CSOR 4231: Analysis of Algorithms I, Problem Set 4

Jing Qian - jq2282

November 9, 2018

Problem 1

We could use Greedy algorithm to solve this problem. The code is like following:

G = $\emptyset$
g = $-\infty$
for i = 1 to n:
    if (x[i] > g + 1) then g = x[i] + 1; G = G $\cup$ g
return G

It is an O(n) algorithm.

We could use the induction method to prove the optimality. For the base case, $n = 1$, one guard is optimal. If the solution is optimal for the (i-1) case with (j-1) guards and the (j-1)-th guard could not cover the i-th painting, then we must add one guard and the location of the j-th guard at x[i] +1 is able to cover the i-th painting while is also closest to the (j+1, ..., n)-th paintings. In other words, the i case is also optimal. So this algorithm is optimal.

Problem 2

a.

Greedy algorithm: select a task with the shortest processing time.

The algorithm is:
Sort S in increasing p_i.
sumC = 0
c = 0
for i = 1 to n:
    c = c + p[i]
    sumC = sumC + c
return sumC/n

Sort tasks according to their shortest processing time: $O(n \log n)$. Greedy select: $O(n)$. So the total running time: $O(n \log n)$.

For the base case, $n = 1$, it is optimal. If there is an optimal solution S' whose first task is b while the task with the shortest processing time is a. Then if we switch the position of a and b, the completion time of the tasks after a will not change while the sum of the processing time between a and b decreases . So the average completion time would decrease which shows the greedy algorithm gives optimal solution.

b.

Because here each task cannot start until its release time, we do some modification on the greedy algorithm in Part a. Here we sort S according to their earliest release time. Then for each release time, we run the task with the shortest remaining processing time. If at release time r_i, a new released task has shorter processing time than the remaining processing time of the current running task, we preempt the current task and run the new one. If the running task finishes before the next release time, we run the released task with the shortest remaining process time. Similar to Part a, this algorithm is optimal.

Sorting tasks takes $O(n \log n)$. Greedy select: $O(n^2)$ because for each release time (from 1 to n) we have to find the available task with the shortest remaining processing time. So the total running time: $O(n^2)$.

Problem 3

1.

Here we denote the starting time of activity a_i as s_i, finishing time as f_i.
(i) $Q_1 = \{a[1]\}$, $Q_2 = \{a[2], a[3]\}$. $w(Q_1) = 10$ while $w(Q_2) = w_2 + w_3 = 11 > w(Q_1)$.

i	s[i]	f[i]	w[i]
1	1	8	10
2	1	3	5
3	4	7	6

Selecting an actitivy of largest weight (here Q_1) is not optimal.

(ii) $Q_1 = \{a[1]\}$, $Q_2 = \{a[2], a[3]\}$. $w(Q_1) = 10$ while $w(Q_2) = w_2 + w_3 = 7 < w(Q_1)$.

i	s[i]	f[i]	w[i]
1	1	7	10
2	1	3	3
3	4	8	4

Selecting an actitivy of earliest finishing time (here Q_2) is not optimal.

2.

Here we use dynamic programming where q_i are the compatible activities with activity i.
Sort S in increasing finishing time.

```
for i = 1 to n:
    wQ[i] = 0

def maxW (i)
    if i ≠ 0:
        wQ[i] = max(w_i + maxW(q_i), maxW(i-1))
    return wQ[i]

sol = ∅
def output (i)
    if w_i + maxW(q_i) > maxW(i-1):
        sol = sol ∪ a_i
        output(q_i)
    else:
        output(i − 1)
```

Sorting takes $O(n \log n)$ time, initializing wQ takes $O(n)$ time while finding the optimal solution takes $O(n)$ time. Output solution takes $O(n)$ time. So total time is $O(n \log n)$.

Problem 4

1.

$$\begin{aligned} \alpha(v) &= \max[\beta(v), w(v) + \sum_{c_i \in C(v)} \beta(c_i)], \\ \beta(v) &= \sum_{c_i \in C(v)} \alpha(c_i). \end{aligned} \tag{1}$$

2.

The algorithm is based on the recurrence of $\alpha(v)$ and $\beta(v)$ in Part 1.

```
sol = ∅
dict = {}

def DP (root)
    beta = 0
    for i in C (root):
        if i in dict:
            beta = beta + dict[i]
        else:
            beta = beta + DP (i)
```

```
alpha = w(v)
for i in C (root):
    for j in C (i):
        if j in dict:
            alpha = alpha + dict[j]
        else:
            alpha = alpha + DP (j)
if alpha > beta:
    sol = sol ∪ root
    dict[root] = alpha
    return alpha
else:
    dict[root] = beta
    return beta
```

Here we build a look-up table "dict" to deposit subproblem we have already solved and it takes constant time to look up the nodes. So we only calculate the maximum weight independent set problem for each node once, and the total time is O(n). And the solution is saved in "sol".

Problem 5

Let $P(x, y)$ be the maximum profit for the cloth with dimension $x \times y$, then there are three possibilities: no cut, cut through the X-dimension and cut through the Y-dimension:

$$P(x,y) = \max[p(x,y), \max_{1\le i<x}[P(i,y) + P(x-i,y) - y], \max_{1\le j<y}[P(x,j) + P(x,y-j) - x]] \quad (2)$$

where $p[x][y]$ is the price for the cloth $x \times y$. If $x \times y$ or $y \times x$ is the j-th in the n possible types, it has a price $p(x, y) = p_j$. Other wise, $p(x, y) = 0$.

The algorithm is as following:

```
for i = 1 to X:
    for j = 1 to Y:
        p(i, j) = 0
for i = 1 to n:
    p(a_i, b_i) = p_i
    p(b_i, a_i) = p_i

def CUT-CLOTH(X, Y)
    for i = 1 to X:
        for j = 1 to Y:
            hCut = 0
            for k = 1 to i-1:
                hCut = max(hCut, CUT-CLOTH(k, j) + CUT-CLOTH(i-k, j) - j)
```

```
            vCut = 0
            for k = 1 to j-1:
                vCut = max(vCut, CUT-CLOTH(i, k) + CUT-CLOTH(i, j-k) - i)
            p(i, j) = max(p(i, j), hCut, vCut)
    return p(i, j)
```

The running time for p(i, j) = 0 is O(X·Y), the running time for "for i = 1 to n" is O(n), and the running time for the CUT-CLOTH function is O(X·Y·(X+Y)). So the total running time is O(X·Y·(X+Y) + n).