The Australian National University
School of Computing

Semester 2, 2024
Tutorial 10

Convenor(s): Yun Kuen Cheung (Marco) & Ahad N. Zehmakan

Concrete version: Sunday 6th October, 2024, 19:04 (Compilation date and time)

Algorithms (COMP3600/6466)

Problems marked with (∗) are challenge exercises. They will not be discussed in tutorials, and solutions will not be released. Once confident in your solution for a challenge exercise, you are welcome to discuss it with your tutor during the consultation period in the tutorial session, or schedule a time with Marco to present it.

Exercise 1 — *Basics of Dynamic Programming (DP)*

1. Let $M[i]$ denote the i-th *Marconacci* number, defined as below. $M[0] = 0$, $M[1] = 1$, $M[2] = 4$. For $n \geq 3$,
$$M[n] \;=\; (n-2) \cdot M[n-1] + M[n-2] + (2n+1) \cdot M[n-3] \;.$$
Write down pseudocode, or code in your favorite PL, that implements a DP algorithm to compute $M[n]$ for a given n in $\mathcal{O}(n)$ time.

Solution

```
def marconacci(n):
    if n ≤2:
        return pow(n,2)

    M = [0 for i in range(n+1)]
    M[1] = 1
    M[2] = 4
    for i in range(3,n + 1):
        M[i] = (i-2) * M[i − 1] + M[i − 2] + (2*i+1) * M[i − 3]
    return M[n]
```

2. The running time of a divide-and-conquer algorithm is given by the following recurrence. $T[0] = T[1] = 1$, and for $n \geq 2$,
$$T[n] \;=\; T\left[\left\lfloor \frac{n}{2} \right\rfloor\right] + T\left[\left\lceil \frac{n}{2} \right\rceil\right] + (3n-4) \;.$$
Write down pseudocode, or code in your favorite PL, that implements a DP algorithm to compute $T[n]$ for a given n in $\mathcal{O}(n)$ time.

Solution

```
import math
def runtime(n):
    if n ≤1:
        return 1

    T = [1 for i in range(n+1)]
    for i in range(2,n + 1):
        T[i] = T[math.floor(i/2)] + T[math.ceil(i/2)] + (3*i-4)
    return T[n]
```

3. Two sequences F, G of numbers are defined as below. $F[0] = 2$, $F[1] = 3$, $G[0] = 5$ and $G[1] = 7$. For $n \geq 2$,
$$F[n] \;=\; \max\{2 \cdot F[n-2], G[n]\} - 1$$
$$G[n] \;=\; F[n-1] + 2 \cdot G[n-1]$$
Write down pseudocode, or code in your favorite PL, that implements a DP algorithm to compute $F[n]$ and $G[n]$ for a given n in $\mathcal{O}(n)$ time.

Solution

Note that $G[n]$ does not depend on $F[n]$, but $F[n]$ depends on $G[n]$. Thus, in each iteration below, we should compute $G[i]$ first.

```
def fg(n):
    if n ==0:
        return 2, 5
    if n ==1:
        return 3, 7

    G = [0 for i in range(n+1)]
    F = [0 for i in range(n+1)]
    G[0] = 5
    G[1] = 7
    F[0] = 2
    F[1] = 3
    for i in range(2,n + 1):
        G[i] = F[i-1] + 2 * G[i-1]
        F[i] = max(2 * F[i-2], G[i]) - 1
    return F[n], G[n]
```

Exercise 2 ***Rod Cutting and Generalizations***

4. We consider the rod cutting problem discussed in the lecture, but now each cut costs \$$x$ dollars, where x is an input parameter of the problem. Suppose the prices of rods of different lengths are given to you.

   (a) Write down a recurrence that can be used to compute the maximum possible revenue when given an original rod of length n. What are the boundary case(s)?

   (b) Write down pseudocode, or code in your favorite PL, that implements a DP algorithm to compute the maximum possible revenue when given an original rod of length n.

   (c) Analyze your algorithm's runtime.

