# Assignment 4

## Answer to the question no.1

Bellman equation={opt[i]=max(opt[j])+1}

```
int longestPath() {
  for (int v = 1; v <= V; ++v) opt[v] = -1;
  opt[1] = 0;
  for (int v = 1; v <= V; ++v) {
    if (opt[v] >= 0) {
     for_each( v' can be reached from v)
       opt[v'] = max(opt[v]+1, opt[v']);
   }
 }
 return opt[V];
```

Each edge is visited at most one so, the run time is bounded by the number of edges.

## Answer to the question no.3

```
Activity(activity, start, finish){

//Sort activity by finish times

result={activity[1]}

Size=activity.length;

j = 1;

    for i = 2 to size:

       if start[i] ≥ finish[j]:

          result = result U {Activity[i]};

          j = i

return result;

}

Lec-scheduling(S){

            hallcount = 1;

            while( S.length() != 0){
```

```
            Sh = Activity-Selection( S, s , f );
        }

        If( Sh  != NULL ){
            reserve other lecHall for Sd;
        }
        hallcount++;
        S= S – Sh ;

}
```

Analysis:

Two classes with different times can be assigned to the same room. First activity is assigned the first lecture hall and the remaining are passed to the Activity function call to determine the maximum number of compatible classes. Running time $O(n^2)$ in the worst case as the function Activity runs in O(n) time and the loop in function Lec-scheduling runs in O(n) time.

## Answer to the question no.4

```
Opt_BST( r, a , b){

        If ( a==1 && b == r.rows)
            Print kroot[1,b]
If ( a>b )
            return
        x = r[ a, b];

        if( a != x)
            print kroot[ a , x-1 ];

        else{
            print dx-1
```

```
}

Opt_BST( r , a , x-1 );

If ( b !=x )
        Print kroot[ x +1, b ];

else
        Print dx

Opt_BSt( r , x+1, b );
Return;

}
```

Probability sum table:

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 7 | 1 | 0.90 | 0.84 | 0.64 | 0.56 | 0.41 | 0.24 | 0.05 |
| 6 | 0.81 | 0.71 | 0.59 | 0.40 | 0.37 | 0.22 | 0.05 | |
| 5 | 0.64 | 0.54 | 0.42 | 0.28 | 0.20 | 0.05 | | |
| 4 | 0.49 | 0.39 | 0.27 | 0.13 | 0.05 | | | |
| 3 | 0.42 | 0.32 | 0.20 | 0.06 | | | | |
| 2 | 0.28 | 0.18 | 0.06 | | | | | |
| 1 | 0.16 | 0.06 | | | | | | |
| 0 | 0.06 | | | | | | | |

Root table:

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 7 | 5 | 4 | 5 | 6 | 6 | 7 | 7 |
| 6 | 3 | 4 | 5 | 5 | 6 | 6 | |
| 5 | 3 | 2 | 4 | 5 | 5 | | |
| 4 | 2 | 2 | 3 | 4 | | | |
| 3 | 2 | 2 | 3 | | | | |
| 2 | 1 | 2 | | | | | |
| 1 | 1 | | | | | | |

Search cost table e:

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 7 | 3.12 | 2.61 | 2.13 | 1.55 | 1.20 | 0.78 | 0.34 | 0.05 |
| 6 | 2.44 | 1.96 | 1.48 | 1.01 | 0.72 | 0.32 | 0.05 | |
| 5 | 1.83 | 1.41 | 1.04 | 0.57 | 0.30 | 0.05 | | |
| 4 | 1.34 | 0.93 | 0.57 | 0.24 | 0.05 | | | |
| 3 | 1.02 | 0.68 | 0.32 | 0.06 | | | | |
| 2 | 0.62 | 0.30 | 0.06 | | | | | |
| 1 | 0.28 | 0.06 | | | | | | |
| 0 | 0.06 | | | | | | | |

## Answer to the question no.5

Consider the set m={v1/w1,v2/w2,…vn/wn} be the ratio of valuable load/weight.

Choose an element k from set m, using quicksort-like median selection algorithm.

Case 1: M1={vj/wj if vj/wj>k} then W1=$\sum_{jEM1}$Wj

Case 2: M2={vj/wj if vj/wj=k} then W2=$\sum_{jEM2}$Wj

Case 3: M3={vj/wj if vj/wj<k} then W3=$\sum_{jEM3}$Wj

If(W1>W){

return recursive case 1 and return the computed solution;

}

Else{

While(blank space in knapsack and m2 still have some element)

Then add some items from the M2.

If(Knapsack is full){

Return the items in case 1 and the item just added from the case M2.

}

Else

W1+W2

Recursion on the case 3 and then return the items in M1 UM2 and items returned from the recursive call

}

//Repeat the above steps until found the optimum knapsack result.

Recursive the fraction knapsack 0-1 problem. Apply the quicksort like median algorithm to selection the median element. Three cases of recursion and selection algorithm creates half size of knapsack problem. The problem solved in linear time by recurrence T(n)=T(n/2)+BIGTHETA(n), whose solution is T(n)=O(n).

