Fundamental Algorithms, Greedy Algorithms, HW - 8

Tulsi Jain

1. Activity Selection, DP

Let's S_{ij} is set of activities starting after a_i is finished and completed before a_j starts. Problem is to find set of maximum number of compatible activities using Dynamic Programming. We assume that the activities are sorted in monotonically increasing order of finish time. We are are two activities in one in front and one in last. First activities has " $f_0 = 0$ and last activity has $s_l = \infty$. Lets create a two dimensional matrix c[i,j], it denotes maximum number of compatible activities starting after f_i and finishing before s_j.

$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} \text{ contains nothing} \\ \max_{i<=k<=j} c[i,k] + c[k,j] + 1 & \text{if } S_{ij} \text{ is not Null} \end{cases}$$

Let's c[i, j] denotes the number of maximum set of compatible activities starting after a_i is finished and complete before a_j starts.
When i == j then c[i, j] = 0 and when j = i + 1 then c[i, j] = 0 as there is no activity starts after a_i is finished and finishes a_j is starts.

Algorithm 1 Activity Selection

```
function ActivitySelection(s, f, n)    ▷ where s startTime, f finishTime, n countActivities
    int[] c = new int[n + 2] [n + 2];
    int[] activities = new int[n + 2] [n + 2];
    for i = 0 to n do
        c[i, i] = 0;
        c[i, i + 1 ] = 0;
    end for
    c[n + 1 , n + 1 ] = 0;
    for i = 0 to n - 1 do
        for j = i + 2 to n + 1 do
            c[i, j] = 0;
            k = j - 1;
            while f[i] < f[k] do
                if f[i] <= s[k] And f[k] <= s[j] And c[i, j] < c[i, k] + c[k, j] + 1 ; then
                    c[i, j] = c[i, k] + c[k, j] + 1
                    activities[i, j] = k
                end if
                k = k - 1;
            end while
        end for
    end for
    return c, activities;
end function
```

Maximum number of compatible activities are c[0, n + 1]. Running time of this algorithms in O(n^3)

Algorithm 2 ActivitySubSet

```
function ActivitySelection(c, activities, 0, n + 1) ▷ where c numberOfComActi, activities, i starting , j ending
    if c[i, j] > 0 then
        print act[i, j]
        ActivitySubSet[c, activities, i, k]
        ActivitySubSet[c, activities, k, j]
    end if
end function
```

2. Greedy Approach

Instead of picking the first finish activity we are picking last activity to start, a_m. If this is last activity to start and this is part of optimal solution we are keeping maximum possible number of resources for other activities. If they are multiple activities with same last start time then we can pick any of them.
Let S_k is set to activities finishes before s_k starts.
S_k = { a_i in S : s_f =< s_k }
Now, we are left with only one problem to solve in S_k. This shows that approach is greedy as we just picking what is best for given a state and solving the left-out only one of problem. But we need to show that, s_l activity lies in some optimum solution. **Some** is for as their can be multiple optimum solution.
Let A_k is a set of maximum possible mutually compatible set of activities and let a_j be the last activity to start in A_k. if $a_j == a_m$ our work is done. As we have to show a_j lies in some optimal solution and it is shown. if a_j != a_m, then $A'_k = A_k - a_j + a_m$ be A_k but substituting a_m for a_j. $|A_k|$ and $|A'_k|$ are equal as we removed one activity and added one. The activities in A'_k are disjoint, which follows because the activities in A_k are disjoint, a_j is the last activity in A_k to start, and $a_j <= s_m$. Hence A'_k is a maximum-size subset of mutually compatible activities of S_k, and it includes a_m.

