# Algorithms M2–IF TD 5

November 10, 2021

## 1 Weighted Independent Set on Paths

(Exercise 6.3 of [DPV]) We are considering opening restaurants along a highway from city A to city B. The possible locations are given to us as an array $D[1 \dots n]$, where $D[i]$ is the distance of location $i$ from A. Each location has an expected profit $P[i]$. We have unlimited budget, however, we do not want to open two restaurants which are at distance at most $k$ kilometers. Given this constraint, describe a polynomial-time (in $n$) algorithm that selects the locations that maximize the total expected profit.

**Solution:**

Let $M[i]$ be the maximum profit we can expect from a solution that is only allowed to use locations $1, \dots, i$. Clearly, $M[1] = P[1]$ and the value we are interested in is $M[n]$. We now claim the following relation:

$$M[i] = \max\{M[i-1], P[i] + \max_{j:D[i]-D[j]\geq k} M[j]\}$$

Indeed, if we are given the option to use location $i$, we have two choices: either we don't use it, so the best we can do is the best we can do using locations $1 \dots i-1$, or we do, which gives us profit $P[i]$ plus the best we can do if we only use locations $j$ such that $D[i] - D[j] \geq k$.

Example: suppose $k = 1$ and all locations are at distance 1 from each other. Let

$$P[] = [3, 2, 5, 4, 3, 7, 5]$$

We have

$$M[] = [3, 3, 8, 8, 11, 15, 16]$$

so the best solution takes locations $1, 3, 5, 7$.

## 2 Max Sum Sub-interval

We are given an array $A[1 \dots n]$ of positive and negative integers. We want to calculate two integers $a, b$ such that $\sum_{i=a}^{b} A[i]$ is maximized.

- Observe that this problem is trivial to solve in $O(n^3)$.
- Give an algorithm that solves this problem in time $O(n)$.

**Solution:**

It is clear that $a, b \in \{1, \ldots, n\}$ so we can try all possible $O(n^2)$ values for $a, b$. For each pair $a, b$, calculating the sum of $A[a \ldots b]$ can be done in $O(n)$ time. So the total running time is $O(n^3)$.

Let us try to find a better algorithm. We first construct an array $S[1 \ldots n]$ defined as $S[i] = \sum_{i=1}^{i} A[i]$. Observe that the obvious way to compute $S$ takes time $O(n^2)$ (we spend $O(n)$ per element), however, we observe that $S[i+1] = S[i]+A[i+1]$. Therefore, $S$ can be computed in time $O(n)$. We now see that what we are looking for is the numbers $a, b$ which maximize $S[b]-S[a-1] = \sum_{i=a}^{b} A[i]$. Clearly, this problem can be solved in $O(n^2)$ (we check all pairs $a, b$).

Let us try to do even better. Define $M[i] = \max_{j<i} \sum_{k=j}^{i} A[k]$. In other words, if we fix $i$ to be the last element of the interval we are looking for, $M[i]$ is the value of the optimal solution we seek. Therefore, the value we want is $\max_{i \in \{1,\ldots,n\}} M[i]$. We would like to show that $M[i]$ can be computed in $O(1)$ time per element.

Indeed, assume that we have calculated $M[i-1]$. Then, $M[i] = A[i] + \max\{M[i-1], 0\}$. To see this observe that the best interval ending at $i$ will either contain nothing else (so it has value $A[i]$), or it must contain the best interval ending at $i-1$.

# 3 Longest Common Subsequence

We are given two strings over the alphabet $\{A, B, C, \ldots, Z\}$. For example, the strings $s_1$ =`ILOVEALGORITHMS` and $s_2$ =`VACANCESDENOEL`. A subsequence of a string $s$ is a string we can obtain by deleting some of the letters of $s$. For example, `LOVE` is a subsequence of $s_1$, `LOLO` is also a subsequence of $s_1$ (it doesn't matter that its letters are not consecutive is $s_1$), but `AGORA` is not a subsequence of $s_1$.

1. Given two strings $s_1, s_2$, give a polynomial-time algorithm which decides if $s_2$ is a subsequence of $s_1$. Prove the correctness of your algorithm.
2. Given two strings $s_1, s_2$, give a polynomial-time algorithm which calculates the length of the longest string $s'$ which is a subsequence of both $s_1, s_2$.