Solution

(a) As in the lecture, to write down a recurrence when given a rod of length n, we first decide the length of the leftmost rod we will cut. If $1 \le \ell \le n-1$, that means a cut does exist, so we need to afford a cost of \$$x$ dollars. But when $\ell = n$, no cut is really made, so we do not need to pay the cutting cost, and the revenue is $p[n]$. Thus, we have the recurrence:

$$v[n] \;=\; \max\left\{ \max_{1\le\ell\le n-1} \{p[\ell] + v[n-\ell] - x\} \;,\; p[n] \right\}$$

(b) The code is given below.

```
def rodcutting_with_cutcost(n, p, x):
    if n <=1:
        return p[n]
    if n >=2:
        v = [0 for i in range(n + 1)]
        v[1] = p[1]
        for k in range(2, n + 1):
            runningmax = p[k]
            for ℓ in range(1, k):
                runningmax = max(runningmax, p[ℓ] + v[k − ℓ] - x)
            v[k] = runningmax
        return v[n]
```

Note the similarity and differences from the rodcutting procedure given in the lecture.

(c) When $n \geq 2$, the initialization of v takes $\mathcal{O}(n)$ time. In the double-for loop, the inner-for loop's running time is $\mathcal{O}(k)$, and there are $\mathcal{O}(1)$ operations for each iteration of the outer-for loop. So the overall running time is $\mathcal{O}(2+3+\ldots+n)+\mathcal{O}(n)=\mathcal{O}(n^2)$.

5. We consider the rod cutting problem discussed in the lecture, but now we are only allowed at most k cuts, where k is an input parameter of the problem. Suppose the prices of rods of different lengths are given to you. Let $v[n,i]$ denote the maximum possible revenue when we are given an original rod of length n, and we are allowed to make at most i cuts to the original rod.

   (a) What is the value of $v[n,0]$ for $n \geq 0$? Explain your answer.

   (b) Write down a recurrence of $v[n,i]$ for $i \geq 1$. (Hint: if you decide the length of the leftmost rod you will cut is ℓ, what is the maximum possible revenue you can obtain from the remaining portion?)

   (c) Write down pseudocode, or code in your favorite PL, that implements a DP algorithm to compute the maximum possible revenue when given an original rod of length n.

   (d) Analyze your algorithm's runtime.

Solution

(a) For computing $v[n,0]$, since no cut is allowed, we can only sell the rod as is, so $v[n,0]=p[n]$.

(b) Let ℓ denote the length of the leftmost rode we will cut. If $\ell=n$, then the revenue is $p[n]$. If $1 \leq \ell \leq n-1$, we consume one quota of cutting, so for the remaining portion of the rode, the maximum possible revenue is $v[n-\ell,i-1]$. Thus, we have the recurrence:

$$v[n,i] \quad = \quad \max\left\{\max_{1\leq \ell \leq n-1}\{p[\ell]+v[n-\ell,i-1]\}\ ,\ p[n]\right\}$$

(c) The code is given below.

```
def rodcutting_with_cutquota(n, p, k):
    if n <=1 or k <=0:
        return p[n]
    if n >=2:
        v = [[0 for q in range(k + 1)] for j in range(n + 1)]

        for j in range(n + 1):
            v[j][0] = p[j]   # boundary case when no cut is allowed; see part (a)

        for i in range(1, k + 1):   # iterate i as the maximum cuts allowed
            v[0][i] = 0
            v[1][i] = p[1]
            for j in range(2, n + 1):   # iterate j as the length of the original rod
                runningmax = p[j]
                for ℓ in range(1, j):
                    runningmax = max(runningmax, p[ℓ] + v[j − ℓ][i − 1])
            v[j][i] = runningmax

        return v[n][k]
```

(d) When $n \geq 2$, the initialization of v takes $\mathcal{O}(nk)$ time. The first for-loop that handles the boundary case takes $\mathcal{O}(n)$ time. The main runtime is from the triple-for loop above. The first for-loop (that iterates i) has $\mathcal{O}(k)$ iterations. The second for-loop (that iterates j) has $\mathcal{O}(n)$ iterations. The third for-loop (that iterates ℓ) has $\mathcal{O}(j)$ iterations, while the maximum value of j iterated by the second for-loop is n. So the overall running time is $\mathcal{O}(nk)+\mathcal{O}(n)+\mathcal{O}(k)\cdot\mathcal{O}(n)\cdot\mathcal{O}(n)=\mathcal{O}(n^2k)$.

6. (*) We consider a 2-dimensional generalization of the rod cutting problem, which we refer to as the "chocolate problem". We are given a piece of rectangular chocolate with integer length n and integer

width. The width is at most 3, but the length n can be arbitrary. The prices of any piece of rectangular chocolate with length ℓ and width w, for $1 \leq \ell \leq n$ and $1 \leq w \leq 3$, are given to you. Present a polynomial-time algorithm (i.e., the runtime is polynomial in n) that computes the maximum possible revenue. Analyze your algorithm's runtime.