Design & Analysis of Algorithms

Assignment #2

Ahmad Abdullah i22-1609

Question #1 (a)

- Reasons to use Merge Sort over Quicksort

- Worst time complexity of merge of sort is $O(n \log n)$ regardless of input while Quick sort's worst case time complexity is $O(n^2)$. In this case Merge sort is more reliable.

- Merge sort performs better on linked list since it does not rely on random access which is better for large data sets.

- Merge sort divides ideal array into independent subarrays which allows for parallel processing which is a lacking feature in quicksort.

- Merge sort is preffered when stability is required. Since it preserves relative order of the elements.

(b)

- Quick Sort's worst-Case performance

The worst-case of quick sort depends on the pivot position selection. Poor selection of pivot position lead to worst case performance.

- Quick sort gives worst-case performance when:
  1) The selection pivot is smallest or greatest number.
  2) The array is already sorted or reversed-sorted.

(c)

- Insertion Sort performing Faster than expected

Insertion Sort has a best-case time complexity of O(n), which occurs when the array is already sorted or nearly sorted. In such cases, each element requires minimal shifting to find its correct position which makes the algorithm give faster performance.

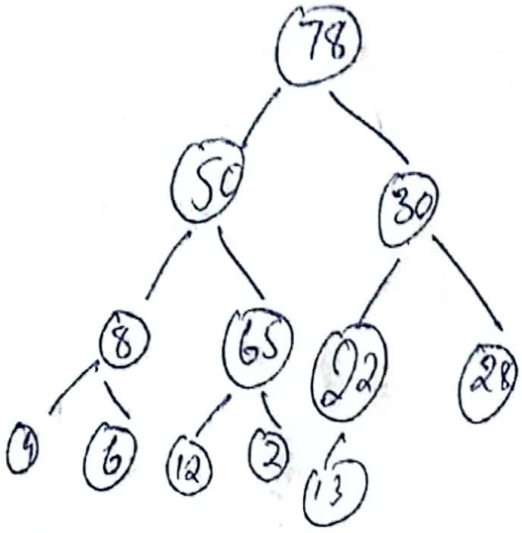
(d)

- Sorting an Array with n-Elements and k Out of place Elements

Insertion sort is well suited for an array that is mostly sorted with only k elements out of place. Insertion sort can quickly place those k-elements in their correct position with minimal processing. It's average case complexity would be O(n+k).

Question # 2 (a)

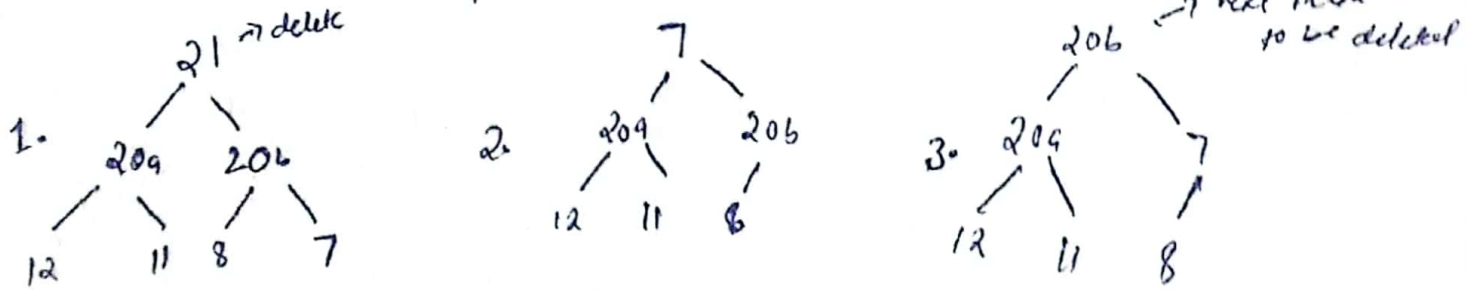
(a) Checking if an array is Max-Heap



The provided array is not a max-heap because 65 is bigger than 50 as its leaf node.

