Solutions of Introduction to Algorithms

Saman Saadi

Contents

Chapter 1

Dynamic Programming

1.1 Rod cutting

1.1.1 Exercise 2

No it cannot always produce an optimal solution. Consider the following example.

l_i	1	2	3
p_i	1	50	72
$\frac{p_i}{l_i}$	1	25	24

For n = 3 the greedy approach cut the rod in 2 pieces. The length of one of them is 2 and the other's is 1. So the profit is $50\$ + 1\$ = 51\$$. But the optimal solution is to keep the rod intact so the profit is 72\$.

1.1.2 Exercise 3

We can keep the rod intact so we don't need to incur the fixed cost c or we can have at least one cut. We need to choose the best solution among all of them:

$$r(i) = \begin{cases} \max\limits_{1 \le k < n} (p_i, r(i-k) + p_k - c) & i > 0 \\ 0 & i = 0 \end{cases}$$

So the solution is $r(n)$. We have n distinct subproblem. In each step we need to choose between keeping the rod intact or have at least one cut which divide the rod into two pieces. The length of one of them is k and the other's $n - k$. We don't know the exact value of k so we need to try all possible values. This can be done in $O(n)$. Therefore the overall running time is $O(n^2)$

```
1: function F(p, n, c)
2:     let r[0..n] be a new array
3:     r[0] ← 0
4:     for j from 1 to n do
5:         q ← p[j]
6:         for i from 1 to j − 1 do
7:             q = max(q, r[j − i] + p[i] − c)
8:         end for
9:         r[j] = q
10:    end for
11:    return r[n]
12: end function
```

1.2 Matrix-chain multiplication

1.2.1 Exercise 4

I've used the following equations:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \tag{1.1}$$

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \tag{1.2}$$

Each node of the graph represents a distinct sub-problem. Suppose we have two nodes v and u. There is an edge from v to u, if the solution of subproblem v is depended on subproblem u. In other words, there is an edge from m[i, j] to all m[i, k] and m[k + 1, j] for $i \leq k < j$.
Usually $|V|$ determines space complexity and $|V| + |E|$ time complexity. we know for every subprobelm m[i, j], $j \geq i$. Hence we have $n - i + 1$ subproblems which starts with A_i. So the number of vertices is:

$$\begin{aligned} |V| &= \sum_{i=1}^{n} n - i + 1 \\ &= \sum_{i=1}^{n} i \\ &= \frac{n(n+1)}{2} \end{aligned} \tag{1.3}$$

Hence the space complexity is is $O(n^2)$. We don't use all of the array cells when $j < i$. So we waste $\frac{n^2-n}{2}$ of allocated array. By analyzing lines 5 - 10 of MATRIX-CHAIN-ORDER pseudocode in the text book we can compute the number of edges. As you can see in line 10, m[i, j] is depends on two subproblem m[i, k] and m[k + 1, j]. We visit each distinct subproblem exactly once. So by

```
5   for l = 2 to n                    // l is the chain length
6       for i = 1 to n − l + 1
7           j = i + l − 1
8           m[i, j] = ∞
9           for k = i to j − 1
10              q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
```

counting the outdegree of each node we can calculate the number of edges in a

directed graph:

$$
\begin{aligned}
|E| &= \sum_{l=2}^{n} \sum_{i=1}^{n-l+1} \sum_{k=i}^{i+l-2} 2 \\
&= \sum_{l=2}^{n} \sum_{i=1}^{n-l+1} 2(l-1) \\
&= 2\sum_{l=2}^{n} (n-l+1)(l-1) \\
&= 2\sum_{l=2}^{n} (n-(l-1))(l-1) \\
&= 2\sum_{l=1}^{n-1} (n-l)l \\
&= 2(\sum_{l=1}^{n-1} nl - \sum_{l=1}^{n-1} l^2) \\
&= 2(n\sum_{l=1}^{n-1} l - \sum_{l=1}^{n-1} l^2) \\
&= 2[n\frac{(n-1)n}{2} - \frac{(n-1)(n)(2n-1)}{6}] \\
&= n^2(n-1) - \frac{n(n-1)(2n-1)}{3} \\
&= \frac{3n^2(n-1) - n(n-1)(2n-1)}{3} \\
&= \frac{n(n-1)(3n-2n+1)}{3} \\
&= \frac{n(n-1)(n+1)}{3} \\
&= \frac{n(n^2-1)}{3} \\
&= \frac{n^3-n}{3}
\end{aligned}
\tag{1.4}
$$

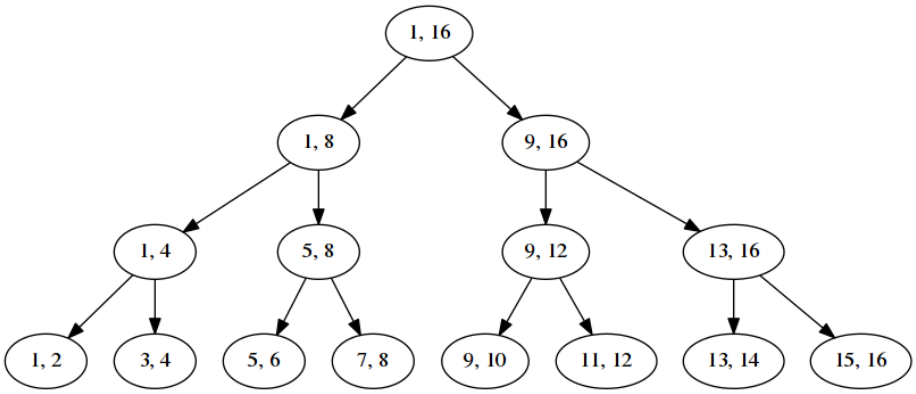
So the running time is $|V| + |E| = \frac{n^2+n}{2} + \frac{n^3-n}{3} = O(n^3)$

1.3 Elements of dynamic programming

1.3.1 Exercise 2

Each node is filled with (p, r). p is the index of leftmost element and r is the index of rightmost element of array which the subprolem wants to sort. As you

can see there is no overlapping between subproblems so dynamic programming is not a good idea for merge sort. In other words, we don't see a previously solved subproblem again and we only waste memory. As a general rule if the subproblem graph is a tree, dynamic programming cannot be applied.



Chapter 2

Amortized Analysis

2.1 Aggregate analysis

2.1.1 Exercise 1

No it doesn't hold. The maximum number of pops, including multipop, is proportional to the number of previous push operations. If we can only push one item, the number of pushed elements is at most n. If we add a new operation named multipush, then the number of pushed items is at most $n \times k$. So the amortized cost is $O(k)$. For example we can have two operations. One is multipushing 10^9 items and the other is multipopping 10^9 items. It is obvious the total cost is not $O(n) = O(2)$ and is $O(nk) = O(2 \times 10^9)$.

2.1.2 Exercise 2

The following pseudo-code explains how to implement DECREMENT. The

```
1: function DECREMENT(A)
2:     i = 0
3:     while i < A.length and A[i] == 0 do
4:         A[i] = 1
5:         i = i + 1
6:     end while
7:     if i < A.length then
8:         A[i] = 0
9:     end if
10: end function
```

worst case happens when we start with 0 and then decrements it to get $2^k - 1$ which all bits are set to 1 and then increments it to get 0. We repeat this loop

until we have n operations. For $n = 4$ and $k = 3$ we have:

000
111
000
111