**Solution:**

For the first problem, we treat the two strings as arrays $s_1[1 \ldots n]$ and $s_2[1 \ldots m]$. If $m > n$ then we reply NO and this is clearly correct as $s_2$ cannot be a subsequence of a shorter string. If $m == 0$ (that is, $s_2$ is empty) we answer YES, as the empty string is trivially a subsequence of any string.

If $m \leq n$, then we compare $s_1[1] == s_2[1]$. If $s_1[1] == s_2[1]$ then we call the same algorithm for strings $s_1[2 \ldots n]$ and $s_2[2 \ldots n]$ and return its response. If

$s_1[1]! = s_2[1]$ then we call the same algorithm on $s_1[2 \dots n]$ and $s_2$ and return its response.

This algorithm clearly runs in polynomial time, since each recursive call reduces $n$ by 1 (complexity: $T(n) \leq T(n-1) + O(1) = O(n)$).

Let us prove correctness. For the base cases ($m == 0$ or $m > n$) the algorithm is trivially correct, so let us look at the two recursive cases. Suppose $s_1[1] == s_2[1]$. If the recursive call returns YES, that is, $s_2[2 \dots n]$ is a subsequence of $s_1[2 \dots n]$, we see that $s_2$ is a subsequence of $s_1$. If on the other hand the recursive call returns NO, it cannot be the case that $s_2$ is a subsequence of $s_1$, so our response is correct. Suppose then that $s_1[1] \neq s_2[1]$. Then, the first letter of $s_1$ cannot be used in finding the subsequence $s_2$, so our algorithm correctly discards it.

Let us now discuss the second problem. We define $L[i, j]$ as the length of the longest common subsequence of the strings $s_1[1 \dots i]$ and $s_2[1 \dots j]$. What we are interested in calculating is $L[n, m]$. To simplify presentation we consider the variable ranges $i \in \{0, \dots n\}$ and $j \in \{0, \dots, m\}$. By definition $L[0, j] = L[i, 0] = 0$ for all $i, j$, as the longest common subsequence of any string with the empty string has length 0.

The table $L[i, j]$ has size $O(nm)$. We now need to show how to compute $L[i, j]$ using values $L[i', j']$ where $i' < i, j' < j$. Suppose that $s_1[i] \neq s_2[j]$. Then to obtain a common subsequence we must delete either $s_1[i]$ or $s_2[j]$. Therefore, in this case $L[i, j] = \max\{L[i-1, j], L[i, j-1]\}$. If on the other hand, $s_1[i] = s_2[j]$, then the common subsequence must contain this last letter, so $L[i, j] = 1 + L[i-1, j-1]$. We have:

$$L[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ 1 + L[i-1, j-1] & \text{if } s_1[i] = s_2[j] \\ \max\{L[i-1, j], L[i, j-1]\} & \text{otherwise} \end{cases}$$

Calculating each $L[i, j]$ value takes $O(1)$ time, so the algorithm takes $O(nm)$ time.

# 4 The Gas Station Problem

We want to drive from city A to city B along a highway of length $k$ kilometers. Our car has a tank with a capacity of $L$ liters and we start out with a full tank from city $A$. To simplify things, suppose that our car uses 1 liter per kilometer of driving.

We are given two arrays $D[1 \dots n]$ and $P[1 \dots n]$. The first array contains the positions of gas stations along the way (that is, the distance of each gas station from $A$). The second array has the price of gas for each gas station. So, if $D[i] = d_i$ and $P[i] = p_i$, this means that the $i$-th gas station is at $d_i$ kilometers from $A$ and sells gas for $p_i$ euros per liter.

Give a polynomial-time algorithm which selects a set of gas stations to stop at and the amount of gas to buy at each one in order to minimize the total cost

of driving from $A$ to $B$. You may assume that it's OK to arrive at $B$ with an empty tank. Your algorithm should run in time polynomial in $n, L$.

**Solution:**

To simplify presentation, assume that $D$ is sorted (it gives gas stations in order of increasing distance from $A$), that gas station 0 is in $A$ and that the $n$-th gas station in $D$ is at distance $k$ from $A$ (so the last gas station is in $B$).

We want to calculate the value $C[i, j]$ which represents the minimum cost of driving from gas station $i$ to $B$ if we start off with $j$ liters of gas. We have:

- The value we want to calculate is $C[0, L]$.
- For $C[i, j]$ the variables take values $i \in \{0, \ldots, n\}$ and $j \in \{0, \ldots, L\}$.

