Week 7 Tutorial Sheet

(Solutions)

Useful advice: The following solutions pertain to the theoretical problems given in the tutorial classes. You are strongly advised to attempt the problems thoroughly before looking at these solutions. Simply reading the solutions without thinking about the problems will rob you of the practice required to be able to solve complicated problems on your own. You will perform poorly on the exam if you simply attempt to memorise solutions to the tutorial problems. Thinking about a problem, even if you do not solve it will greatly increase your understanding of the underlying concepts. Solutions are typically not provided for Python implementation questions. In some cases, pseudocode may be provided where it illustrates a particular useful concept.

Tutorial Problems

Problem 1. (Preparation) Implement the solution to the coin change problem described in the lectures. Your solution should return the number of coins needed, along with how many of each denomination are required.

(a) Use the bottom-up strategy to compute the solutions.

(b) Use the top-down strategy to compute the solutions.

Consult the notes if you are unclear on the difference between the two approaches.

Problem 2. (Preparation) Suppose that you are a door-to-door salesman, selling the latest innovation in vacuum cleaners to less-than-enthusiastic customers. Today, you are planning on selling to some of the n houses along a particular street. You are a master salesman, so for each house, you have already worked out the amount c_i of profit that you will make from the person in house i if you talk to them. Unfortunately, you cannot sell to every house, since if a person's neighbour sees you selling to them, they will hide and not answer the door for you. Therefore, you must select a subset of houses to sell to such that none of them are next to each other, and such that you make the maximum amount of money.

For example, if there are 10 houses and the profits that you can make from each of them are 50, 10, 12, 65, 40, 95, 100, 12, 20, 30, then it is optimal to sell to the houses $1,4,6,8,10$ for a total profit of \$252. Devise a dynamic programming algorithm to solve this problem, i.e. to return the maximum profit you can obtain.

divide and conquer, subproblem does not overlap

(a) Describe in plain English, the optimal substructure present in the problem.

dynamic programing does

(b) Define a set of overlapping subproblems that are based on the optimal substructure.

(c) What are the base case subproblems and what are their values?

(d) Write a recurrence relation that describes the solutions to the subproblems.

(e) Write pseudocode that implements all of this as a dynamic programming algorithm.

Solution

Let the houses on the street be numbered from 1 to n from left to right. We begin by observing that the constraint that we cannot sell to the neighbours of a house is equivalent to the constraint that if we sell to house i, then we cannot sell to house $i-1$ (it is not necessary to explicitly prevent ourselves from selling to house $i+1$, since if we sell to house $i+1$, then they will not allow us to sell to house i).

Suppose that we find ourselves in front of house i, deciding whether we should sell to it. If we do decide to sell to it, then we cannot have sold anything to house $i-1$, but we are allowed to sell to a valid subset of houses from 1 to $i-2$. If we do not decide to sell to house i, then any valid subset of houses from 1 to $i-1$ is acceptable to sell to.

The optimal substructure of the problem can then be observed as follows. Suppose that house i is the last house that we will consider selling to. If we chose to sell to house i, then we cannot sell to house $i-1$, and must therefore sell to the houses $[1..i-2]$, and we must do this in a way that makes us the maximum profit. If we decide not to sell to house i, then we must sell to the houses $[1..i-1]$ in such a way that we make maximum profit.

Let us therefore define our subproblems to be

$$\mathtt{DP}[i] = \{\text{the maximum profit that we can make from selling to a subset of the houses } [1..i]\}$$

for all $1 \le i \le n$. When there is just a single house, we should sell to it, so a suitable base case is $\mathtt{DP}[1] = c_1$. We leverage the optimal substructure to write the recurrence

$$\mathtt{DP}[i] = \max\begin{cases} \mathtt{DP}[i-1], & \text{(if we decide not to sell to house } i\text{)}, \\ \mathtt{DP}[i-2] + c_i, & \text{(if we decide to sell to house } i\text{)} \end{cases}.$$

for all $2 \le i \le n$, where we define $\mathtt{DP}[0] = 0$ (the profit we can make by selling to the empty set of houses). The optimal solution to the problem is the value of $\mathtt{DP}[n]$.

An bottom-up implementation of this algorithm might look like this.

```
function SALESMAN($c[1..n]$)
    Set DP[0..n] = 0
    $\mathtt{DP}[1] = c_1$
    for i = 1 to $n$ do
        $\mathtt{DP}[i] = \max(\mathtt{DP}[i-1], \mathtt{DP}[i-2] + c_i)$
    end for
    return $\mathtt{DP}[n]$
end function
```

Problem 3. Extend your solution to Problem 2 so that it returns an optimal subset of houses to sell to, in addition to the maximum possible profit.

