Algorithms and Complexity COMP_90038

Semester 2 August2020
Name: Sakshi Chandel
Student ID:1124298

Question 1:

Solution 1(a):

The basic idea of this algorithm is to initialize an array Z of size such that it can have elements of both array X and Y(size of array X+ size of array Y).Then from array X put all elements to array Z .Then check each element of array Y if it is present in array X using binary search .If present ,don't add to array Z else add to array Z. Hence will get union of elements from array X and array Y in array Z.

```
procedure  FindSetUnion(X,Y )
        n ← size of array X  ;
        k ← n;                                          // n is the size of array X and Y
        initialize  array Z with size 2n;                // create array Z of size 2*n
        for i ← 0  to i ←n-1 do :                       //place all elements of set X to array Z
           Z[i]  ← X[i];
       for j  ← 0 to j  ← n-1 do :      // using binary search to find each element of Y in X
          index  ← BinarySearch(X,0,m,Y[j]); //complexity of binary search  -> logn
          if(index = -1)                    // if any element of Y is not in X then add to array Z
             Z[k]  ← Y[j];
             k++;
```

Since here size of both arrays is n that is why complexity will be n+nlogn,
which is in O(nlogn).

Question 1:
Solution 1 (b):

If we assume there is no collision ,then we can know that search and insert function is in O(1).
But if there is collision ,we can create a hashtable of size k (maximum of largest element in set X and set Y) which will avoid the collision and then search and insert function will be in O(1).But will definitely increase the space complexity if k is very large.

"Assuming there is no collision"

```
procedure FindSetUnion(X,Y)
    initialise hashtable as  hash_table;
    n  ← size of X ;                                   //size of X=size of Y =n
    for i ← 0 to i ← n-1 do :
        insert(hash_table,X[i]);              //insert elements from set X to hashtable
    for j ← 0 to j←n-1 do:
        //lookup will insert elements from set Y to hashtable if not present in hastable
        lookup(hash_table,Y[j]);
```

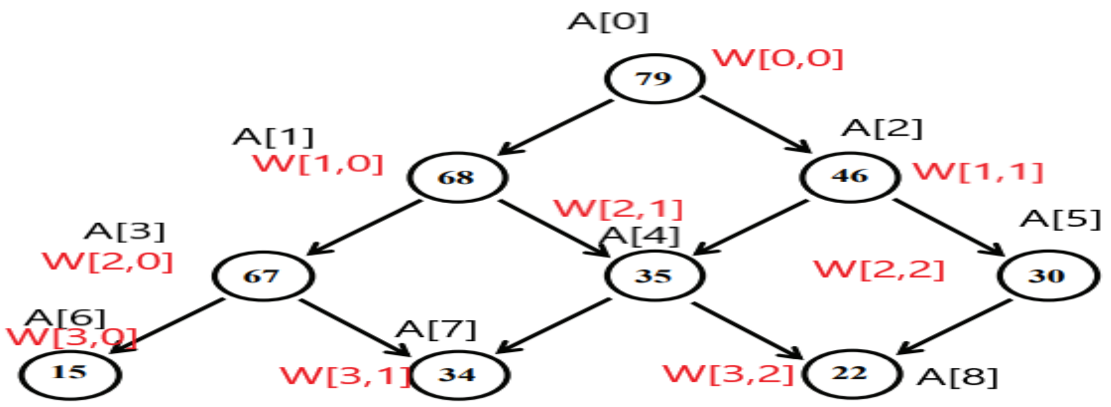
"Assuming collision"

```
procedure FindSetUnion(X,Y)
   //largestElementInSet(x) will return largest element in set X
    k=max(largestElementInSet(X), largestElementInSet(Y));
    initialise hashtable as  hash_table of size k ;       // hash function =element % k
    n  ← size of X ;                                    //size of X=size of Y =n
    for i ← 0 to i ← n-1 do :
        insert(hash_table,X[i]);          //insert elements from set X to hashtable
    for j ← 0 to j←n-1 do:
        //look up will insert element to hashtable if not present in hastable
        lookup(hash_table,Y[j]);
```

Inserting a key **into a hash table** is **O**(1) and lookup is **O(1)** , since **look up** the bucket is a constant time operation. But Iterating an array of size n will make it a O(n) time complexity .