3. Example

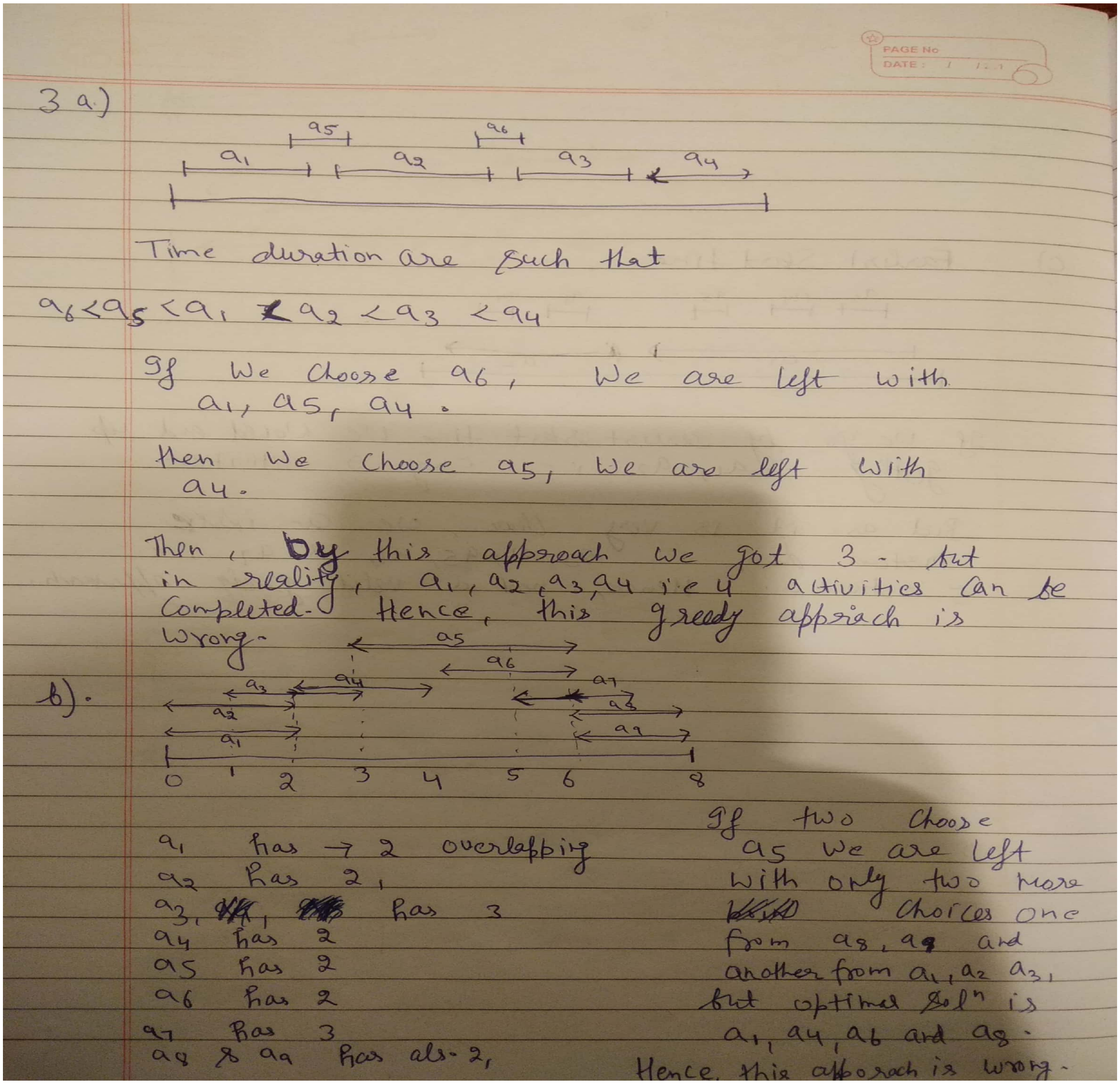
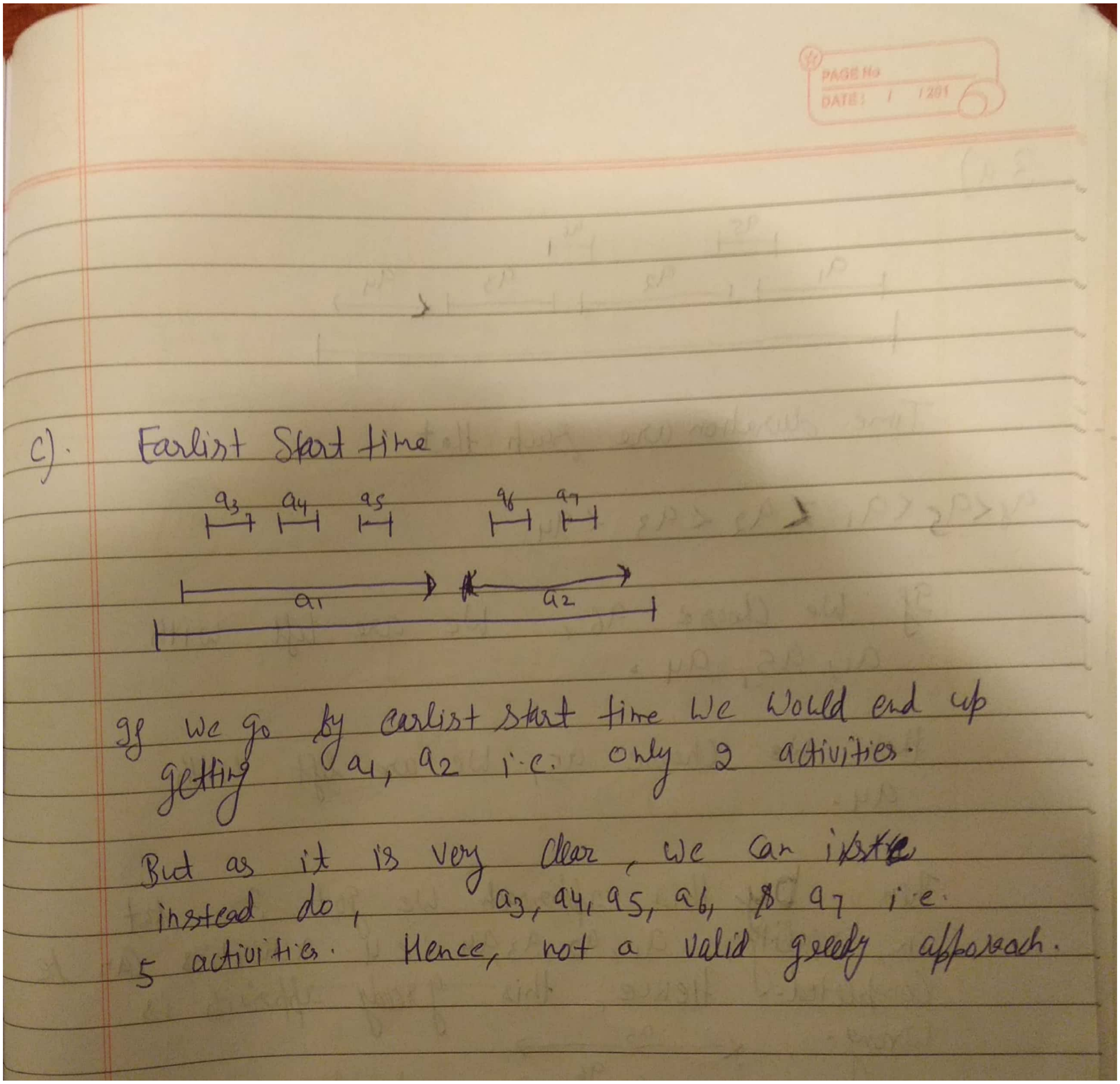


