21st CSEC – Past Year Paper Solution (2017 – 2018 Semester 2)
CX2001 – Algorithm

1 (a) (i) FALSE. It implies that Θ(n) is symmetric, not reflexive.

(ii) TRUE. It follows the property of transitivity.

(iii) TRUE. In addition, the best-case complexity is Θ(1).

(iv) FALSE. The maximum load factor for open address hashing is 1.

(v) TRUE. A problem in NP-Complete implies that it is also NP and NP-Hard.

(b) (i) It will iterate through the first six elements without repetitions. Hence, there are 6 comparisons.

(ii) The best case is when "BE" is the first two characters in the array. This only requires 2 comparisons regardless the number of characters in the array. The time complexity is Θ(1). Meanwhile, the worst case is when there are no substrings in the array or 'E' does not appear before the final character. This way, the system needs to iterate through every letter in the array. The time complexity is Θ(n).

(iii) If 'E' is found in the n^{th} character, there are n comparisons needed.
The average time-complexity can be found by adding the number of comparisons needed in all possible cases.

Hence, the average time-complexity is $\frac{2+3+4+..+n}{n-1} = \frac{\frac{(n+2)(n-1)}{2}}{n-1} = \frac{n+2}{2}$, which is Θ(n).

(c) (i) T(n) = T(n-1) + T(n-2) + 2

(ii) We may treat the equation above as quadratic equation $m^2 - m - 1 = 0$

$$m = \frac{1 \pm \sqrt{5}}{2}$$

Hence, $T(n) = A * \left(\frac{1+\sqrt{5}}{2}\right)^n + B * \left(\frac{1-\sqrt{5}}{2}\right)^n + C$

$C = C + C + 2$

$C = -2$

Through further substitution of n = 0 and n = 1,

$A = \frac{5+\sqrt{5}}{5}$ and $B = \frac{5-\sqrt{5}}{5}$

We may observe that $\left(\frac{1+\sqrt{5}}{2}\right)^n \sim 2^n$. Q.E.D.

An alternative to the algorithm is to use iteration instead of recursion.
It will iterate from 2 up to n, storing the last two values of the sequence. This will reduce the time complexity to Θ(n).

(*Algorithm with pseudocode as comments*)

```
int mFibonacci (int n):
{
  int a=0, b=1, c; // Initialize variables
  if (n==0) {
```

```
    return 0;
  }
  if (n==1) {
    return 1;
  }
  for (int i=2; i≤n; i++) { // Iterate up to n
    c = 2*b + a; // Calculate following value
    a = b; // Store latest two values
    b = c;
  }
  return c; // Return latest value of the sequence
}
```

2 (a) (i)

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = \lim_{n\to\infty}\frac{\log_2 n^2}{\log_4 n^4}$$
$$= \lim_{n\to\infty}\frac{2 * \log_2 n}{4 * \log_4 n}$$
$$= \lim_{n\to\infty}\frac{\log_2 4}{2}$$
$$= \frac{\log_2 4}{2}$$

Hence, f is in O(g), f is in **Ω**(g), and f is in **Θ**(g).

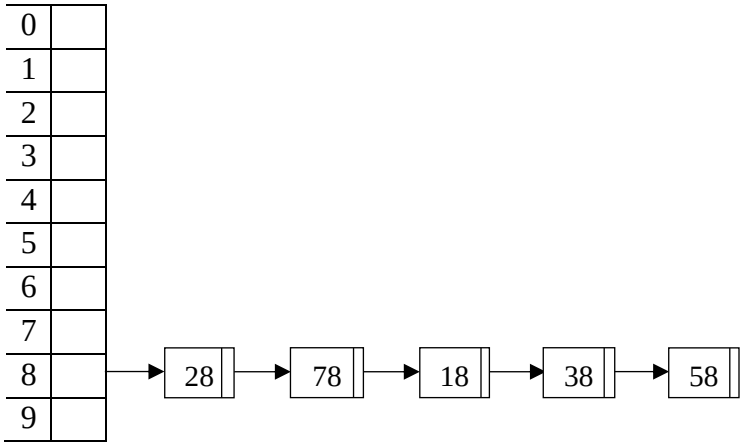
(ii)

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = \lim_{n\to\infty}\frac{(2n^3)^{\frac{1}{2}}}{2n+1}$$
$$= \lim_{n\to\infty}\frac{\sqrt{2}n^{\frac{3}{2}}}{2n+1}$$
$$= \lim_{n\to\infty}\frac{\frac{3\sqrt{2}}{2}n^{\frac{1}{2}}}{2}$$
$$= \infty$$

Hence, f is NOT in O(g), f is in **Ω**(g), and f is in **Θ**(g).

