CSORE 4231 ANALYSIS OF ALGORITHMS I Homework 3

Francis Zhang xz3279

March 4, 2024

1. Modified Log Cutting

(a) CLRS 15.1-3 Prove correctness and runtime of your algorithm.

(b) CLRS 15.3-5

Solution:

(a) :

Algorithm 1 Modified-Cut-Rod(p, n, c)

```
1: Let r[0, ..., n] be a new array
2: r[0] = 0
3: for j = 1 to n do
4:     q = p[j]
5:     for i = 1 to j − 1 do
6:         q = max(q, p[i] + r[j − i] − c)
7:     r[j] = q
8: return r[n]
```

Prove the loop invariant of Modified-Cut-Rod(p, n, c).
Initialization: At the start of the first iteration of the loop of lines 3-7 we have $j = 1$, $q = p[1]$. For $i = 1$ to $1 - 1 = 0$, there's no need to do the internal loop. Now the optimal revenue of cutting length 1 rod is $r[1] = q = p[1]$, which is correct.
Maintenance: Suppose at the start of the k^{th} iteration of the loop of lines 3-7. $j = k$, $q = p[k]$. Go through $i = 1$ to $k - 1$. First time $q = max(p[k], p[1] + r[k - 1] - c)$, we compare which one is optimal revenue form no cut to k-long rod and cut the rod to length 1 and $k - 1$. Second time is $q = max(q, p[2] + r[k - 2] - c)$. This time the q on the RHS comes form the last try. If no cut to the rod is better, than q is still $p[k]$, if dividing to 1 and $k - 1$ is better than q now is $p[1] + r[k - 1] - c$. Third time... Always keep the larger one for the next loop's comparison. We can finally find out the best revenue within no cut, cut to 1 and $k - 1$, cut to 2 and $k - 2$,..., cut to $k - 1$ and 1. After the loop we then have the optimal revenue of k-long rod, assign it to $r[k]$.
Termination: When $j = n$, we find no cut's revenue, different cut into two pieces revenues. And got the optimal $r[n]$.
Consider the worst case of Nested loops, the runtime should be $O(n^2)$.

(b) : When going through the loop, we start from 1 and divide to two pieces. While one piece directly reflects p[i], the other part is calculated from the iterations done before, and we already assume this part has the optimal-substructure. However as l_i is set here, l_i needs to be satisfied. For instance, if we have $l_1 = l_2 = l_3 = l_4 = 1$, for a length 4 rod. The result can no longer be "$2 + 2$" as $l_2 = 1$.

2. Breaking a string

CLRS 15-9
Solution:

Algorithm 2 Min-Cost-Breaks-With-Sequence(n, L)

Let L be the array of breakpoints with added virtual breakpoints at 0 and n, sorted.
Let $m =$ length of L.
Initialize a 2D array $dp[1..m][1..m]$ with ∞.
Initialize a 2D array $solution[1..m][1..m]$ with $None$.
for $i = 1$ to m **do**
    $dp[i][i] = 0$
for $length = 2$ to m **do**
    for $i = 1$ to $m - length + 1$ **do**
        $j = i + length - 1$
        for $k = i$ to $j - 1$ **do**
            $cost = L[j] - L[i-1] + dp[i][k] + dp[k+1][j]$
            if $cost < dp[i][j]$ **then**
                $dp[i][j] = cost$
                $solution[i][j] = k$
function RECONSTRUCT-SOLUTION(i, j)
    if $i \geq j$ **then**
        return []
    $k = solution[i][j]$
    return RECONSTRUCT-SOLUTION(i, k) + $[L[k]]$ + RECONSTRUCT-SOLUTION$(k+1, j)$
sequence of breaks = RECONSTRUCT-SOLUTION$(1, m-1)$
return $dp[1][m-1]$, sequence of breaks

The runtime is $O(m^3)$ in worst case, because of 3 level nested loop.

3. Inventory planning

CLRS 15-11
Solution: Our subproblems are indexed by an integer $k \in [1, n]$ and another integer $s \in [0, D]$, where k represents how many months are left until the end of the planning horizon, and s indicates how many machines we have in stock at the beginning of month k. Thus, we consider the future demand starting from month k to month n, represented as $(d_k, \ldots, d_n)$.

The variable s will represent the inventory level at the start of month k. For each subproblem indexed by k and s, we will try producing all possible numbers of machines from 0 to $D - s$, the remaining capacity for storage.

Since the index space for our subproblems has size $O(nD)$ and we are iterating through at most D options to find the minimum cost for each subproblem, the total runtime of our algorithm will be $O(nD^2)$.