Overall complexity = O(n)

Question 2:
Solution 2 (a):
As shown in the diagram :



Since we can see that each level i such that 0<=i< l , has i+1 nodes.
Then,
Level 1 (i=0) has 1 nodes
Level 2 (i=1) has 2 nodes
Level 3 (i=2) has 3 nodes and so on.. This is forming a triangular sequence.
1+2+3………………….i= (i)(i+1)/2
(since 1+2+3….n = n*(n+1)/2)
Sum of all nodes till level i= i(i+1)/2

i) Wi,j = **A[(i*(i+1))/2 + j]** (0<=i<l)
The node Wi,j will have index in array as sum of all nodes to level where l=0 to l=i plus j (since each level has nodes j starts with o to i)
Example : $W_{3,1}$ = A[3*4/2+1]= A[7], $W_{2,2}$= A[2*3/2+2] =A[5]

ii)Left and right children of Wi,j
Left child = **A[((i+1)*(i+2))/2 + j]**
Right child= **A[((i+1)*(i+2))/2 + j + 1]**

Sum of nodes from 0 to level i+1 plus j (since each level j starts with 0 to i) gives left child
Sum of nodes from 0 to level i plus j+1(since each level j starts with 0 to i) gives right child
Example : W3,1 : Left child = A[4*5/2+1]= A[11] ,
Right child = A[4*5/2+1+1] = A[12]

iii) Left and right parents of Wi,j:

Left parent = **A[((i-1)*(i))/2 + j-1]** if(j>=1) & if j=0 then there is no left parent. Since j=0 to j=i .And also every node has two children. If j=0 then parent will be -1 and 0 which is not possible.

Right parent= **A[((i-1)*(i))/2 + j]**

Example : $W_{3,1}$: Left parent = A[(2*3/2)+1-1]= A[3] ,

Right child = A[((2*3)/2)+1] = A[4]

If in $W_{i,j}$ j = 0 then there will not be left parent as it will be the first element of the level and it will not have a left parent (0<=j<=i)

iv) Left and right children of the node corresponding to A[k] with level i:

As we can see ,this web data structure will form a triangular sequence, and the first element of each level will be a triangular level.

If an index k is given ,then we can calculate level (i) by formula:

Here i is level and j is 0<=j<=i of node Wi,j.

```
i = floor(-0.5 + sqrt(0.25 + 2 * k))
triangularNumber = i* (i + 1) / 2
j= k – triangularNumber
Left child = A[(i+1)*(i+2)/2 +j]
Right child =A[(i+1)*(i+2)/2 +j+1]
```

Example : k=7 then

i=floor(-0.5+sqrt(0.5+2*7)) , i=3

j= 7- 4*3/2 , j=1

Left child = (4*5/2)+ 1= 11 .Hence left child A[11]

Right child = (4*5/2)+ 1 + 1 = 12.Hence right child A[12]

Solution 2 b):

Upper bound and lower bound of level l
Upper bound nodes n= (l*(l+1))/2
Lower bound nodes n = ((l-1)*l)/2 + 1

Explanation : If the level of web is l , then for upper bound(maximum nodes) number of nodes will be keeping in the mind the structural property that the all but the last level will be completely filled.
Also , any level will have at most l nodes where l>=1 then upper bound will have maximum l nodes in the last level where l>=1.
Hence for level 1 to level l :
Total nodes will be 1+2+3+……l
(Since sum of 1+2+3…to n = n(n+1)/2)
And by formula sum of 1+2+3…..l = l*(l+1)/2.