Solution

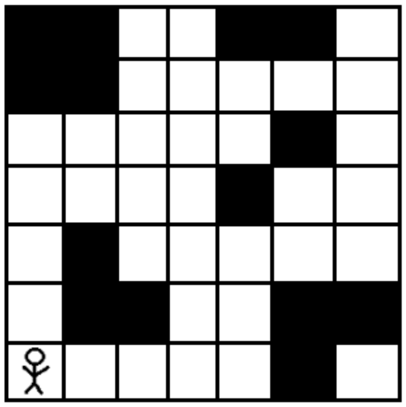
To reconstruct the optimal set of houses, we make the observation that in a range of houses $[1..i]$, house i is optimal to sell to if $\mathtt{DP}[i] > \mathtt{DP}[i-1]$. Why is this true? If $\mathtt{DP}[i] > \mathtt{DP}[i-1]$, then since $\mathtt{DP}[i] = \max(\mathtt{DP}[i-1], \mathtt{DP}[i-2] + c_i) > \mathtt{DP}[i-1]$, it must be true that $\mathtt{DP}[i] = \mathtt{DP}[i-2] + c_i > \mathtt{DP}[i-1]$. In other words, it is more profitable to sell to house i than to not. Using this observation, a simple backtracking procedure could work as follows:

```
function SALESMAN($c[1..n]$)
    Set DP[0..n] = 0
    $\mathtt{DP}[1] = c_1$
    for i = 1 to $n$ do
        $\mathtt{DP}[i] = \max(\mathtt{DP}[i-1], \mathtt{DP}[i-2] + c_i)$
    end for
    Set houses = []
    Set $i = n$
    while $i > 0$ do
        if $\mathtt{DP}[i] > \mathtt{DP}[i-1]$ then
            houses.append($i$)
            $i = i - 2$
        else
```

```
            $i = i - 1$
        end if
    end while
    return DP[$n$], houses
end function
```

Note that we subtract 2 from i if we decide to include house i, so that we cannot accidentally sell to house $i-1$ as well, otherwise we subtract 1.

Problem 4. You find yourself curiously stranded on an $n \times n$ grid, unsure of how you got there, or how to leave. Some of the cells of the grid are blocked and cannot be walked through. Anyway, while you're here, you decide to solve the following problem. You are currently standing at the bottom-left corner of the grid, and are only able to move up (to the next row) and to the right (to the next column). You wonder, how many ways can you walk to the top-right corner of the grid while avoiding blocked cells? You may assume that the bottom-left and top-right cells are not blocked. For example, in the following grid, the answer is 19.



Write a dynamic programming algorithm that given a grid as input, counts the number of valid paths from the bottom-left cell to the top-right cell. Your algorithm should run in $O(n^2)$ time.

Solution

Let's denote the bottom-left corner as cell $(1,1)$ and the top-right corner as cell (n,n). Suppose that you are standing in cell (i,j). In general, you have two choices, to move to cell $(i+1,j)$ or to move to cell $(i,j+1)$. Let p_1 and p_2 denote the number of paths from cell $(i+1,j)$ to cell (n,n) and the number of paths from cell $(i,j+1)$ to cell (n,n) respectively. These paths must all be different, since paths from the first set use cell $(i+1,j)$, and those from the second use $(i,j+1)$, and a valid path cannot contain both of these (think about why this is true). Hence the total number of paths from cell (i,j) to cell (n,n) is just $p_1 + p_2$.

The edge/base cases are if cell (i,j) is blocked, then there are no paths from it, or if you are currently on the top row $(i = n)$ or rightmost column $(j = n)$, in which case you only have one cell that you can move to. Finally, the last base case is that the number of paths from cell (n,n) to cell (n,n) is just 1. Let's write a dynamic programming algorithm along these lines.

Let's denote by DP$[i,j]$, the following subproblems:

$$\text{DP}[i,j] = \{\texttt{The number of valid paths from cell } (i,j) \texttt{ to cell } (n,n).\}$$

for all $1 \le i, j \le n$. Then we can write the following recurrence:

$$\text{DP}[i,j] = \begin{cases} 1 & \text{if } (i,j) = (n,n) \\ 0 & \text{if } (i,j) \text{ is blocked} \\ \text{DP}[i, j+1] & \text{if } i = n \\ \text{DP}[i+1, j] & \text{if } j = n \\ \text{DP}[i+1, j] + \text{DP}[i, j+1] & \text{otherwise} \end{cases}$$

The value of DP[1,1] is the solution. There are a total of n^2 subproblems, and each of them can be evaluated in constant time, hence the time complexity of this algorithm will be $O(n^2)$.