Algorithm 3 Inventory Planning for Rinky Dink Company

Require: n (months), m (full-time capacity), c (overtime cost per machine), d (demand array of length n), $h(j)$ (holding cost function)
Ensure: Plan that minimizes manufacturing and holding costs

Let $D = \sum_{i=1}^{n} d_i$ (total demand over n months)
Let $cost[1 \dots n][0 \dots D]$ and $make[1 \dots n][0 \dots D]$ be new tables
for $s = 0$ to D **do**
    $cost[n][s] = c \cdot \max(d_n - s - m, 0) + h(s + \max(d_n - s, 0))$
    $make[n][s] = \max(d_n - s, 0)$
$U = d_n$
for $k = n - 1$ downto 1 **do**
    $U = U + d_k$
    for $s = 0$ to D **do**
        $cost[k][s] = \infty$
        for $f = \max(d_k - s, 0)$ to $U - s$ **do**
            $val = cost[k + 1][s + f - d_k] + c \cdot \max(f - m, 0) + h(s + f - d_k)$
            if $val < cost[k][s]$ **then**
                $cost[k][s] = val$
                $make[k][s] = f$
Print the total cost from $cost[1][0]$
Call PRINT-PLAN($make, n, d$) to print the manufacturing plan
function PRINT-PLAN($make, n, d$)
    $s = 0$
    for $k = 1$ to n **do**
        Print "For month k, manufacture $make[k][s]$ machines"
        $s = s + make[k][s] - d_k$

4. Don't be Greedy

(a) CLRS 16.1-1 Prove correctness and runtime of your algorithm.

(b) CLRS 16.1-2

(c) CLRS 16.1-3

Solution: (a)

Algorithm 4 Dynamic Activity Selector

Require: Sorted arrays $s[0 \dots n]$ and $f[0 \dots n]$ of start and finish times
Ensure: The maximum-size subset of mutually compatible activities

```
1: Let c[0...n+1][0...n+1] and act[0...n+1][0...n+1] be new tables
2: for i = 0 to n do
3:     c[i][i] = 0
4:     c[i][i+1] = 0
5: c[n+1][n+1] = 0
6: for l = 2 to n+1 do
7:     for i = 0 to n − l + 1 do
8:         j = i + l
9:         c[i][j] = 0
10:        k = j − 1
11:        while f[i] < s[k] do
12:            if f[i] ≤ s[k] and f[k] ≤ s[j] and c[i][k] + c[k][j] + 1 > c[i][j] then
13:                c[i][j] = c[i][k] + c[k][j] + 1
14:                act[i][j] = k
15:            k = k − 1
16: Print "A maximum size set of mutually compatible activities has size c[0][n+1]."
17: Print "The set contains:"
18: PRINT-ACTIVITIES(c, act, 0, n+1)
```

Algorithm 5 Print Activities

```
1: function PRINT-ACTIVITIES(c, act, i, j)
2:     if c[i][j] > 0 then
3:         k = act[i][j]
4:         Print k
5:         PRINT-ACTIVITIES(c, act, i, k)
6:         PRINT-ACTIVITIES(c, act, k, j)
```

Proof:
Loop Invariant: At the start of each iteration of the loop from lines 7 to 15, $c[i][j]$ represents the maximum number of mutually compatible activities between activity i and activity j, where activities are indexed from 0 to $n+1$ with 0 and $n+1$ acting as sentinel activities. For all p and q where $0 \leq p < i$ and $j < q \leq n+1$, the table $c[p][q]$ will already contain the maximum number of mutually compatible activities for the subproblem defined by activities p through q.
Initialization: Before the first iteration of the loop from lines 7 to 15, $c[i][i]$ and $c[i][i+1]$ are initialized to 0 for all i from 0 to n. This is correct because there are no activities between i and itself, and no activities can occur between i and the immediate next activity $i+1$.
Maintenance: The loop from lines 7 to 15 correctly maintains the invariant by updating $c[i][j]$ to the maximum number of mutually compatible activities between activities i and j including the newly considered activity k. It does this by ensuring that $f[i] \leq s[k]$ and $f[k] \leq s[j]$ to maintain mutual compatibility and by

selecting k such that $c[i][k] + c[k][j] + 1$ is maximized.
Termination: At the termination of the loop from lines 7 to 15, the invariant provides us with the correct value of $c[0][n+1]$, which represents the maximum size set of mutually compatible activities for the entire set of activities from 1 to n.
GREEDY-ACTIVITY-SELECTOR runs in $\Theta(n)$ time and DYNAMIC-ACTIVITY-SELECTOR runs in $O(n^3)$ time. As the algorithm has an 3 level nested loop.