Figure 3.1: 3.a and 3.b



c). Earlist Start time

a_3 a_4 a_5 a_6 a_7

a_1 a_2

If we go by earlist start time we would end up getting a_1, a_2 i.e. only 2 activities.

But as it is very clear, we can ~~inste~~ instead do, $a_3, a_4, a_5, a_6,$ & a_7 i.e. 5 activities. Hence, not a valid greedy approach.

Figure 3.2: 3.c

4. Coin Change

4.1 Greedy

Lets consider we need change for n cents.
Greedy Approach:
let say s_k is largest possible coin such that n - s_k is >= 0. We replace n with n - s_k. **Step 1** We repeat the step until n becomes zero. This algorithms is would execute finite number of times as n is finite so provides a solution as coins contains penny.
Now, lets argue whether proposed solution is a optimum solution or not. Out approach chooses largest possible coin for which n - s_k is >= 0. If optimum solution contains largest possible coin for which s_k is >= 0. We are done otherwise lets consider different cases.
Case 1: 1 <= n < 5, Largest possible coin is 1 and is part of optimum solution.
Case 2: 5 <= n < 10, Largest possible coin is 5 and is part of optimum solution.
Case 3: 10 <= n < 25, Largest possible coin is 10 and is part of optimum solution.
Case 4: 25 <= n Largest possible coin is 25 and is part of optimum solution.
Hence this approach provide the greedy choice and remaining sub problem.
Running time of this Algorithms is $\Theta(k)$, where k is number of coins used. We know $k <= n$, hence running time is O(n).

4.2 Dynamic Programming

Algorithm 3 CoinCount

```
function CoinCount(c, n, k)        ▷ where c coins, k coinsNumber, n changeRequired
    int[] minCoins = new int[n + 1];
    int[] coinsType = new int[n + 1];
    minCoins[0] = 0;
    coinType[0] = 0;
    For all i form 0 to k-1 count[0][i] = 0;
    int i = 1;
    while i <= n do
        int j = 1;
        min = i;
        while j <= k  and  c[j] <= i do
            temp1 = i - c[j];
            temp2 = 1 + minCoins[temp1];
            if temp2 > min then
                min = temp2;
                coinsType[i] = c[j];
            end if
            j = j + 1;
        end while
        minCoins[i] = min;
        i = i + 1;
    end while
    return minCoins[n], coinType;
end function
```

Algorithm 4 CoinUsed

```
function CoinUsed(c, n, k)        ▷ where w coinType, n changeRequired
    int[] coinsused;
    while n > 0 do
        coinsused.add( n - w[n]);
        n = n - w[n];
    end while
    return coinsused;
end function
```

Running time of complete algo is O(nk) as asked in question. Coin Count running time is O(nk) and CoinUsed running time is linear.