Problem 5. You somehow find yourself on yet another $n \times n$ grid, but this time, it is more exciting. Each cell of the grid has a certain non-negative amount of money on it! Denote the amount of money in the cell of row i, column j by $c_{i,j}$. You are standing on the bottom-left corner $(1,1)$ of the grid. From any cell, you can only move up (to the next row), or right (to the next column). What is the maximum amount of money that you can collect?

Given $c_{i,j}$ for every cell, describe a dynamic programming algorithm to solve this problem. Your algorithm should run in $O(n^2)$ time.

Solution

This problem is very similar to Problem 4. If we are standing in cell (i, j), then we have two choices, to move up to cell $(i+1, j)$ or to move right to cell $(i, j+1)$. We should select the best of the two, whichever leads to a greater amount of money. This motivates the following subproblems:

$$\text{DP}[i,j] = \{\text{The maximum amount of money we can make starting from cell } (i,j)\}$$

The recurrence must take into account the boundary cases (if we are in the top row or rightmost column, we only get one choice) and the base case (if we are in cell (n,n) we are finished and can't move anywhere else). The following recurrence captures these ideas:

$$\text{DP}[i,j] = c_{i,j} + \begin{cases} 0 & \text{if } (i,j) = (n,n) \\ \text{DP}[i+1, j] & \text{if } j = n \\ \text{DP}[i, j+1] & \text{if } i = n \\ \max(\text{DP}[i, j+1], \text{DP}[i+1, j]) & \text{otherwise} \end{cases}$$

The optimal solution is the value of DP[1,1]. There are n^2 subproblems, and each takes constant time to solve, so the solution takes $O(n^2)$ time.

Problem 6. Consider a sequence $a_1, a_2, ..., a_n$ of length n. A *subsequence* of a sequence a is any sequence that can be obtained by deleting any of the elements of a. Devise a dynamic programming algorithm that finds the length of a *longest increasing subsequence* of a. That is, a longest possible subsequence that consists of elements in strictly increasing order. Your algorithm should run in $O(n^2)$ time.

For example, given the sequence $\{\mathbf{0}, 8, 4, 12, \mathbf{2}, 10, \mathbf{6}, 14, 1, \mathbf{9}, 5, 13, 3, \mathbf{11}, 7, \mathbf{15}\}$, the longest increasing subsequence is $\{0, 2, 6, 9, 11, 15\}$ of length 6 (shown in **bold** in the original sequence).

Solution

Consider a particular sequence $a_1, a_2, ... a_n$ and a longest increasing subsequence of it, say $a_{j_1}, a_{j_2}, ..., a_{j_k}$, where $j_1, j_2, ..., j_k$ represent the indices of the elements of a that are part of the subsequence. Consider now some prefix of the subsequence, say $a_{j_1}, a_{j_2}, ..., a_{j_{k-1}}$. The key observation is that it must be the case that of all subsequences of a that end with the element at position j_{k-1}, this one is the longest. If it were not, then we could replace the prefix of our proposed longest increasing subsequence with a longer one. In other words, the prefixes of a longest increasing subsequence are themselves longest increasing subsequences that end at a particular earlier element. This suggests the following subproblems for a dynamic programming approach:

$$\texttt{DP}[i] = \{\text{The length of the longest increasing subsequence that ends with the element at position } i\},$$

for $1 \le i \le n$. To find the longest increasing subsequence that ends with the element at position i, we need to figure out what its prefix could have been. Since the sequence must be increasing, this is simple, the prefix could be any subsequence that ends with an element $a[j]$ such that $a[j] < a[i]$ (so that we maintain the increasing property). So, let's just try all of the preceding elements and pick the best one.

$$\texttt{DP}[i] = 1 + \begin{cases} 0 & \text{if } a[i] \le a[j] \text{ for all } j < i, \\ \max\limits_{\substack{1 \le j < i \\ a[j] < a[i]}} \texttt{DP}[j] & \text{otherwise.} \end{cases}$$

The solution will then be the maximum value of $\texttt{DP}[i]$ (Note that the solution is **not** necessarily the value of $\texttt{DP}[n]$, since the longest increasing subsequence might not include the element a_n). There are n subproblems, each of which takes $O(n)$ time to evaluate, hence the solution takes $O(n^2)$ time.