(b) $f(n) \in \Omega(g(n))$ implies that $\lim_{n\to\infty}\frac{f(n)}{g(n)} \neq 0$. To observe the relationship between g(n) and f(n), we need to search the value of $\lim_{n\to\infty}\frac{g(n)}{f(n)}$, which equals to $\frac{1}{\lim_{n\to\infty}\frac{f(n)}{g(n)}} \neq \frac{1}{0} = \infty$. We have proven that $\lim_{n\to\infty}\frac{g(n)}{f(n)} \neq \infty$. This implies that $g(n) \in O(f(n))$. Hence, the statement is TRUE.

(c) **(i)**



28 is first inserted to index 8, knowing that 28 mod 10 = 8 and index 8 is still empty. 78 is next. 78 mod 10 = 8, so it needs to be inserted to index 8. It checks whether index 8 is empty, it is NOT. So, it is linked to the next address in the list. Later, we have found out that all 18, 38, and 58 are all hashed to index 8. They will all be linked to the list.

(ii)

Key values	Number of comparisons
28	0
78	1
18	2
38	3
58	4

The total number of key comparisons is 0+1+2+3+4 = 10 key comparisons.

(iii) When searching for 78, it checks the first space, which contains 28. Then, it checks the next in list, which is 78, our target. Hence, 2 comparisons are done.

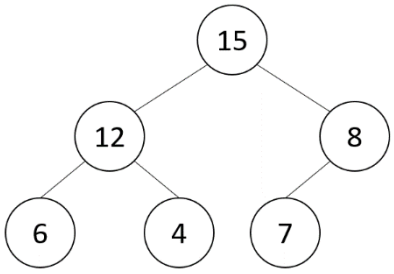
Note that 68 does not exist in the hash table. The system would check all available elements in index 8, before finding out that 68 does not exist in the table. Hence, 5 comparisons are done.

(iv) The main issue of the current design is the possibility of one key being overused due to lack of randomness, especially when the load factor approaches 1. An improvement is to use double hashing in form of an open address hashing. Two numbers may not be mapped into one index and a number is rehashed should its mapped key is occupied. This will add a layer of randomness to the hashing algorithm. However, the drawback is the decrease in the hash table's storage space.

3 **(a)** The linear-time algorithm involves using a hash table of size 4n. For every value of key in the array, it increments the key in the hash value by 1. This stage would have a time complexity of n ($\Theta(n)$).

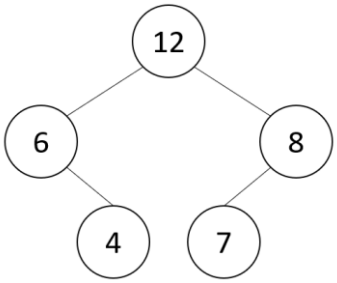
Then, compare each value of the key index in the hash table and return the highest value. This process would have a time complexity of 4n (Θ(n)) as we traverse through all 4n indexes of the hash table. In total, the time complexity of this algorithm depends on the greatest complexity among the stages, in this case, Θ(n).

(b) **(i)**

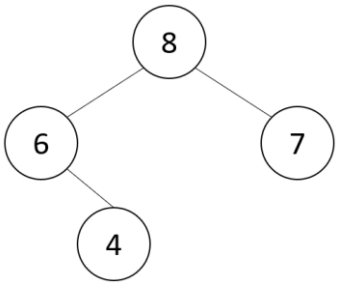


Note that all child nodes MUST have smaller values compared to their parent node.

(ii) *(not required) the tree after one call of deleteMax()*



The tree after two calls of deleteMax()



(c) One of the worst-case input arrays is when all the values in the array are sorted in an increasing order. This way, the pivot would always be the smallest number of the bunch, hence each iteration would never divide the group of members into two non-empty subgroups. (one subgroup will always have no members)
Worst-case input array of size n: [1, 2, 3, …, n]

The first pivot is 1. There will be (n-1) key comparisons in this iteration.
The second pivot is 2. There will be (n-2) key comparisons in this iteration.
There will always be one less key comparison in every iteration until there is only one member, n.

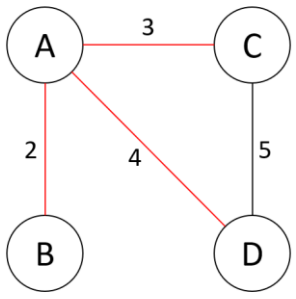
The number of key comparisons is $(n-1)+(n-2)+\cdots+1=\frac{n(n-1)}{2}=\frac{1}{2}n^2-\frac{1}{2}n$. This results in a time complexity of $\Theta(n^2)$.

(d) We will first search the time complexity needed to perform insertion sort the n/s sub-arrays. Each sub-array consists of s members. Note that the worst time-complexity to insert sort an array of n members is n^2. Hence, the time needed to sort 1 sub-array is s^2. Some more, sorting n/s sub-arrays will require a time of $\frac{n}{s}*s^2=ns$.