## Answer to the questions no.6

**Part A:**

Operations cost and cost is given below:

1 =1

2=2

3=1

4=4

5=1

6=1

7=1

8=8

Total cost will be = $1+2^1+1+2^2+\ldots..$

$=1+2^1+1+2^2+\ldots..+(n\text{-terms}) <= 1*n+2^1+2^2+2^3 <= n+\sum_{j=0}^{floor(\lg n)} 2^j$

$\sum^{n}_{i=1} c_i <= n+\sum_{j=0}^{floor(\lg n)} 2^j$

$\sum_{j=0}^{floor(\lg n)} 2^j$ is a geometric series so,

$\sum_{j=0}^{floor(\lg n)} 2^j = \sum_{j=0}^{floor(\lg n)} 2^j - 1$

$\sum^{n}_{i=1} c_i < n+\sum_{j=0}^{floor(\lg n)} 2^j$

$=\sum_{j=0}^{floor(\lg n)} 2^j - 1 < n+2n-1$

$\sum^{n}_{i=1} c_i < 3n$

By the aggregate analysis, the worst case for sequence of n operations are O(3n)=O(n).

**Amortized cost per each operation is, O(n)/n=O(1).**

**Part B:**

The accounting method assumes amortized costs for a given operation with different costs than actual cost. The cost can be less or more than the actual operation. The amount for each operation is called amortized cost. The actual cost will be the sum of the credit which is accumulated and the amortized cost of its own which is one unit.

$C_i = (1+2+4+8+\ldots.+2^n) + (1+3+7+\ldots..(2^{n-1}-1))$

$=(2^n-1)+(\sum^{n-1}_{k=0} 2^k - 1)$

$=(2^n-1)+((2^n-1)-(n-1))$

$=(2^n-1)+(2^{n-1}-n)$

$=(2^n -1 +2^{n-1}-n)=3(2^{n-1}) – (n+1)$

Value of $C_i^{\wedge}$ is as follows:

$C_i^{\wedge}= (2+2+2+2+\ldots\ldots+2) + 3(1+3+7+ \ldots. (2^{n-1}-1))$

$=2^n + 3(\sum^{n-1k}_{=0} (2^{k-1})) = 3(2^{n-1})+(2^n-3n)$

Since, the value of $(2^n-3n)$ is always greater tan 1 for n=0 and for all n>1 therefore, the value of $C_i\hat{}$ will always be greater than $c_i$ for n=0 and n>1. For n=1, the value of $C_i\hat{}$ is 2 and the value of $C_i$ is 1.

Thus, the value of $C_i$ is always less than $C_i\hat{}$.

Therefore, as per the accounting analysis, amortized cost is always greater than the actual cost which can be **represented as $C_i=O(C_i\hat{})$.**

**Part C:**

Define the potential function as follows:

$\Phi(D_o)=0$ and $\Phi(D_1)=1$

For n operations the actual cost is as follows:

$\sum^{n}_{i=0} C_i <= n+\sum^{floor(\log n)}_{j=0} 2^j <= n+(1+2+4+..+2^{floor(\log n)})$

Since $2^{(\log n)} \sim n$, so the geometric progression sum is (2n-1) <= n+(2n-1) = 3n-1.

If n is not an exact power of 2 then the amortized cost, for n operations is:

$\sum^{n}_{i=0} C_i\hat{} = \sum^{n}_{i=0} C_i + \Phi(D_n)- \Phi(D_0) <= 3n-1+1-0$ **<=3n**

If n is an exact power of 2, then the amortized cost is as follows:

$\sum^{n}_{i=0} C_i\hat{} = \sum^{n}_{i=0} C_i + \Phi(D_n)- \Phi(D_0) <= 3n-1+n-0$ **<=4n**

## Answer to the question no.7

The dynamic multi-set has can grow to accommodate a new element. The set does not have to order the elements. It can be randomly placed. Repetition of elements is allowed.

**INSERT(S,x)**

The insertion is done in dynamic array, at the end of previous element. When there is space available insert the element at the end of the current element. Time complexity is O(n). When there is no space, allocate double size of the previous set and copy the elements of old array to the new array. Time complexity is O(n). Once the size double it does not have to create another array for another new array/2 operations. We use the amortized analysis.

Size of the array is K. K+1 insert operations are needed.

Cost of first k operations=k O(1) and $k+1^{th}$ operation=O(k).

Approximately 2k operations are needed for k insert operations. The time complexity per INSERT operation is O(1) by amortized analysis.

**DELETE-LARGER-HALF(S):**

The first step is to calculate the median using quick select linear time algorithm. Time complexity is O(n). Partition the array into two halves comparing to the median value. One half should be larger than the other half. Delete the larger half. Time complexity is O(1). The chance of such operation after another is very low so, we use amortized analysis.

**Amortized analysis:**

Let there be a sequence of INSERT and DELETE-LARGER-HALF operations such that the total number of operations is m. The Banker's method of amortized analysis is the following:

The Banker's method imposes extra cost on cheaper operations (INSERT) to pay for the costlier operation(DELETE-LARGER-HALF).

Let us say the cost of INSERT be 2 tokens. DELETE-LARGER-HALF, first partitioning along the median is done. This takes cost proportional to the number of elements in the array which is O(n) so each element pays one token for this operation. When we delete the larger half, remaining tokens of deleted elements if distributed to the smaller half. All the elements in the smaller half have 2 tokens. Each element starts with 2 and ends with 2 elements.

Therefore, each operation in the sequence of operations is done in constant time O(1).

There are total of m operations in the sequence. Total time complexity is O(m) by amortized analysis. The elements of the multi-set can be output in linear time which is O(|S|).