Problem 7. Consider a pair of sequences $a_1, a_2, ..., a_n$ and $b_1, b_2, ..., b_m$ of length n and m. Devise a dynamic programming algorithm that finds the length of a *longest common subsequence* of a and b. A common subsequence is a sequence that is both a subsequence of a and a subsequence of b. Your algorithm should run in $O(nm)$. [Hint: This problem is very similar to the edit distance problem]

Solution

Consider two sequences $(a_i)_{1 \le i \le n}$ and $(b_i)_{1 \le i \le m}$ and a longest common subsequence of the two $(c_i)_{1 \le i \le k}$. Suppose that the element c_k, the final element in the longest common subsequence occurs at position i_1 in a, and at position i_2 in b. The remaining prefix of c must be a longest common subsequence of $a[1..i_1 - 1]$ and $b[1..i_2 - 1]$. If it were not, we could make our longest common subsequence even longer. This suggests that our subproblems should involve the prefixes of the sequences a and b (the same subproblems used by the edit distance problem).

$$\texttt{DP}[i, j] = \{\text{The length of a longest common subsequence of } a[1..i] \text{ and } b[1..j]\}$$

for all $0 \le i \le n, 0 \le j \le m$. To write the recurrence, we note that if the prefixes $a[1..i]$ and $b[1..j]$ end in the same element, ie. if $a[i] = b[j]$, then that element is the final element of a longest common subsequence of $a[1..i]$ and $b[1..j]$. If they differ, then it is not possible for both of them to be part of a longest common subsequence of $a[1..i]$ and $b[1..j]$, because they would necessarily be the last elements and they are not the same. In this case, we simply try removing either $a[i]$ or $b[j]$. Hence we can write the following recurrence:

$$\texttt{DP}[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ 1 + \texttt{DP}[i-1, j-1] & \text{if } a[i] = b[j], \\ \max(\texttt{DP}[i-1, j],\ \texttt{DP}[i, j-1]) & \text{otherwise.} \end{cases}$$

The solution is the value of $\texttt{DP}[n, m]$. There are $O(nm)$ subproblems, and each of them can be evaluated in constant time, hence this solution runs in $O(nm)$ time.

Problem 8. Consider a sequence of integers $a_1, a_2, ..., a_n$ of length n. Devise a dynamic programming algorithm that finds the subarray of a with maximum sum. A subarray or substring of a is a contiguous subsequence, ie. a subsequence consisting of consecutive elements.

(a) Your algorithm should run in $O(n^2)$ time.

(b) Your algorithm should run in $O(n)$ time. [Hint: Be greedy]

Solution

To solve the problem in $O(n^2)$, we compute the sums $a[i..j]$ for all substrings $[i..j]$. Doing so naively would take $O(n^3)$ time, but since

$$\text{sum}(a[i..j]) = \text{sum}(a[1..j]) - \text{sum}(a[1..i-1]),$$

we can use dynamic programming to compute them all in $O(n^2)$ time. Formally, we write the subproblems

$$\texttt{DP}[i] = \{\text{The sum of the elements in } a[1..i]\},$$

for $0 \le i \le j \le n$. The recurrence is then given by

$$\texttt{DP}[i] = \begin{cases} 0 & \text{if } i = 0, \\ \texttt{DP}[i-1] + a_i & \text{otherwise.} \end{cases}$$

The solution is the maximum value of $\texttt{DP}[j] - \texttt{DP}[i]$ for $0 \le i \le j \le n$. We have $O(n)$ subproblems, each of which takes constant time to compute, hence we spend $O(n)$ time computing the DP array. We then spend $O(n^2)$ time trying all intervals $[i..j]$, hence the total time spent is $O(n^2)$.

We can make this much faster by adding a greedy element to our solution. Suppose that we are looking at index $j+1$ and the subarray with largest sum that finishes at position $j+1$. If for some $i \le j$ the sum in the interval $a[i..j]$ is positive, then it is better to extend that sum using element $j+1$ than to start a new sum. And of course we would like to extend the largest such sum. If for all $i \le j$, the sum in the interval $a[i..j]$ is non-positive, then we can simply starting with a_{j+1} on its own. Using this observation, lets $\texttt{DP}'[0] = 0$ and for $1 \le j \le n$ lets write the subproblems

$$\texttt{DP}'[j] = \max_{1 \le i \le j} \sum_{k=i}^{j} a_k.$$

Note that in $\texttt{DP}'[j]$, a_j needs to be part of the subarray. We can use the observation above to write the recurrence

$$\texttt{DP}'[j] = \begin{cases} 0 & \text{if } j = 0, \\ \texttt{DP}'[j-1] + a_j & \text{if } \texttt{DP}'[j-1] > 0, \\ a_j & \text{otherwise.} \end{cases}$$

The solution is the maximum value of $\texttt{DP}'[j]$ over all j. There are $O(n)$ subproblems and each can be evaluated in constant time, hence the solution runs in $O(n)$ time. Note that since each subproblem only cares about the previous subproblem, we can apply the space-saving trick and solve the problem in $O(1)$ auxiliary space (as long as we keep the maximum value $\texttt{DP}'[j]$ seem so far in one variable)!

Problem 9. You are a young child, out for a walk with your parent. You are walking down a quiet street. Your parent is in a hurry, but you are more interested in collecting pretty rocks. Your parent has agreed to a compromise where they are happy for you to cross the street back and forth to pick up rocks, but only if you keep up.

