Solutions of Introduction to Algorithms

Saman Saadi

Contents

Chapter 1

Dynamic Programming

1.1 Rod cutting

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$.

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 $O(n^2)$

```
function F(p, n, c)
    let r[0..n] be a new array
    r[0] ← 0
    for j from 1 to n do
        q ← p[j]
        for i from 1 to j − 1 do
            q = max(q, r[j − i] + p[i] − c)
        end for
        r[j] = q
    end for
    return r[n]
end function
```

1.2 Matrix-chain multiplication

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

```
for l = 2 to n                    // l is the chain length
    for i = 1 to n - l + 1
        j = i + l - 1
        m[i, j] = ∞
        for k = i to j - 1
            q = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j
```

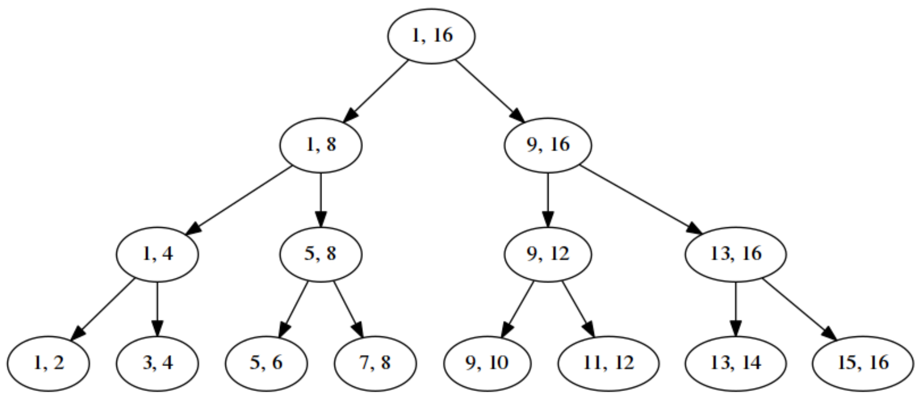
directed graph:

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

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

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

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 $O(nk) = O(2 \times 10^9)$.

Exercise 2 The following pseudo-code explains how to implement DECREMENT. The worst case happens when we start with 0 and then decrements it

```
function DECREMENT(A)
    i = 0
    while i < A.length and A[i] == 0 do
        A[i] = 1
        i = i + 1
    end while
    if i < A.length then
        A[i] = 0
    end if
end function
```

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:

$$\begin{array}{c} 000 \\ 111 \\ 000 \\ 111 \end{array}$$

2.2 Accountant method

For another example of "accountant method" see exercise 5 of 3.1.

Chapter 3

Elementary Graph Algorithms

3.1 Representation of graphs

Exercise 1 We know that $adj[u]$ is a list. Depends on the list implementation, it can take $O(1)$ to determine its size. In that case the running time for finding the out-degree of each vertex is $O(V)$. If we cannot determine size of the list in $O(1)$, then the overall running time of algorithm is $O(V + E)$.
The running time for finding in-degree of each vertex is $O(V + E)$.

Exercise 3 For adjacency-matrix it takes $O(V^2)$ and for adjacency-list it takes $O(V + E)$.

Algorithm 1 G' using adjacency matrix

```
function TransposeGraph(G)
    Let G' be a new graph
    G' ← G
    for all u ∈ V do
        for all v ∈ V do
            G'.A[v][u] = G.A[u][v]
        end for
    end for
    return G'
end function
```

Algorithm 2 G' using adjacency list

```
1: function TransposeGraph(G)
2:     Let G' be a new graph
3:     G'.V = G.V
4:     for all u ∈ G.V do
5:         for all v ∈ G.Adj[u] do
6:             G'.Adj[v].insert(u)
7:         end for
8:     end for
9: end function
```

Exercise 4 We create a new adjacency-list for G' called adj. For each vertex u in G, suppose v is its neighbor. If $u \neq v$, then $adj[u].insert(v)$ and $adj[v].insert(u)$. If there are multiple edges between u and v, we see v as u's neighbor more than once. So if the last element if $adj[v]$ is u, it means there are more than one edges between them so we shouldn't insert v again. Traversing G takes $O(V+E)$. Finding out there are more than one edge between two vertices is $O(1)$. So the overall running time is $O(V+E)$. Note that I supposed G is also undirected.

```
1: function F(G)
2:     let G' be a new graph
3:     G'.V = G.V
4:     for all u ∈ G.V do
5:         for all v ∈ G.adj[u] do
6:             if u ≠ v ∧ G'.adj[v].last() ≠ u then
7:                 G'.adj[v].insert(u)
8:             end if
9:         end for
10:    end for
11:    return G'
12: end function
```

Exercise 5 The running time of matrix-list implementation is $O(V^3)$. For analyzing the running time of adjacency-list implementation we can use amortized analysis. We use "accountant method".