If the level of web is l , then for lower bound(minimum nodes) number of nodes will be keeping in the mind the structural property that the all but the last level will be completely filled.
Also, any level will have at most l nodes where l>=1 then for lower bound will have minimum 0 nodes in the last level where l>=1.
Hence for level 1 to level l:
Total nodes will be 1+2+3…..+l-1+0 (where 0 node is at level l)
(Since sum of 1+2+3…to n = n(n+1)/2)
And by formula sum of 1+2+3…..+l-1+0= (l-1)*(l)/2 .(Here, n=l-1)

Solution 2 c):

According to the **Order Property** of this web data structure The priority of a node is always greater than or equal to the priority of either child node. By this the maximum element of the web structure will be always $W_{0,0}$ i.e A[0].

- Replace $W_{0,0}$ with the element farthest of the web data structure.(Replace A[0] by A[n-1]).
- Now we need to maintain the order property and structure property such that the current node is always greater than it left child and right child.
- Keep iterating the array A[0…n-1] from k=0 to k=n-1
- Left and right child of the current element can be calculated by left child can be calculated by A[(i+1)*(i+2)/2 +j] and right child by A[(i+1)*(i+2)/2 +j] where i is floor(-0.5 + sqrt(0.25 + 2 * k)) and j= k-(i*(i+1)/2)
- Replace the current element with the max(left child, right child) if current element is smaller.
  Time complexity = O(n).Because iterating an array of size will n will take O(n) complexity.

Question 3:
Solution 3

a) **Recurrence relation:** T(N) = 1 + Sum j = 1 to N-1 (T(j))

Let A[0..n-1] be the input array and L(i) be the length of the LongestIncreasingLengths ending at index i such that A[i] is the last element of the LongestIncreasingLengths.

```
L(i) = max{v1,1 +  L(j)}  where 0 < j < i and A[j] < A[i];
or
L(i) = 1, if no such j exists.
```

b)***Procedure LongestIncreasingLengths(A[0……n-1])*** where
Input = Array of size n from index 0 to n-1
Output = longest increasing length

```
procedure LongestIncreasingLengths (A[0….n-1])
    n ← size of array A;
    if(n == 0)  then return 1; // Only one subsequence ends at first index, the number itself
    ans ← 1;
    for(i ← 1 to i ← n-1 ) do:
        if(A[i-0] < A[n-1])
            //Recursively get all LIL ending with A[0], A[1] ...A[n-2].
            //Replace ans with maximum of ans and 1+LIL(A[i…n-1]) where i is from 1 to n-1
                ans = max(ans, 1 + LongestIncreasingLengths(A[i…n-1]));
    return ans ;
```

c)**Time Complexity:** O(2^N)
The time complexity is *exponential*. There will be 2^n - 1 nodes will be generated for a n sized array. The complexity of this approach is *O(2^n)* because an array of size *n* contains 2^*n* subsets. For example *[1,2,3]* contains subsets *{}*, *{1}*, *{2}*, *{3}*, *{1,2}*, *{1,3}*, *{2,3}*, *{1,2,3}*.