Since you are a very organised child, you have determined the value to you of each rock along both sides of the streets, and represented this data in 2 arrays, `left` and `right`, of equal length. The i^{th} element in an array represents the value of the rock on the i^{th} square of pavement on that side of the street.

Because of your parent's rule about keeping up, you have the following restrictions to your movement. If you are on pavement square i on one side of the street you can either:

- Pick up the rock on square i on that side, and move to square $i+1$ on the same side.

- Cross the street diagonally to square $i+1$ on the other side (but not pick up the rock).

You start on pavement 0 on the left side of the street.

Devise a dynamic programming algorithm to determine the maximum value of rocks that you can obtain. Your algorithm should run in O(n), where n is the length of `left`.

Solution

Since the street has 2 sides, and which side we are on affects our options, we need to keep track of this. Lets have two arrays, `DPleft` and `DPright`, with the following overlapping subproblems:

$$\texttt{DPleft[i]} = \{\text{The maximum value we can obtain if we started from pavement i on the left}\}.$$

$$\texttt{DPright[i]} = \{\text{The maximum value we can obtain if we started from pavement i on the right}\}.$$

Clearly the last cell in each array is just the value of the rock on that square of pavement. For any other cell, the value of the cell would be the maximum of picking up the rock and moving one cell along, or not picking up the rock, and crossing the road.

$$\texttt{DPleft[i]} = \begin{cases} \texttt{left[n]} & \text{if } i = n, \\ \texttt{max(left[i] + DPleft[i+1], DPright[i+1])} & \text{otherwise} \end{cases}$$

$$\texttt{DPright[i]} = \begin{cases} \texttt{right[n]} & \text{if } i = n, \\ \texttt{max(right[i] + DPright[i+1], DPleft[i+1])} & \text{otherwise} \end{cases}$$

Supplementary Problems

Problem 10. A ferry that is going to carry cars across the bay has two lanes in which cars can drive onto. Each lane has a total length of L. The cars that wish to board the ferry line up in a single lane, and are directed one by one onto one of the two lanes of the ferry until the next car cannot fit in either lane. Depending on which lanes the cars are directed onto, fewer or more cars may fit on the ferry. Given the lengths of each of the cars (which will not be longer than the length of the ferry L), and assuming that the distance between cars when packed into the ship is negligible, write a dynamic programming solution that determines the maximum number of cars that can be loaded onto the ferry if distributed optimally.

(a) Solve the problem in whatever time complexity you can (without resorting to brute force).

(b) Improve your solution to $O(nL)$ time complexity (if it is not already).

For example, suppose that the car lengths are 2, 2, 7, 4, 9, 8, 1, 7, 3, 3 and the ferry is 20 units long. An optimal solution is to load 8 cars in lanes arranged like 2, 2, 7, 9 and 4, 8, 1, 7.

Solution

Let's denote the lengths of the cars in the queue by $l_1, l_2, ..., l_n$. We are looking to find the longest prefix of (l_i) that we can fit onto the ferry, without going over its length limit L. Intuitively, this problem is similar to knapsack, since we have a capacity constraint (the length of the ferry vs the capacity of the bag) and items that we want to fit into it. In this case though, its like we have two knapsacks, since there are two lanes for cars on the ferry. Suppose that the ferry currently has L_1 room remaining in the first lane, and L_2 room remaining in the second lane. Then, when we board a car with length l_i, we will be left with either $L_1 - l_i$ and L_2 room on the ferry, or with L_1 and $L_2 - l_i$ room left on the ferry, depending on whether we board the car into lane 1 or lane 2. We should select whichever of the two can accommodate more of the remaining cars when done optimally.

The first solution that comes to mind is to write a dynamic programming algorithm with the following subproblems:

$$\texttt{DP}[i, L_1, L_2] = \begin{cases} \text{The maximum number of cars from car } i \text{ onwards that can be loaded onto} \\ \text{the ship with } L_1 \text{ room left in the first lane and } L_2 \text{ room left in the second lane} \end{cases}$$

for all $1 \leq i \leq n+1$ and $0 \leq L_1, L_2 \leq L$.

