24th SCSE – Past Year Paper Solution (2023 – 2024 Semester 1)
SC2001 – Algorithm Design & Analysis

1 (a) (i) The answer is **False**
The number of comparisons in both the cases would remain the same. Hence, using a linked list won't affect the time complexity.

(ii) The answer is **True**
Since the median will be chosen as the pivot, it will, in every case, divide the list into 2 almost-equal parts. Thus, the worst-case time-complexity would be improved to $O(\text{nlgn})$.

(iii) The answer is **False**
This is because a path with a greater number of nodes would be more affected than a graph with fewer nodes. A path with 3 nodes would have its weight increased by 3*offset while a path with 2 nodes would have its weight increased by 2*offset.

(iv) The answer is **False**
Prim's algorithm has no such limitation on the edge weight unlike Djikstra's algorithm.

(v) The answer is **True**

(b) (i) The best-case time complexity for the given algorithm in terms of "swaps" would be **O(1).** Consider the list is [1,2,3]. Since it is already sorted, no "swaps" would be done by the algorithm. Hence, it would have a constant time complexity.

This best case happens when the list is already sorted. In this case, since the swapped variable would not be set to True after the first for loop, the function would end.

(ii) **Best Case**
In this case, the list is already sorted. The for loop runs only once. Hence the number of comparisons that take place are (n-1). Therefore, the time-complexity is $\mathbf{O(n)}$.

Worst Case
In this case(reverse sorted list), the for loop would run n times. Each iteration of the for loop does (n-1) comparisons. Hence, the total comparisons would be $n * (n-1)$. The time complexity is $\mathbf{O(n^2)}$**.**

Average Case
On average, the for loop will be run n/2 number of times. Each for loop does (n-1) comparisons. Hence, total comparisons would be $(n-1) * \left(\frac{n}{2}\right)$. The time complexity is $\mathbf{O(n^2)}$**.**

2 **(a)** MergeSort works by repeatedly dividing an array into 2 parts and then merging them using a "merge" function.

Best Case:
In the best-case scenario, the input array is already sorted. In this case, the merging step is still performed. The merge function in this step would perform only (n/2) comparisons.
The recurrence relation is:

$$T(n) = T\left(\frac{n}{2}\right) + \left(\frac{n}{2}\right)$$

We can solve this relation using the Master Method (Case 2: $f(n) = O\left(n^{\log_b a}\right)$)
The time complexity would be $O(n * logn)$

Worst Case:
In the worst-case scenario, the input array is in reverse order. In this case, each level of recursion will involve the maximum number of comparisons for the merge function (n-1). The recurrence relation is:

$$T(n) = T\left(\frac{n}{2}\right) + (n-1)$$

We can solve this relation using the Master Method (Case 2: $f(n) = O\left(n^{\log_b a}\right)$)
The time complexity would be $O(n * logn)$

Average Case:
The average case time-complexity would be bound by the best- and worst-case time complexity. We can use squeeze theorem to analyse the average-case.

$$T_{worst}(n) \leq T_{average}(n) \leq T_{best}(n)$$

Using squeeze theorem, we can conclude that the average-case time complexity of Mergesort should also be $\boldsymbol{O(n * logn)}$.

(b) In its original form, Kruskal's algorithm selects edges in ascending order based on their weights and adds them to the spanning tree if adding the edge does not create a cycle. To modify Kruskal's algorithm to find a maximum spanning tree, you can simply reverse the order in which edges are considered, choosing edges in descending order based on their weights.

Let's use a proof by contradiction to demonstrate the correctness of the modified Kruskal's algorithm for finding a maximum spanning tree.

- Suppose the tree *T* produced by Kruskal's is not a Maximum Spanning Tree, i.e., it doesn't satisfy the Maximum Spanning Tree property.
- Hence, there is some edge *u*-*v* not in *T* such that adding *u*-*v* creates a cycle, in which some other edge *x*-*y* has weight W(*x*-*y*) < W(*u*-*v*).

- As W(*x*-*y*) < W(*u*-*v*), edge *x*-*y* must be processed AFTER edge *u*-*v* in our modified algorithm.
- At the time when *u*-*v* is processed, it must have been added into T because it doesn't form a cycle at that time.
- This contradicts that *u*-*v* is not in *T.*
- Hence, Kruskal's algorithm can be used to find the Maximum Spanning Tree.

