CPSC 320 2020S2: Tutorial 8

In the following two problems, the quantities of interest can be defined inductively, that is, using recurrences. See how these definitions can then be used to define efficient algorithms, using either divide and conquer or memoization/dynamic programming.

1 Travel Itinerary

You are the manager of the Abbotsford branch of a large multinational company, and your company just bought out a small business based in Brampton, Ontario. However, your company has not found a manager for the Brampton branch yet, so they have asked you to oversee both branches until the end of the year. There are n days remaining in the year.

Because you will be spending a fair amount of time in Brampton, you have determined that it will be more convenient to stay in hotels in both cities, rather than continuing to rent your apartment in Abbotsford. You cannot stay in either location for more than 7 days because you do not want to be away from either branch for too long. You must determine how to split your time between Abbotsford and Brampton in a way that minimizes the total cost of your hotels and flights.

The cost of your flight/hotel depends on the day (weekends/holidays might be more expensive, or there might be promotions on certain days) and the city. Fortunately, you are given four arrays that provide the cost of the hotels and flights for each day.

The arrays f_A, f_B, h_A, and h_B, each of length n, are defined as follows:

$f_A[i]$ is the cost of flying *to* Abbotsford on day i

$f_B[i]$ is the cost of flying *to* Brampton on day i

$h_A[i]$ is the cost of staying in a hotel in Abbotsford on day i

$h_B[i]$ is the cost of staying in a hotel in Brampton on day i

You are looking to minimize the cost of accommodations plus flights, subject to the constraint that you cannot spend more than 7 days in one location. You can assume that your flights are always in the morning (so if you are flying between cities on day i, you only have to pay for a hotel in your destination city that day).

Let $C_A[i]$ be the lowest possible value of the objective function for days 1 to i, assuming you are staying in Abbotsford on day i. Similarly, let $C_B[i]$ be the lowest possible value of the objective function for days 1 to i, assuming you are staying in Brampton on day i. In either case, assume that you are staying in Abbotsford on day 0.

1. Find recurrence relations for $C_A[i]$ and $C_B[i]$, for all $i, 1 \leq i \leq n$. You can express the recurrence for $C_A[i]$ in terms of $C_B[i]$, and vice versa.

   SOLUTION:

$$C_A[i] = \begin{cases} \min_{1 \le x \le 7} \left(C_B[i-x] + f_A[i-x+1] + \sum_{j=i-x+1}^{i} h_A(j) \right) & i \ge 8 \\ \min \left(\min_{1 \le x < i} \left(C_B[i-x] + f_A[i-x+1] + \sum_{j=i-x+1}^{i} h_A(j) \right), \ \sum_{j=1}^{i} h_A(j) \right), & 1 < i < 8 \\ h_A(1) & i = 1 \\ 0 & i = 0 \end{cases}$$

$$C_B[i] = \begin{cases} \min_{1 \le x \le \min(7,i)} \left(C_A[i-x] + f_B[i-x+1] + \sum_{j=i-x+1}^{i} h_B(j) \right) & i > 1 \\ f_B(1) + h_B(1) & i = 1 \\ 0 & i = 0 \end{cases}$$

2 Exponentiation

Given two non-negative integers x and n and a positive integer N, the exponentiation problem is to compute $x^n \bmod N$. For example, if $x = 3, n = 4$, and $N = 15$ then $x^n = 3^4 = 81$ and $81 \bmod 15 = 6$.

1. Fill in the missing parts to describe $x^n \bmod N$ inductively:

   SOLUTION:

$$x^n \bmod N = \begin{cases} 1 \bmod N, & \text{if } n = 0 \\ x \bmod N, & \text{if } n = 1 \\ (x^{n/2} \bmod N)^2 \bmod N, & \text{if } n > 1 \text{ is even} \\ ((x^{(n-1)/2} \bmod N)^2 \times x) \bmod N, & \text{if } n > 1 \text{ is odd.} \end{cases}$$

2. Using this inductive definition, write a recursive algorithm that computes $x^n \bmod N$. You can assume that multiplication and mod operations on n-bit numbers are available, that we can test if a number is even, and so on.

   SOLUTION: A divide and conquer method works well here:

```
Algorithm power(x, n, N):
    If n == 0
        Return 1 mod N
    Else If n == 1
        Return x mod N
    Else
        xpowerhalf = power(x, ⌊n/2⌋, N)
        If n mod 2 == 0
            Return xpowerhalf × xpowerhalf mod N
        Else
            Return (xpowerhalf × xpowerhalf ×x) mod N
```