Our choices are to either load car i into the first lane (if it fits) or into the second lane (if it fits). We return zero when there is no room left in either lane for the next car. A recurrence expressing this might look as follows:

$$\texttt{DP}[i, L_1, L_2] = \begin{cases} 0 & \text{if } i = n+1 \\ 0 & \text{if } l_i > L_1 \text{ and } l_i > L_2 \\ 1 + \texttt{DP}[i+1, L_1 - l_i, L_2] & \text{if } l_i > L_2 \\ 1 + \texttt{DP}[i+1, L_1, L_2 - l_i] & \text{if } l_i > L_1 \\ 1 + \max(\texttt{DP}[i+1, L_1 - l_i, L_2], \texttt{DP}[i+1, L_1, L_2 - l_i]) & \text{otherwise} \end{cases}$$

This solution however has $O(nL^2)$ subproblems, which is very expensive. We can optimise this by noticing that we are actually storing redundant information in our subproblems. If we know that we have loaded the cars from 1 to i, then we know that their total weight is $\text{sum}(l[1..i])$. The cars loaded must have taken up this much space, so it must be true that

$$(L - L_1) + (L - L_2) = \text{sum}(l[1..i]),$$

and hence if we have L_1 space remaining in lane 1, then we know that there is precisely $L_2 = 2L - \text{sum}(l[1..i]) - L_1$ space remaining in lane 2, so the third parameter of our subproblems is redundant. To ensure that we can solve each subproblem in constant time, we need to be able to compute $\text{sum}(l[1..i])$ in constant time. To do so, just precompute the sums before running the dynamic programming algorithm. A better solution would then have subproblems as follows:

$$\texttt{DP}[i, L_1] = \begin{cases} \text{The maximum number of cars from car } i \text{ onwards that} \\ \text{can be loaded onto the ship with } L_1 \text{ room left in the first lane} \end{cases}$$

for all $1 \leq i \leq n+1$ and $0 \leq L_1 \leq L$. The recurrence is the same, but we compute L_2 using the formula above instead of storing it as a parameter to the subproblem.

$$\texttt{DP}[i, L_1] = \begin{cases} 0 & \text{if } i = n+1 \\ 0 & \text{if } l_i > L_1 \text{ and } l_i > 2L - \text{sum}(l[1..i-1]) - L_1 \\ 1 + \texttt{DP}[i+1, L_1 - l_i] & \text{if } l_i > 2L - \text{sum}(l[1..i-1]) - L_1 \\ 1 + \texttt{DP}[i+1, L_1] & \text{if } l_i > L_1 \\ 1 + \max(\texttt{DP}[i+1, L_1 - l_i], \texttt{DP}[i-1, L_1]) & \text{otherwise} \end{cases}$$

The answer is the value of $\texttt{DP}[1, L]$. This solution has $O(nL)$ subproblems, each of which can be evaluated in constant time, hence it can be computed in $O(nL)$ time.

Problem 11. You and your friend are going to play a game. You start with a row of n coins with values $a_1, a_2, ..., a_n$. You will each take turns to remove either the first or last coin. You continue until there are no coins left, and your score is the total value of the coins that you removed. You will make the first move of the game.

(a) Show that the greedy strategy in which you simply pick the most valuable coin every time is not optimal.

(b) Devise a dynamic programming algorithm that determines the maximum possible score that you can achieve if both you and your friend make the best possible moves.

For example, given the coins 6, 9, 1, 2, 16, 8, the maximum value that you can achieve when going first is 23.

Solution

First, we rule out the greedy strategy. We cannot simply take the larger of the two elements and hope to achieve the best score. For instance, suppose the sequence was $2, 100, 1, 1$. If you select 2, then your friend gets to take 100, and you surely lose. If we take the 1 from the right, then no matter which element your friend takes, you get the 100 and win.

At each stage of the game, the coins that remain will be some contiguous subarray of the original coins. Suppose that we are currently looking at the subarray from i to j inclusive. We have exactly two choices, to take element i or to take element j. If we take element i, then our friend has to play with elements $i+1$ to j inclusive, and has exactly the same goal – to maximise the best sum he can get. Similarly, if we take element j, our friend has to play with elements i to $j-1$ inclusive, and tries to maximise his sum. This illustrates the substructure of the problem. Let us therefore define the following subproblems:

$$\texttt{DP}[i,j] = \{\text{The maximum score that we can obtain playing first from the subarray } a[i..j]\}$$

for $1 \le i \le j \le n$. The obvious base is when $i = j$, we have $\texttt{DP}[i,j] = a_i$. How do we write the recurrence? Well, we have two choices, to take element i or to take element j. If we take element i, then our friend will take the maximum score possible from elements $a[i+1..j]$, so his score will be $\texttt{DP}[i+1,j]$, and hence our score will be what is left:

$$a_i + (\text{sum}(a[i+1..j]) - \texttt{DP}[i+1,j]) = \text{sum}(a[i..j]) - \texttt{DP}[i+1,j]$$

Similarly, if we select element j, then our score will be

$$a_j + (\text{sum}(a[i..j-1]) - \texttt{DP}[i,j-1]) = \text{sum}(a[i..j]) - \texttt{DP}[i,j-1]$$