3 **(a)** **(i)**

$$W(1) = 0$$
$$W(n) = W(n-1) + n + 1$$

Expanding $W(n-1)$,

$$W(n) = (W(n-2) + (n-1)) + 1 + n + 1$$
$$W(n) = W(n-2) + \big(n + (n-1)\big) + 2$$

Expanding $W(n-2)$,

$$W(n) = (W(n-3) + (n-2) + 1) + \big(n + (n-1)\big) + 2$$
$$W(n) = W(n-3) + (n + (n-1) + (n-2)) + 3$$

Furthur expanding,

$$W(n) = W(n-(n-1)) + (n + (n-1) + (n-2) \ldots . + 2) + (n-1)$$
$$W(n) = W(1) + \sum_{i=0}^{n-2} (n-i) + (n-1)$$
$$W(n) = 0 + n^2 + n - 1$$

Hence, the solution for the recurrence equation is,

$$\boldsymbol{W(n) = \theta(n^2)}$$

(ii)

$$W(n) = 3W\left(\frac{n}{2}\right) + cn$$

Hence,

$$f(n) = O(n)$$
$$a = 3$$
$$b = 2$$

$$n^{\log_b a - \varepsilon} = n^{\log_2 3 - \varepsilon} = n^{1.585 - \varepsilon}$$
$$f(n) = O(n^{1.585-\varepsilon})$$

Hence, using the first case of master method, the solution for the recurrence equation is,

$$\boldsymbol{W(n) = \theta(n^{1.585})}$$

(b) Text: BOWNOWOWWWOLOWL
Pattern: WOWWOW

Simple Boyer-Moore

j is the index of the element on the text

k is the index of the element on the pattern.
m is the length of the pattern = 6
When there is a mismatch, the following shift will occur:

$$j += \max(\text{charJump}[T[j]], m - k + 1)$$

							W	O	W	W	O	W			
						W	O	W	W	O	W				
					W	O	W	W	O	W					
T	W	O	W	W	O	W									
P	B	O	W	N	O	W	O	W	W	W	O	L	O	W	L
Ind	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

First mismatch occurs at j = 4 (comparisons=3)
j will shift by charJump[N] = 6

Second mismatch occurs at j = 9 (comparisons=2)
j will shift by (m-k+1) = 2 (since charJump[W] < (m-k+1))

Third mismatch occurs at j = 11 (comparisons=1)
j will shift by charJump[O] = 1

Fourth mismatch occurs at j = 12 (comparisons=1)

No more comparisons since further shifting would push j out of the length of the text.
Total character comparisons = 7

Boyer-Moore Algorithm
j is the index of the element on the text
k is the index of the element on the pattern.
m is the length of the pattern = 5
When there is a mismatch, the following shift will occur:

$$j += \max(\text{charJump}[T[j]], \text{matchJump}[k])$$

							W	O	W	W	O	W			
						W	O	W	W	O	W				
T	W	O	W	W	O	W									
P	B	O	W	N	O	W	O	W	W	W	O	L	O	W	L
Ind	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

First mismatch occurs at j = 4 (comparisons=3)
j will shift by matchJump[N] = 7

Second mismatch occurs at j = 11 (comparisons=1)
j will shift by charJump[O] = 1

Third mismatch occurs at j = 12 (comparisons=1)
No more comparisons since further shifting would push j out of the length of the text.
Total character comparisons = 5

(c) The required code is as below, with the lines that are changed underlined:

```
Void RKscan(P1,P2,P3,T) {
 m=Length(P1);
  n=Length(T);
  dM=1;
  for (j=1 to m-1)  dM=dM*d%q;
 hp1 = hash(P1, m, d);
 hp2 = hash(P2, m, d);
 hp3 = hash(P3, m, d);
 found1, found2, found3 = 0;
  ht = hash(T, m, d);
  for (j=0; j<=n-m; j++)  {
    if (hp1==ht && equal_string(P1, T, 0, j, m))
     {  output j; found1= 1;}
    if (hp2==ht && equal_string(P2, T, 0, j, m))
     {  output j; found2= 1;}
    if (hp3==ht && equal_string(P3, T, 0, j, m))
     {  output j; found3= 1;}
    If (j<n-m)  ht=rehash(T, j, m, ht); }
 if (found1==0){ output "Pattern 1 not found";}
 if (found2==0){ output "Pattern 2 not found";}
 if (found3==0){ output "Pattern 3 not found";}
  return; }
```

Editor's note: The original return statements inside the if condition were removed since we need to output ALL instances of the three patterns. Three new found variables are introduced to counter the possibility that a certain pattern is not found within the text. Since the length of all 3 patterns is the same, no need to have multiple m and dM variables for each pattern.