(b) This becomes exactly the same as the original problem if we imagine time running in reverse, so it produces an optimal solution for essentially the same reasons. It is greedy because we make the best looking choice at each step.

(c) As a counterexample to the optimality of greedily selecting the shortest, suppose our activity times are {(1, 9), (8, 11), (10, 20)}. Then, picking the shortest first, we have to eliminate the other two, where if we picked the other two instead, we would have two tasks not one.
As a counterexample to the optimality of greedily selecting the task that conflicts with the fewest remaining activities, suppose the activity times are {(-1, 1), (2, 5), (0, 3), (0, 3), (0, 3), (4, 7), (6, 9), (8, 11), (8, 11), (8, 11), (10, 12)}. Then, by this greedy strategy, we would first pick (4, 7) since it only has a two conflicts. However, doing so would mean that we would not be able to pick the only optimal solution of (-1, 1),(2, 5),(6, 9),(10, 12).
As a counterexample to the optimality of greedily selecting the earliest start times, suppose our activity times are {(1, 10), (2, 3), (4, 5)}. If we pick the earliest start time, we will only have a single activity,(1, 10), whereas the optimal solution would be to pick the two other activities.

5. Of Theives and Professors

(a) CLRS 16.2-2 Prove correctness and runtime of your algorithm.

(b) CLRS 16.2-4

Solution: (a)

Algorithm 6 0-1 Knapsack Problem

Require: n (number of items), W (maximum weight), items i have i.weight and i.value
Ensure: Maximum value that can be achieved with weight not exceeding W

```
1:  Initialize an (n+1) × (W+1) table K with all values as 0
2:  for i = 1 to n do
3:      K[i, 0] = 0
4:  for j = 1 to W do
5:      K[0, j] = 0
6:  for i = 1 to n do
7:      for j = 1 to W do
8:          if j < i.weight then
9:              K[i, j] = K[i − 1, j]
10:         else
11:             K[i, j] = max(K[i − 1, j], K[i − 1, j − i.weight] + i.value)
12: return K[n, W]
```

Invariant for the outer loop (over items i): At the start of each iteration of the outer loop (line 6), $K[i-1][j]$ for all $0 \leq j \leq W$ correctly represents the maximum value achievable using the first $i-1$ items and with a knapsack capacity of j.

Invariant for the inner loop (over weights j): At the start of each iteration of the inner loop (line 7), $K[i][j-1]$ correctly represents the maximum value achievable using the first i items and with a knapsack

capacity of $j - 1$.

Initialization: Before the first iteration of the inner loop, $K[i][0] = 0$ for all i, which means no items can be added to the knapsack when the capacity is 0. Similarly, $K[0][j] = 0$ for all j, which means no value can be achieved without any items.

Maintenance: Each iteration of the inner loop (lines 8-12) updates $K[i][j]$ to reflect the maximum value achievable with the first i items and a knapsack of capacity j. This update is based on whether including the i-th item is beneficial or not, maintaining the loop invariant.

Termination: The outer loop terminates when $i = n + 1$, at which point $K[n][j]$ for all $0 \leq j \leq W$ represents the maximum value achievable with all n items for each knapsack capacity j. The inner loop terminates when $j = W + 1$, ensuring $K[i][W]$ is correctly computed for the current item i.

By maintaining these invariants, the algorithm correctly computes the maximum value achievable within the knapsack's weight limit, with $K[n][W]$ providing the solution to the problem. The runtime is in $O(nW)$.

(b)

The greedy solution solves the problem of maximizing the distance covered from a particular point, under the constraint that there must exist a place to get water before running out, optimally. Specifically, the first stop is chosen at the furthest point from the starting position which is less than or equal to m miles away. This problem exhibits optimal substructure; given a first stopping point p, the subproblem assumes starting at p and solving for the maximum distance from there. Combining the plan from the starting point to p with the plan from p onwards yields an optimal solution for the original problem.

Let $O = \{o_1, o_2, \ldots, o_k\}$ be any optimal solution with stops at positions $o_1, o_2, \ldots, o_k$. Let g_1 denote the furthest stopping point that can be reached from the starting point within the water constraint. We propose a modified solution G by replacing o_1 with g_1, assuming without loss of generality that $o_2 - o_1 < o_2 - g_1$. This implies that it is feasible to reach the subsequent stops in G without running out of water. Since G has the same number of stops as O, and we have shown that reaching the stops in G is feasible, we conclude that g_1 is contained in some optimal solution. Thus, the greedy strategy of selecting the furthest reachable stopping point as the first stop is validated.