We should pick the best of the two, so our best possible score is

$$\max\begin{cases}\text{sum}(a[i..j]) - \texttt{DP}[i+1,j] \\ \text{sum}(a[i..j]) - \texttt{DP}[i,j-1]\end{cases} = \text{sum}(a[i..j]) - \min(\texttt{DP}[i+1,j],\ \texttt{DP}[i,j-1]))$$

Note that the max flipped to a min when we factor out the $-$ sign (since $\max(-a_i) = -\min(a_i)$). Finally, we can write the recurrence as follows:

$$\texttt{DP}[i,j] = \begin{cases} a_i & \text{if } i = j, \\ \text{sum}(a[i..j]) - \min(\texttt{DP}[i+1,j],\ \texttt{DP}[i,j+1]) & \text{otherwise.}\end{cases}$$

The solution is the value of $\texttt{DP}[1,n]$. This gives us a total of $O(n^2)$ subproblems. We can evaluate each subproblem in constant time provided that we can evaluate $\text{sum}(a[i..j])$ in constant time. This can be achieved by precomputing the partial sums, ie. $\text{sum}(a[1..i])$ and using the fact that $\text{sum}(a[i..j]) = \text{sum}(a[1..j]) - \text{sum}(a[1..j-1])$. With this achieved, we solve the problem in $O(n^2)$ time.

Problem 12. A sequence is a *palindrome* if it is equal to its reversal. For example, `racecar` and $5, 3, 2, 3, 5$ are palindromes. Given a sequence $a_1, a_2, ..., a_n$ of length n, solve the following problems.

(a) Devise a dynamic programming algorithm that determines the length of the longest palindromic **subsequence** of a. Your algorithm should run in $O(n^2)$ time.

(b) Devise an algorithm that determines the length of the longest palindromic **substring** of a. Your algorithm should run in $O(n^2)$ time. (You do not have to use dynamic programming).

(c) **(Advanced)** Devise a dynamic programming algorithm that determines the length of the longest palindromic **substring** of a. Your algorithm should run in $O(n)$ time.

Solution

(a) Longest palindromic subsequence in $O(n^2)$

The recursive substructure of palindromes is straightforward to see. A sequence $a[1..n]$ is a palindrome if and only if $a_1 = a_n$ and $a[2..n-1]$ is also a palindrome. An empty sequence or a single element sequence are also palindromes. To build a palindromic subsequence from a, we are either going to use the endpoints of a, ie. a_1 and a_n if they are equal, or we will ignore one of them. If we decide to use the endpoints, then we recursively seek a palindromic subsequence of $a[2..n-1]$. This motivates the following subproblems:

$$\texttt{DP}[i, j] = \{\text{The length of the longest palindromic subsequence of a[i..j]}\},$$

for all $1 \le i \le j \le n$. For each subproblem $\texttt{DP}[i, j]$, we check whether $a[i]$ and $a[j]$ are equal, if so, they are part of a palindrome, otherwise, they are not, and one of them must stay unused. This gives us the following recurrence:

$$\texttt{DP}[i, j] = \begin{cases} 1 & \text{if } i = j, \\ 2 + \texttt{DP}[i+1, j-1] & \text{if } a[i] = a[j], \\ \max(\texttt{DP}[i+1, j],\ \texttt{DP}[i, j-1]) & \text{otherwise.} \end{cases}$$

The answer is the value of $\texttt{DP}[1, n]$. There are $O(n^2)$ subproblems, and each of them can be evaluated in constant time, hence this algorithm runs in $O(n^2)$.

(b) Longest palindromic substring in $O(n^2)$

The naive solution to this problem would involve trying all substrings $a[i..j]$ and checking whether it is a palindrome. Since there are $O(n^2)$ substrings, and checking whether a substring is a palindrome takes $O(n)$ time, this solution would take $O(n^3)$ time.

Instead, let's think about palindromes in a slightly different way. A palindrome is a string that is symmetric about the middle. So, one way to detect a palindrome is to simply try all middle points and see how wide we can make the substring before it becomes asymmetrical. We must try each element of the sequence as the middle (which would yield odd length palindromes) and also all gaps between elements as the middle (which would yield the even length palindromes). Since we try $O(n)$ middle points and each substring can only be $O(n)$ long, this solution takes $O(n^2)$.