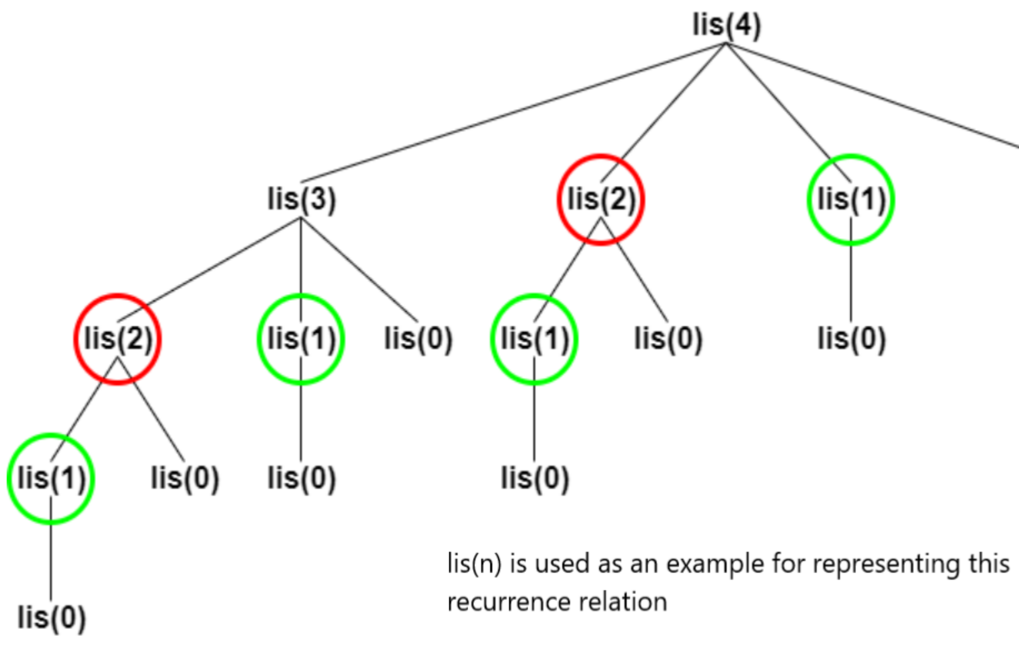
Question 3 b)
Solution 3b)

i)

```
procedure LongestIncreasingLengths(A[0…n-1])
    list,p ← array of size n;
    max,maxIndex,k ← 0;
    initialize each element in list with 1; //initialise each element of array ‘list’ with 1
    initialize each element in p with -1; //initialise each element of array ‘p’ with -1
    for ( i ← 1 to i ← n-1 ) do:
        for ( j ← 0 to j ← i-1 ) do:
            if ( A[i] > A[j] && list[i] < list[j] + 1)
                list[i] ← list[j] + 1;
                p[i]=j;
    for ( i ← 0 to i ← n-1) do:        //Pick maximum of all LIS values
        if ( max < list[i] )
            max ← list[i];      // max will contain the size of longest increasing array
            maxIndex ← i;
    //This array will contain longest increasing array in reverse order
    create array sequence of size max;
    while maxIndex != -1:
        sequence[k] ← A[maxIndex];
        maxIndex ← p[maxIndex];
        k++;
    sequence ← sequence.reverse(); // reversing array will give the longest increasing array
    return sequence,sequence.size();
```

Here variable **max** will contain the size of longest increasing length and array **sequence** will contain the array having longest increasing length.

ii)



As we can clearly see in the recursion tree, there are overlapping subproblems and also holds an optimal substructure property. The recurrence tree we can see that sub problems are solved again and again marked in red and green.

iii)

The complexity of this algorithm is O(n^2) As we can see the basic operation is the comparison if (A[i] > A[j] && list[i] < list[j] + 1) which iterates from i <- 0 to i <- n-1 and also the inner loop of j iterates from j <- 0 to j<- i-1.And i has max value as n-1.As we iterate through each element of Array A with index i and find the longest increasing array till i for that we again iterate from j=0 to j=i-1. We traverse the array once in the outer loop, yielding a complexity of O(n), and then for every element i, we make a linear search of all elements j from 1 up to i. For calculating LIL[i] value for each i in range of N, we traverse the LIL[ ] array from j=0 to j<i.
N iterations are done in the first loop and 1, 2, 3, 4,....N iterations are done for each i in the first loop.
Thus the complexity is N∗(1+2+3+4...N)=O(N^2).