We therefore have a table of size $O(nL)$. We now need to describe how to compute $C[i, j]$ assuming we have already computed $C[i', j]$ for $i' > i$.

Suppose that we find ourselves at gas station $i$ (so, $D[i]$ kilometers from $A$), with $j$ liters of gas. If $j \geq k - D[i]$ then we can just drive to $B$, so $C[i, j] = 0$. Otherwise, we can buy $b$ liters of gas from this station (at a cost of $b \cdot P[i]$) and then drive to another gas station $i'$. When we arrive there we will have $j' = j + b - (D[i'] - D[i])$ liters of gas left. The optimal solution from that point will therefore cost $C[i', j']$. We now select the combination of $b, i'$ that minimizes this total cost. In particular:

$$C[i, j] = \min_{b \in \{0,\ldots,L\}} \min_{i' \in \{i+1,\ldots,n\}} (b \cdot P[i] + C[i', j + b - D[i'] + D[i]])$$

Calculating each entry of the table above therefore takes $O(nL)$, so the total complexity is $O(n^2 L^2)$.

# 5 Inventory Problem

You own a company that sells cars. For this exercise we will try to minimize the inventory cost of your company, that is, the cost of storing the cars in a garage.

You have a garage with space for $S$ cars. Storing a car for a week in this garage costs $C$ euros **per car**. When your garage is running low you can order more cars from the factory. This has a delivery cost of $K$, independent of how many cars you ordered.

You are given an array which estimates the expected demand for cars in the next few weeks: the array $D[1 \ldots n]$ has value $D[i]$ for the number of cars you would like to have available in your garage to sell during week $i$.

We want to calculate an ordering and storage schedule which satisfies the following:

- We never store more than $S$ cars in the garage.

- At the beginning of each week $i$, the number of cars in the garage is at least $D[i]$. In other words, if at the beginning of week $i$ the garage contains fewer than $D[i]$ cars, we **must** order more cars (and pay $K$ for the delivery) so that this week's demand is satisfied.
- The total storage cost is minimized.

You may assume that the garage contains $S_0 < S$ cars in the beginning. Your algorithm should run in time polynomial in $n$ and $S$.

**Solution:**

We want to calculate the following value: $A[i, j]$ is defined as the minimum cost of any feasible storage plan which satisfies all demands from week $i$ up to week $n$, if we start with a garage that contains $j$ cars. Note that $i \in \{1, \dots n\}$ and $j \in \{0, \dots, S\}$ so the size of the table $A[i, j]$ is $O(nS)$.

We observe that:

- The value we are interested in is $A[1, S_0]$. We want to satisfy demands from week 1 to $n$ and start with $S_0$ cars.
- The values $A[n, j]$ are easy to compute. Indeed, if $j \geq D[n]$ then $A[n, j] = 0$, otherwise $A[n, j] = K$ (because we have to order cars to satisfy the demand for week $n$).

We now have a base case. Building on this we want to calculate $A[i, j]$ using values $A[i + 1, j']$. Consider now an entry $A[i, j]$ for which $j < D[i]$. In this situation we have to order more cars, but we have to decide how many. Suppose that we decide to order $\ell$ cars, such that $\ell + j \geq D[i]$. Then, we will pay $K$ for delivery, and $(\ell + j - D[i]) \cdot C$ for storage for one week. For the rest of the schedule, we can look up the value $A[i + 1, j + \ell - D[i]]$. On the other hand, if $j \geq D[i]$, we do not have to order more cars but we can still do it. If we order $\ell$ cars our cost is as previously, while if we don't, we pay $(j - D[i]) \cdot C$ for storage, plus the cost for the remaining weeks which is given by $A[i + 1, j - D[i]$. We have

$$A[i, j] = \min_{\ell : j+\ell \geq D[i] \wedge \ell \geq 0} (A[i + 1, j + \ell - D[i]] + C \cdot (j + \ell - D[i]) + K \cdot [\ell > 0])$$

where $[\ell > 0]$ is the value that is 1 if $\ell > 0$ and 0 otherwise. Each entry of this table can be computed in $O(S)$ time, so the algorithm takes $O(nS^2)$ in total.