> $\boldsymbol{in_u}$ **:** The number of edges that enter u
>
> $\boldsymbol{out_u}$ **:** The number of edges that leave u
>
> $\boldsymbol{e_u}$ **:** And edge from u to an arbitrary vertex $v \neq u$

We assign to all e_u cost $c_{e_u} = 1 + in_u$. Because by traversing the graph, we visit e_u at least once (line 6). For each edge that enters u we visit or revisit e_u

(lines 7 - 8). We know that $\sum_{u=1}^{|V|} in_u + \sum_{u=1}^{|V|} out_u = 2|E|$. So we can easily calculate the total cost.

$$\begin{aligned}\sum_{e_u \in E} c_{e_u} &= \sum_{e_u \in E} 1 + in_u \\ &= \sum_{e_u \in E} 1 + \sum_{u=1}^{|V|} in_u \\ &= |E| + \sum_{u=1}^{|V|} in_u \\ &\leq 3|E|\end{aligned}$$

We execute line 6 at most $|E|$ times and lines 7 - 8 at most $2|E|$ times. So the total running time of algorithm using adjacency-list is $O(|V|+3|E|) = O(V+E)$.

Algorithm 3 Finding square graph using matrix-list

```
1: function MakeSquareGraph(G)
2:     Let G' be a new Graph                          ▷ G.A[1..|V|, 1..|V|]
3:     for all u ∈ G.V do
4:         for all v ∈ G.V do
5:             G'.A[u][v] = G.A[u][v]                 ▷ 1-edge paths
6:             if G.A[u][v] = 1 then
7:                 for all k ∈ G.V do
8:                     G'.A[u][k] = G.A[v][k]         ▷ 2-edge paths
9:                 end for
10:            end if
11:        end for
12:    end for
13: end function
```

Algorithm 4 Finding square graph using adjacency-list

```
function MakeSquareGraph(G)
    Let G' be a new graph
    G'.V = G.V
    for all u ∈ G.V do
        for all v ∈ G.Adj[u] do
            G'.Adj[u].insert(v)                ▷ 1-edge paths
            for all w ∈ G.Adj[v] do
                G'.Adj[u].insert(w)            ▷ 2-edge paths
            end for
        end for
    end for
end function
```

Exercise 6 Suppose A is an adjacency matrix for G.

$$A[i,j] = \begin{cases} 1 & \text{i cannot be a universal sink} \\ 0 & \text{j cannot be a universal sink} \end{cases}$$

The following algorithm find the universal sink in $O(V)$. In each step we remove one vertex from all candidates for "universal sink". It takes $O(V)$ to have only one candidate. To determine that candidate is indeed a universal sink we need $O(2V)$ operations. So the overall running time of algorithm is $O(V) + O(2V) = O(V)$.

```
function getUniversalSink(G)
    A = G.A                                  ▷ A[1..|V|, 1..|V|]
    u ← 1
    while u ≤ |V| do
        v ← u + 1
        sink ← u        ▷ Vertices from sink to |V| can be universal sink
        while v ≤ |V| ∧ A[u, v] = 0 do
            v ← v + 1                      ▷ v cannot be a universal sink
        end while
        u ← v                      ▷ u to v − 1 cannot be a universal sink
    end while
    for c from 1 to sink − 1 do
        if A[sink, c] ≠ 0 then
            return "No universal sink"
        end if
    end for
    for r ∈ V − {sink} do
        if A[r, sink] ≠ 1 then
            return "No universal sink"
        end if
    end for
    return sink
end function
```

Exercise 7 We know that B is an $V \times E$ matrix which we show it as $B_{V\times E}$. By definition B^T is an $E \times V$ matrix which we show it as $B^T_{E\times V}$. We define $P_{V\times V} = B_{V\times E} \times B^T_{E\times V}$.

$$p[i,j] = \sum_{k=1}^{E} b[i,k] \times b^T[k,j]$$

We consider two cases.

1. $\boldsymbol{i \neq j}$ **:** It is impossible that both $b[i,k]$ and $b[k,j]$ have the value of "1". Because the k$^{\text{th}}$ edge cannot enter both vertices i and j. With the same argument we can prove that both of them cannot have value of "-1". If the k$^{\text{th}}$ edge connect i to j, then $b[i,k] = -1$ and $b^T[k,j] = 1$. Otherwise both have value of zero. In other words, for $i \neq j$ the value of $p[i,j]$ is the number of edges between i and j.

2. $\boldsymbol{i = j}$ **:** It is obvious both $b[i,k]$ and $b[k,i]$ should have the same value. In this case $p[i,i]$ is the sum of all edges that enter and leave the vertex i.

$$p[i,j] = \begin{cases} \text{number of edges between i and j} & i \neq j \\ indegree(i) + outdegree(i) & i = j \end{cases}$$