Question 3c)
Solution 3 c)

i)

Set of arrays satisfying given 3 conditions :

A[0] = 0. Rule 1. There are no active lists, create one.
[0]

A[1] = 8. Rule 2. Clone and extend.
[0]
[0, 8]

A[2] = 4. Rule 3. Clone, extend and discard.
[0]
[0,4]
~~0, 8~~. Discarded

A[3] = 12. Rule 2. Clone and extend.
0.
0, 4.
0, 4, 12.

A[4] = 2. Rule 3. Clone, extend and discard.
0.
0, 2.
~~0, 4~~. Discarded.
0, 4, 12.

A[5] = 10. Rule 3. Clone, extend and discard.
[0]
[0, 2]
[0, 2, 10]
~~0, 4, 12~~. Discarded.

A[6] = 6. Rule 3. Clone, extend and discard.
[0]
[0, 2]
[0, 2, 6]
~~0, 2, 10~~. Discarded.

A[7] = 14. Rule 2. Clone and extend.
[0]
[0, 2]
[0, 2, 6]
[0, 2, 6, 14]

A[8] = 1. Rule 3. Clone, extend and discard.
[0]
[0, 1]

~~0, 2~~. Discarded.
[0, 2, 6]
[0, 2, 6, 14]

A[9] = 9. Rule 3. Clone, extend and discard.
[0]
[0, 1]
[0, 2, 6]
[0, 2, 6, 9]
~~0, 2, 6, 14~~. Discarded.

A[10] = 5. Rule 3. Clone, extend and discard.
[0]
[0, 1]
[0, 1, 5]
~~0, 2, 6~~. Discarded.
[0, 2, 6, 9]

A[11] = 13. Rule 2. Clone and extend.
[0]
[0, 1]
[0, 1, 5]
[0, 2, 6, 9]
[0, 2, 6, 9, 13]

A[12] = 3. Rule 3. Clone, extend and discard.
[0]
[0, 1]
[0, 1, 3]
~~0, 1, 5~~. Discarded.
[0, 2, 6, 9]
[0, 2, 6, 9, 13]

A[13] = 11. Rule 3. Clone, extend and discard.
[0]
[0, 1]
[0, 1, 3]
[0, 2, 6, 9]
[0, 2, 6, 9, 11]
~~0, 2, 6, 9, 13~~. Discarded.

A[14] = 7. Rule 3. Clone, extend and discard.
[0]
[0, 1]
[0, 1, 3]
[0, 1, 3, 7]
~~0, 2, 6, 9~~. Discarded.
[0, 2, 6, 9, 11]

A[15] = 15. Rule 2. Clone and extend.
[0]
[0, 1]

[0, 1, 3]
[0, 1, 3, 7]
[0, 2, 6, 9, 11]
0, 2, 6, 9, 11, 15. <-- LIS List
Longest Increasing Length = 6
----

ii)

We can use an auxiliary array to keep end elements. The maximum length of this array is that of input. In the worst case the array divided into N lists of size one .To discard an element, we will trace ceil value of A[i] in auxiliary array (again observe the end elements in your rough work), and replace ceil value with A[i]. We extend a list by adding element to auxiliary array. We also maintain a counter to keep track of auxiliary array length.

- Create an array ‘longestLength’ of size n which is size of array

- Initialise longestLength first element to 0

  (longestLength[0] ← 0)
- Keep a check on length of longestLength array by maintaining a variable (len ← 0)Iterate through array A from i ← 0 to i← n-1 and check following three conditions.

- If A[i] is less than the first element of longestLength array(A[i]<longestLength[0]) then there is a possibility that this is the first element of longest increasing length and hence longestLength[0]=A[i].

- Else if A[i] is greater than the last element of array longestLength (A[i]>longestLength[len-1]) then keep A[len++] ← A[i] .

- Else search element A[i] in array using binary search (BinarySearch(A,-1,len-1,A[i])) which will return ‘index’ of the nearest element greater than A[i] and replace longestLength[index] with A[i]. (longestLength[(BinarySearch(A,-1,len-1,A[i])=A[i]).