices	A	B	C	D	E
A	{A,0}	-	-	-	-
A,B	{A,0}	{A,5}	-	-	-
A,B,C	{A,0}	{A,5}	{B,1}	-	-
A,B,C,D	{A,0}	{A,5}	{B,1}	{B,3}	-
A,B,C,D,E	{A,0}	{A,5}	{B,1}	{B,3}	{D,2}

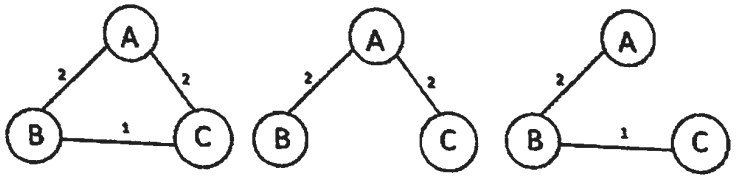
(d)

Dijkstra is a greedy algorithm. Whenever selecting a next edge to add to the shortest path tree it selects the one which has minimum weight and is reachable from already reached nodes. The greedy property here is that, when a next edge with minimum weight is added, the weight of the path from source to new node of that next edge will be minimum. Hence, making it the shortest path. This greedy property will only hold if weights are positive.



However, if there exist negative edge. You will get the minimum weight of path to C to be 0 which should instead be -200. In this case the graph doesn't adhere to the greedy property mentioned earlier.

(e)



Prim's algorithm takes the graph, and returns a minimum weighted tree. Dijkstra's algorithm takes the graph and the node A, and returns a function that gives shortest paths for each node

Prim's algorithm finds a minimum spanning tree for a graph by starting with an arbitrary node and then repeatedly adding the shortest edge that connects a new node. Invariance to the choice of starting node is a property of the minimum spanning tree problem.

Dijkstra's algorithm finds the shortest paths from a selected node to all nodes in the graph. It does this by starting with the selected node and then repeatedly adding the shortest edge that leads to a new node. Note how the choice of starting node is important this time; it's one of the inputs of the algorithm.