4 **(a)** **(i)** The array of the matrices' dimensions is [10, 2, 20, 5, 30]. The cost and last computed are as below:

Cost	**0**	**1**	**2**	**3**	**4**
0		0	400	300	1100
1			0	200	500
2				0	3000
3					0
4					

Last	0	1	2	3	4
0		-	1	1	1
1			-	2	3
2				-	3
3					-
4					

Using the last matrix, we can conclude that the most optimal multiplication would be in the following order:

$$\boldsymbol{A} \times ((\boldsymbol{B} \times \boldsymbol{C}) \times \boldsymbol{D})$$

<u>Editor's note:</u> The recursive definition of the cost function that can be used to get the cost matrix is as follows:

$$Cost(i,j) = \begin{cases} 0\,, & if\ i+1=j \\ \min\big(Cost(i,k) + Cost(k+1,j) + w_i * w_j * w_k\big), \forall\ k\epsilon[i+1, j-1] \end{cases}$$

(ii) The method of considering the position of the first matrix multiplication does not work for finding the optimal order of matrix multiplications. This approach leads to incorrect results because the problem does not exhibit the optimal substructure property.

Optimal substructure property in dynamic programming means that the optimal solution to the problem can be constructed from the optimal solutions of its subproblems. In the case of matrix chain multiplication, considering only the position of the first multiplication does not satisfy this property.

Let's take an example with the matrix dimensions represented by the array $d = [0, 1, 2, 3, 4]$. Here, we have four matrices with dimensions:
$d_0 \times d_1,\ d_1 \times d_2,\ d_2 \times d_3, d_3 \times d_4$.

If we consider the position of the first multiplication, we might start with $d_0 \times d_1$ or some other matrix. Let's consider $d_0 \times d_1$ as the first multiplication.

Now, we need to decide how to multiply the remaining matrices ($d_1 \times d_2$, $d_2 \times d_3\ and\ d_3 \times d_4$). The problem arises here because the optimal solution for the remaining matrices might require a different starting point.

In contrast, the standard matrix chain multiplication problem, where we consider the position of the last multiplication, exhibits optimal substructure. The optimal solution for the entire problem can be constructed from the optimal solutions of its subproblems.

Hence, the matrix chain multiplication problem with the standard method (considering the position of the last multiplication) exhibits optimal substructure and can be solved using dynamic programming. However, the prefix method does not provide a correct solution for the entire problem.

(b) The shortest link algorithm is a heuristic approach to solving the travelling salesman problem. We select the edges in increasing order of weights and choose an edge if it is not making a cycle and it is not the third edge on any vertice.

Edge	Weight
a-c	2
c-d	3
d-a	4
d-e	5
c-b	6
c-e	7
a-e	8
a-b	9
e-b	10

We first select the edge a-c
Then we select the edge c-d
We do not select the edge d-a since it is forming a cycle
We select edge d-e
We do not select the edge c-b since it would be a third edge on vertice c
We do not select the edge c-e since it would be a third edge on vertice c

We do not select the edge a-e since it is forming a cycle
We select edge a-b
Finally, we select edge b-e since all the edges have been reached.

Sequence of edges chosen: (a-c), (c-d), (d-e), (a-b), (b-e)
Weight of the path chosen = 2+3+5+9+10= **29**

This solution is NOT an optimal solution. A better solution would have been :
(a-c), (c-b), (b, e), (e-d), (d-a)
The weight of this path = 2+6+10+5+4 = **27**

The greedy heuristic to solve the travelling salesman problem is a good one since it provides us with a decent solution in polynomial time. The travelling salesman problem is a NP-hard problem, which means that it can be solved optimally only using exponential time complexity algorithms. The greedy heuristic provides us a way to get a “good-enough” solution in most cases using polynomial time complexity. While it does not guarantee the absolute optimal solution, it often provides solutions that are close to optimal and can be obtained in a reasonable amount of time. Hence, the Greedy Heuristic approach is a suitable solution to the travelling salesman problem.

Editor’s note: To show that the solution we obtained using the shortest link path is not optimal, you need to use trial-and-error to find a better solution.

Solver: Raghav Gupta (raghav008@e.ntu.edu.sg)