(b)

- Stability of Heap Sort

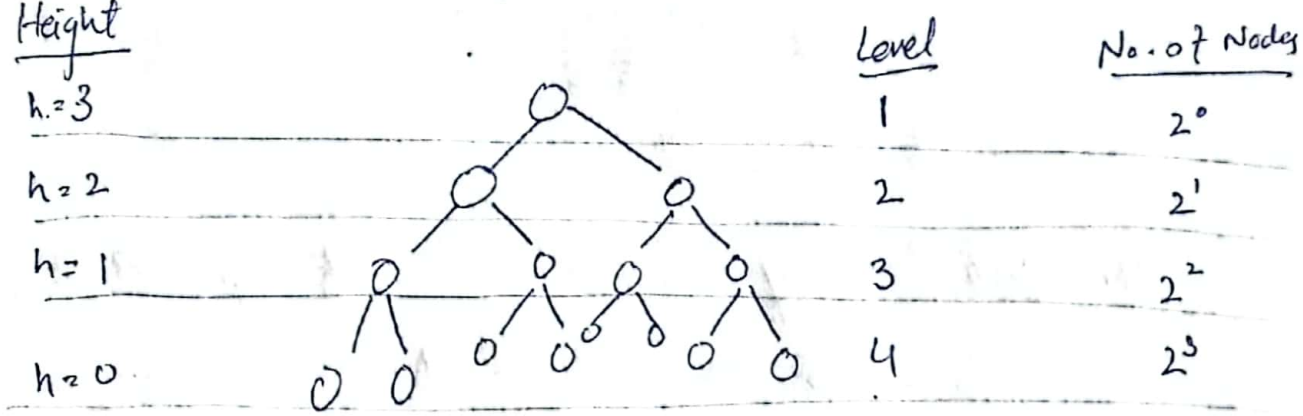


Since next element that will be output/delete is 206 which is out of order in which the array was provided so the Heap Sort is unstable.

- Time Complexity

Building a heap tree or heapify method takes cost on a node 'i' is proportional to the height of the 'i' in the tree ie $T(n) = O(n)$

Consider a following tree with height (h) = 3



Cost of Heapify at level i * number of nodes at that Level $\Rightarrow T(n) = \sum_{i=0}^{h} n_i h_i$

$$= \sum_{i=0}^{h} 2^i (h-i)$$

$$= \sum_{i=0}^{h} \frac{2i\,(h+i)\,2^h}{2^{h-i}}$$ ∵ Multiplied with 2^h

$$= 2^h \sum_{k=0}^{h} \frac{k}{2^k}$$ ∵ replacing i with k

$$2^h \sum_{k=0}^{h} \frac{k}{2^k} \leq n \sum_{k=0}^{\infty} \frac{k}{2^k}$$

$= O(n)$ ∴ Running time of Build-heap

$T(n) = O(n)$.

Question #3 (a)

Swap Ends(S)

```
Var = S.removeFromFront();
S.addtoFront(S.removeFromBack());
S.addtoBack(var);
```

} O(1)

(b)

ShiftLeft(S,k)

```
for i to k
    S.addtoBack(S.removeFromFront());
```

} O(k)

Question #5 (1)

Naïve String Matchmaking: Best & Worst case

Let

- $T = t_1, t_2, t_3, \ldots, t_n$ be a string of length n
- $P = P_1, P_2, P_3, \ldots, P_n$ be the pattern of length m

  where $m < n$

→ Best Case: Best case for naive string matching algorithm

1) The first few character from pattern do not match with the string. This will make our machine work faster by allowing us to skip several comparisons.

2) A mismatch is detected at each position in T. which minimizes comparisons.

Example: Let the T = "abcef" and pattern P = "zd"
the algorithm compares the pattern with string
P[0] to T[0] which is a match and so on

In this case, only one comparison per character so
the time complexity is O(n).

→ Worst-Case: worst case occurs where the string contains
repeated patterns and the pattern mostly identical
to the given string

Example: Let the string T = "aaaaaaab" and pattern P = "aabb"
when comparing P to T[0:3], no mismatch
so the algorithm has to perform m comparisons.
After first comparison, the second P to T[1:4] we
again have to do m comparisons
so the complexity for this scenario is
O(n x m).

---

(2)

- Efficiency of the Naive Algorithm

To improve the efficiency of Naive Algorithm
we can use another algorith Knuth-Morris-Pratt or do
some tweaks in Naive algorithm. I did some changes
in Naive algorithm.

if the last character of the current substring does not match the last character of the pattern then we should skip multiple characters in the text.

Example: Let P = "abc" and T = "aaabcabc"

Now starting by aligning P with the begining of T
If there is a mismatch at the last character, we can safely skip some characters in the text.

