CS 621 - Assignment 3

Ali Hassani

October 30, 2022

1

Given n pairs $(x_0, y_0), (x_1, y_1), ..., (x_{n-1}, y_{n-1})$ we want to find a polynomial $A(x)$ such that, for all i, $A(x_i) = y_i$. As mentioned in class, Lagrange's formula gives us a way to determine A:

$$A(x) = \sum_{k=0}^{n-1} y_k \frac{\prod_{j \neq k} (x - x_j)}{\prod_{j \neq k} (x_k - x_j)}.$$

a. Argue that, for any i $(0 \leq i < n)$, $A(x_i) = y_i$.

b. Show how to compute the coefficient representation of $A(x)$ in $\mathcal{O}(n^2)$ time.

Answer:

a

For every i ($0 \leq i < n$):

$$\begin{aligned}
A(x_i) &= \sum_{k=0}^{n-1} y_k \frac{\prod_{j \neq k} (x_i - x_j)}{\prod_{j \neq k} (x_k - x_j)} \\
&= y_i \frac{\prod_{j \neq k} (x_i - x_j)}{\prod_{j \neq k} (x_i - x_j)} + \sum_{\substack{k=0 \\ k \neq i}}^{n-1} y_k \frac{\prod_{j \neq k} (x_i - x_j)}{\prod_{j \neq k} (x_k - x_j)} \\
&= (y_i)(1) + \sum_{\substack{k=0 \\ k \neq i}}^{n-1} y_k \frac{\prod_{j \neq k} (x_i - x_j)}{\prod_{j \neq k} (x_k - x_j)} \\
&= (y_i)(1) + \sum_{\substack{k=0 \\ k \neq i}}^{n-1} y_k \frac{(x_i - x_i) \prod_{\substack{j \neq k \\ j \neq i}} (x_i - x_j)}{\prod_{j \neq k} (x_k - x_j)} \\
&= (y_i)(1) + \sum_{\substack{k=0 \\ k \neq i}}^{n-1} y_k \frac{(0) \prod_{\substack{j \neq k \\ j \neq i}} (x_i - x_j)}{\prod_{j \neq k} (x_k - x_j)} \\
&= (y_i)(1) + 0 \\
&= y_i
\end{aligned}$$

b

$$\begin{aligned}
A(x) &= \sum_{k=0}^{n-1} y_k \frac{\prod_{j\neq k}(x - x_j)}{\prod_{j\neq k}(x_k - x_j)} \\
&= \sum_{k=0}^{n-1} y_k \frac{(x - x_k)\prod_{j\neq k}(x - x_j)}{(x - x_k)\prod_{j\neq k}(x_k - x_j)} \\
&= \sum_{k=0}^{n-1} y_k \frac{\prod_{j}(x - x_j)}{(x - x_k)\prod_{j\neq k}(x_k - x_j)} \\
&= \big(\prod_{j=0}^{n-1}(x - x_j)\big) \sum_{k=0}^{n-1} \frac{y_k}{(x - x_k)\prod_{j\neq k}(x_k - x_j)}
\end{aligned}$$

We can now pre-compute $\prod_{j=0}^{n-1}(x - x_j)$ with $\mathcal{O}(n^2)$. The summation is a loop over the n pairs, and includes computing the product of $n-1$ subtractions ($\mathcal{O}(n)$), multiplied by a 1-degree polynomial ($\mathcal{O}(1)$), and the division of a scalar, y_k ($\mathcal{O}(1)$). The entire summation is therefore also $\mathcal{O}(n^2)$.

Total runtime: $\mathcal{O}(n^2)$.

2 Chapter 5, Exercise 7

Suppose that you're given an $n \times n$ grid graph G. (An $n \times n$ grid graph is just the adjacency graph of an $n \times n$ chessboard. To be completely precise, it is a graph whose node set is the set of all ordered pairs of natural numbers (i, j), where $1 \leq i \leq n$ and $1 \leq j \leq n$; the nodes (i, j) and (k, ℓ) are joined by an edge if and only if $|i - k| + |j - \ell| = 1$.)

Each node v is labeled by a real number x_v; you may assume that all these labels are distinct. Show how to find a local minimum of G using only $\mathcal{O}(n)$ probes to the nodes of G. (Note that G has n^2 nodes.)

Answer:

We start by taking the middle row and column ($(n+1)/2$-th row and column if n is odd, otherwise either $n/2$-th or $(n+2)/2$-th row and column), which has $\mathcal{O}(n)$ elements ($2n-1$ to be exact). We find the minimum value x_v among these elements ($\mathcal{O}(n)$); labeled o_1. From that point, a maximum of 2 adjacent nodes to o_1 remain unexplored (either its top and bottom or its left and right nodes were already in the list we took the minimum of). Of those remaining adjacent nodes, if neither is smaller than x_{o_1}, then o_1 is a local minimum. Otherwise, we take the sub-grid that contains the smallest of the adjacent nodes. Now the problem size is $n/2$, and we repeat the same process until a local minimum is found. Because the subgrids are bound by the same borders the minimum of which is chosen, we can guarantee that at each step, a local minimum can be found in the sub-grid that is chosen.