Now, we have n/s sorted sub-arrays. To perform MergeSort, we first sort $\frac{n}{2s}$ groups of 2 sub-arrays (of size s). Note that the worst time-complexity to merge sort 2 arrays of n members is n-1. Hence, the time needed to perform this first stage is $\frac{n}{2s}*(2s-1)\sim n$. This would result on n/2s sorted sub-arrays (of size 2s). This method would go all the way until there only remains 1 sorted array of size n. The number of stages, i, can be found using $2^i*s=n\rightarrow\ 2^i=\frac{n}{s}\rightarrow i=\lg(\frac{n}{s})$. Hence, the time complexity of performing MergeSorts is $n*\lg(\frac{n}{s})$.

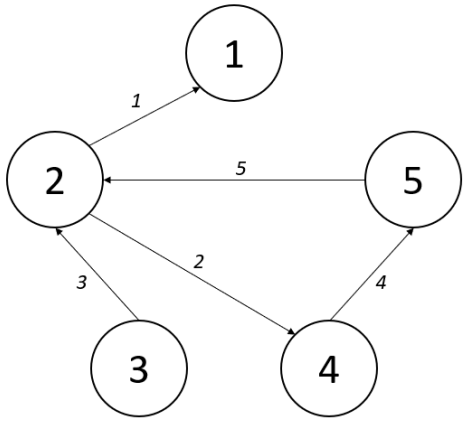
Using these two complexities we acquire, the worst-case time complexity of this algorithm is $ns+n*\lg\left(\frac{n}{s}\right)=n(s+\lg\left(\frac{n}{s}\right))$. Q.E.D.

4 **(a)** Quoting from the Lecture Slides, "Let T be a spanning tree of G, where G = (V, E, W) is a connected, weighted graph. Suppose that for every edge (u, v) of G that is not in T, if (u, v) is added to T it creates a cycle such that (u, v) is a maximum-weight edge on that cycle. Then T has the Minimum Spanning Tree Property."



Consider a graph G above, whose minimum spanning tree T is marked in red. (C, D) is not included in T. If we add (C, D) to T, it will create a cycle ACD. Here, we may see that edge (C, D) has the highest weight among the cycle's three edges (AC-3, AD-4, and CD-5). We may conclude that (C, D) is the maximum-weight edge on that cycle. Hence, it has been proven that T is indeed a minimum spanning tree of the graph G.

(b)

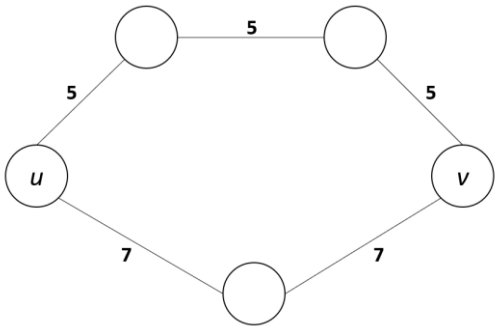


The number in *italic* indicates the edge index.
Incidence matrix may be seen below,

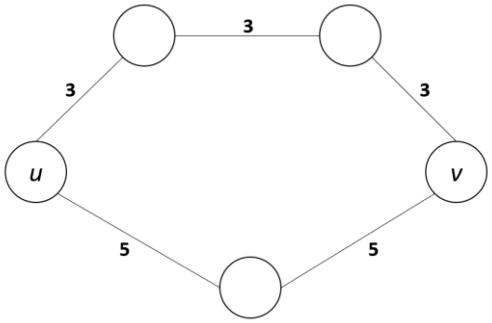
	Edges					
Vertices		*1*	*2*	*3*	*4*	*5*
	1	1	0	0	0	0
	2	-1	-1	1	0	1
	3	0	0	-1	0	0
	4	0	1	0	-1	0
	5	0	0	0	1	-1

(c) We may simply transform the adjacency matrix M by transposing the initial matrix. We initialize the new matrix M' such that M[i][j] = M'[j][i]. This way, $(i,j) \in E \rightarrow (j,i) \in E'$.
As we iterate through every cell in the matrix, time complexity equals to $\Theta(|V|^2)$.

(d) No. One of the counter examples is given below.
Given a graph G,



The number in **bold** indicates the weight of the respective edge. Here, the shortest path P goes through the lower path where the path length equals to 7+7=14.
If all the weights are decreased by 2, refer to the graph G' below,



Now, the path P has a length of 5+5=10, which is longer than the upper path P' whose length equals to 3+3+3=9. This occurs because there are more nodes in the upper path, hence this path is more affected to the change in weights. (An easier way to understand this is, the upper path's length is reduced by 3*2 = 6 and the lower path's length is only reduced by 2*2=4)

The shortest path P will always remain the same if and only if the number of nodes in each path is the same.

Solver: Leonardo Irvin Pratama (lpratama001@e.ntu.edu.sg)