Code:

```
void NotNaiveAnymore(string text, string pattern) {
int m=text.length();
int n= pattern.length();
while (i <= n-m){
    int j= m-1;
    while ( j>=0 && pattern[j] == text[i+j]){
            j--;
    }
    if(j==-1){
      cout << "Pattern" << i;
        i+=m; }
    else{ int skip = max(1,j);
      i += skip;
    }
  }
}
```

Question #6 (b)

Pattern = 'aabaabcab'

P[0] = a has no proper prefix or suffix

P[1]

index	Pattern [0..i]	Lsp (i)
0	a	0
1	aa	1
2	aab	0
3	aaba	1
4	aabaa	2
5	aabaab	3
6	aabaabc	0
7	aabaabca	1
8	aabaabcab	2

The prefix function of Pattern is: [0, 1, 0, 1, 2, 3, 0, 1, 2]

(b)

Code:

```
void PrefixFunction(char* pattern, int m, int* lps){
  int j = 0;
  lps[0] = 0;
  for(int i = 1; i<m; i++){
     while(j>0 && pattern[i] != pattern[j]){
          j = lps[j-1];}

     if (pattern[i] == pattern[j]){ j++;}
      lps[i] = j;
  }
}
```

Question # 4 (4)

Selecting Top K Gardeners

we need to find the top scoring gardeners and return their registration numbers

we use min-heap for this for k number of gardeners

The min-heap will store gardeners score if there are less than k entries in the min-heap.

If the min-heap is full than we check if the current gardeners score s_i is greater than smallest score in the heap and than replace it. This will give us top K gardeners along with their registration numbers.

Time Complexity: Insertion and deletion complexity = $O(\log k)$

- Processing all gardens in A takes = $O(|A|)$
- Heap operation during processing = $O(k \log k)$

⇒ Overall time complexity = $O(|A| + k \log |A|)$

(b)

- Scores Exceeding Threshold:

First we make a max-heap related to scores of the gardners. Then we use iterative approach to traverse throughl the heap to find the gradnes with score above threshold(x).

In this if we find a node that has score less or equal to x we terminate the process. So we only traverse the heap only x times.

The time complexity hence will be = $O(nx)$

Question # 8

- Rabin-Karp for 2D-pattern

To extend Rabin-karp for for 2D-pattern and string let's take a string in a matrix of size nxn and pattern of size mxm.

First we calculate the hash of the pattern by rolling hash technique and then we calculate the hashes of the string matrix but take the size nxn and shift is horizontally and vertically respectively and then comparing the hashes.

Question # 7 (a)

Spurious hit in Rabin-Karp:

3 1 4 15 9 2 6 5 3 5 8 9 7 9 3

P = 26

q = 11

hash for P = 26 % 11 = 4

Now, we calculate the hashes for the string

h = 31 % 11 = 9

h2 => 14 % 11 = ((31 - 3.10).10.4) % 11

= 10 + 4 % 11

= 10 + 4 % 11

= 14 % 11

= 3

15 = 4

59 = 4

92 = 4

26 = 4

65 = 10

53 = 9

35 = 2

58 = 3

89 = 1

97 = 9

79 = 2

93 = 5

4 matches

Number of spurious hits = 3

Pattern occurs at shift = 6.

(b)

C++ code for Part A:

```
void RabinKarpMatcher(string& T, string& P, int d, int q){
  int n = T.size(); int m = P.size(); int h = 1; int p = 0; int t = 0;
  int Hits = 0;

  for(int i = 0 ; i < m-1 ; i++){
    h = (d*h) % q;
  }

  for(int i = 0 ; i < m ; i++){
    p = (d*p + P[i]) % q;
    t = (d*t + T[i]) % q;
  }

  for (int s = 0; s <= n-m; s++) {
    if (p == t){
      bool match = true;
      for (int i = 0; i < m; i++){
        if(T[s+i] != P[i]){
          match = false;
          break;}}
```

```
if (match) {
    cout << "Shifts" << s ;}
else { shifts++;}

if (s<n-m) {
   t=(d*( t-T[s] *h + T[s+m]) %q);
     if (t<0){
        t= t+q;}
   }
```

Question # 9 (a)

Boyer-Moore-Horspool is an algorithm used for string matching. This algorithm skips many characters for efficiency.

→ Another algorithm that can match multiple patterns is Aho-Corasick Algorithm. This helps in search engines, spam filters etc.

Example for Aho-Corasick: Suppose we have multiple patterns 'he', 'she', 'his' and 'hers' and the text is ushers.