Total runtime:

$$\begin{aligned} & \mathcal{O}(n + n/2 + n/4 + ... + 1) \\ = & \mathcal{O}(2n-1) \\ = & \mathcal{O}(n) \end{aligned}$$

3 Chapter 6, Exercise 2

Suppose you're managing a consulting team of expert computer hackers, and each week you have to choose a job for them to undertake. Now, as you can well imagine, the set of possible jobs is divided into those that are low-stress (e.g., setting up a Web site for a class at the local elementary school) and those that are high-stress (e.g., protecting the nation's most valuable secrets, or helping a desperate group of Cornell students finish a project that has something to do with compilers). The basic question, each week, is whether to take on a low-stress job or a high-stress job.

If you select a low-stress job for your team in week i, then you get a revenue of $\ell_i > 0$ dollars; if you select a high-stress job, you get a revenue of $h_i > 0$ dollars. The catch, however, is that in order for the team to take on a high-stress job in week i, it's required that they do no job (of either type) in week $i - 1$; they need a full week of prep time to get ready for the crushing stress level. On the other hand, it's okay for them to take a low-stress job in week i even if they have done a job (of either type) in week $i - 1$.

So, given a sequence of n weeks, a plan is specified by a choice of "low-stress," "high-stress," or "none" for each of the n weeks, with the property that if "high-stress" is chosen for week $i > 1$, then "none" has to be chosen for week $i - 1$. (It's okay to choose a high-stress job in week 1.) The value of the plan is determined in the natural way: for each i, you add ℓ_i to the value if you choose "low-stress" in week i, and you add h_i to the value if you choose "high-stress" in week i. (You add 0 if you choose "none" in week i.)

The problem. Given sets of values $\ell_1, \ell_2, ..., \ell_n$ and $h_1, h_2, ..., h_n$, find a plan of maximum value. (Such a plan will be called optimal.)

Example. Suppose $n = 4$, and the values of ell_i and h_i are given by the following table. Then the plan of maximum value would be to choose "none" in week 1, a high-stress job in week 2, and low-stress jobs in weeks 3 and 4. The value of this plan would be $0 + 50 + 10 + 10 = 70$.

	Week 1	**Week 2**	**Week 3**	**Week 4**
ℓ	10	1	10	10
h	5	50	5	1

a. Show that the following algorithm does not correctly solve this problem, by giving an instance on which it does not return the correct answer.

```
For iterations $i = 1$ to $n$
  If $h_{i+1} > \ell_i + \ell_{i+1}$ then
    Output "Choose no job in week $i$"
    Output "Choose a high-stress job in week $i+1$"
    Continue with iteration $i+2$
  Else
    Output "Choose a low-stress job in week $i$"
    Continue with iteration $i+1$
  Endif
End
```

To avoid problems with overflowing array bounds, we define $h_i = \ell_i = 0$ when $i > n$.

In your example, say what the correct answer is and also what the above algorithm finds.

b. Give an efficient algorithm that takes values for $\ell_1, \ell_2, ..., \ell_n$ and $h_1, h_2, ..., h_n$ and returns the value of an optimal plan.

Answer:

a

The algorithm fails to find the correct answer to this instance:

	Week 1	Week 2	Week 3	Week 4
ℓ	10	10	10	10
h	500	5	5	5

```
For $i = 1$
  $5 < 10 + 10$ then
    Output "Choose a low-stress job in week 1"
    Continue with iteration 2
For $i = 2$
  $5 < 10 + 10$ then
    Output "Choose a low-stress job in week 2"
    Continue with iteration 3
For $i = 3$
  $5 < 10 + 10$ then
    Output "Choose a low-stress job in week 3"
    Continue with iteration 4
For $i = 4$
  $5 < 10 + 0$ then
    Output "Choose a low-stress job in week 4"
```

Final solution: Low stress for all weeks. Profit: 40.
Optimal solution: High stress on week 1, low stress on the rest. Profit: 530.

b

We define $A(i)$ as the solution up to and including week i for all integers $i > 1$, and define $A(0) = 0$ and $A(1) = max(h_1, \ell_1)$.

$$A(i) = \max\big(A(i-1) + \ell_i, A(i-2) + h_i\big).$$

The value for the optimal plan would then be $A(n)$.

Pseudo-code:

```
A := array of $n+1$
A[0] = $0$
A[1] = $max(\ell_1, h_1)$
For iterations $i = 2$ to $n$
  A[i] = $max(A[i-1] + \ell_i, A[i-2] + h_i)$
End
return A[n]
```

Complexity: $\mathcal{O}(n)$ space and time complexity.