- Building a Trie, a trie is prefix tree in which each node represents a character in the pattern.
- Then add failure links, this helps in efficient backtracking and prevents repeated comparisons.
- Process the text, in which we do automation for comparison

(b)

Code:

```
Void buildTrie(){
    for(int i=0; i<pattern.size(); i++){
        TrieNode* nod = root;
        for(char c : pattern[i]){
            if(!node->children.count(c))
                node->children[c] = new TrieNode();
            node = node->children[c]; }
        node->output.push_back(i); }

Void FailureLinks(){
    while(!q.empty()){
        TrieNode* current = q.front();
        q.pop();
        for(auto& p : current->children){
            char ch = p.first;
            child->fail = (fail)? fail->children.count : root;
            q.push.(child) } } }

Void search (string& text){
    TrieNode* node = root;
    for(int i=0; i<text.size(); i++){
        char ch = text[i];
        while(node && !node->children.count(ch))
            node = nod->fail;
        if(!node){ nod = root;}
        else{ node = node->children[ch]; }
```

```
if (!node->output.empty()){
    for (int pattern : node->output)
        cout<<"Pattern "<< patterns[pattern];}
}
}
}
```

c)

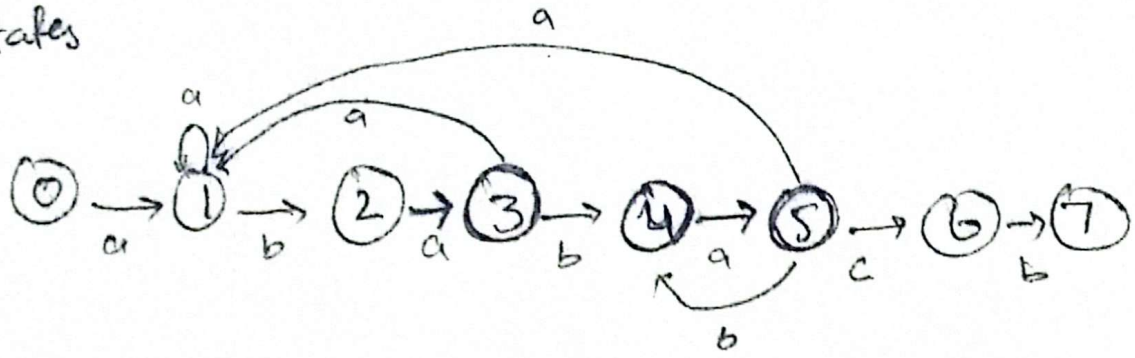
(1)

let the string be "ababababcb"

pattern be "ababacb"

String	State
a	1
b	2
a	3
b	4
a	5
b	4
a	5
c	6
b	7

States



2) (i)

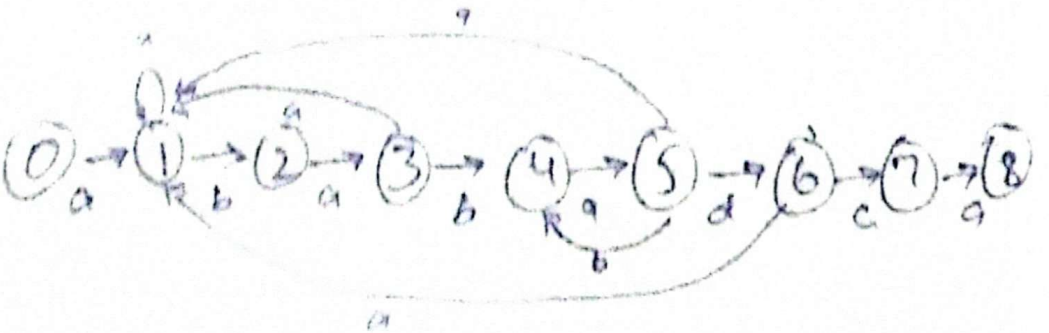
1. a b a b a b d c a a b a b a d c a

S. a b a b a b d c a

A	state
a	1
b	2
a	3
b	4
a	5
b	4
d	6
c	0
a	1
a	1
b	2
a	3
b	4
a	5
d	6
c	7
a	8

state	a	b	c	d
0	1	0	0	0
1	1	2	0	0
2	3	0	0	0
3	1	4	0	0
4	5	0	8	0
5	1	4	0	6
6	1	0	7	0
7	2	0	0	0

String matches at 8th state



(ii)

A= a b a b a d c a a b a b a d c a d

A	State
a	1
b	2
a	3
b	4
a	5
d	6
c	7
a	8 → String matched
a	1
b	2
a	3
b	4
a	5
d	6
c	7
a	8 → String matched
d	1
c	2

(iii)

A= a b a b a b d c b a b a b a d c b

A	State
a	1
b	2
a	3
b	4
a	5
b	4
d	0
c	0
b	0
a	1
b	2
a	3

String Not found

A	State
b	4
a	5
d	6
c	7
b	0