A possible implementation of this idea in pseudocode is given below.

```
function LONGEST_PALINDROME($a[1..n]$)
    Set answer $= 1$
    for $i = 1$ to $n$ do
        Set $j = 1$
        while $i - j \ge 1$ and $i + j \le n$ and $a[i-j] = a[i+j]$ do
            answer $= \max($answer$, 2j+1)$
            $j = j + 1$
        end while
        Set $j = 0$
        while $i - j + 1 \ge 1$ and $i + j \le n$ and $a[i-j+1] = a[i+j]$ do
            answer $= \max($answer$, 2j)$
            $j = j + 1$
        end while
    end for
    return answer
end function
```

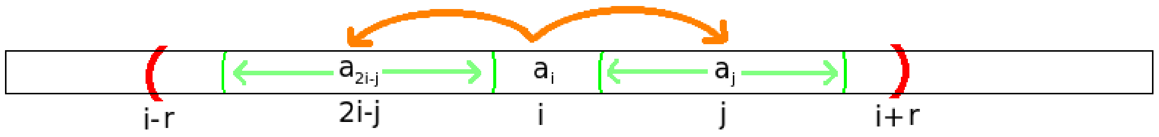
(c) Longest palindromic substring in $O(n)$

To solve this problem in $O(n)$ requires significantly more effort. First, we assume without loss of generality that we only need a solution for seeking palindromes of odd length. Why is this true? Suppose we wish to find the longest palindrome of even length, then we can simply take the input sequence $a_1, a_2, ..., a_n$ and pad it with identical elements in between every adjacent pair, to obtain $a_1, \$, a_2, \$, a_3, \$, ..., a_{n-2}, \$, a_{n-1}, \$, a_n$. Any even-length palindrome of a will show up as an odd-length palindrome of the padded sequence. This observation is important, since we are going to use the symmetry of palindromes as a basis for this solution, and we need a middle element to do so.

Recall the solution from Part (b), in which we determine for each middle point i, how wide we can make a palindrome centred at i. We are going to do the same thing, but more efficiently. Instead of referring to the length of a palindrome, we will refer to the radius of the palindrome, which is the length that it extends in each direction away from the centre. In other words, a palindrome of radius r is length $2r+1$. Our subproblems will therefore be the following:

$$\texttt{DP}[i] = \{\text{The radius of the largest odd-length palindrome centered at position } i\}$$

We compute this DP by using some clever observations about the symmetry of palindromes. These are easiest illustrated with diagrams:



Suppose we know that at position i, the radius of the largest palindrome is r and we want to find the largest palindrome centred at position j. If position j falls inside the radius of the palindrome centred at i, we know that position j is equal to position $i-(j-i) = 2i-j$ by symmetry. Therefore, if the largest palindrome centred at position $2i-j$ falls completely inside the radius of the palindrome centred at i, so too does the largest palindrome centred at position j, since they are exactly the same by symmetry! In this case, we deduce the radius of the largest palindrome centred at j in constant time.

If instead the largest palindrome centred at position $2i-j$ extends beyond the radius of the palindrome centred at i, then we know that the radius of the largest palindrome centred at j is at least size $i+r-j$ by symmetry. From here, we perform the naive algorithm and see how far it can keep going.

Finally, if the position j does not fall inside any existing palindrome, we simply perform the naive algorithm. Throughout this process, we keep track of (i, r) corresponding to the furthest to the right that any palindrome found so far extends to, and use that palindrome's symmetry to avoid computing radii that we already know. We refrain from writing a recurrence relation for this problem since it would be far too verbose and not provide any additional insight. Instead, here is some pseudocode.

```
function LONGEST_PALINDROME($a[1..n]$)
    Set answer $= 1$
    Set DP$[i] = 0$ for all $1 \le i \le n$
    Set $i = 1, r = 0$
    for $j = 2$ to $n$ do
        Set $w = \min(\text{DP}[2i-j], i+r-j)$ if $j < i+r$ else $0$
        while $j - w \ge 1$ and $j + w \le n$ and $a[j-w] = a[j+w]$ do
            $w = w + 1$
        end while
        if $j + w > i + r$ then
            $i = j, r = w$
        end if
        DP[$i$] = $w$
        answer = max(answer, $2w + 1$)
    end for
    return answer
end function
```

Now, we just have to convince ourselves that this algorithm runs in $O(n)$. It does not feel like it, since we still perform the naive algorithm whenever fall outside an existing radius. The key observation is that for each subproblem DP[i], we either solve it in constant time if our reflection is inside the current radius, or we perform the naive algorithm, which is guaranteed to extend the current radius further to the right. Since we only perform non-constant work when the current radius does not cover us, and we always move the current radius further to the right for each unit of work we do, the most extra work we can do in total is $O(n)$, since after this much, the current radius will subsume the entire string. Therefore, the entire algorithm runs in $O(n)$ time.

Problem 13. (Advanced) Consider a pair of sequences $a_1, a_2, ..., a_n$ and $b_1, b_2, ..., b_m$ of length n and m. A supersequence of a sequence a is a sequence that contains a as a subsequence. Devise an algorithm that finds the length of a *shortest common supersequence* of a and b. A common supersequence is a sequence that is both a supersequence of a and a supersequence of b. Your algorithm should run in $O(nm)$.

Solution

There are two ways to solve this problem. The first is to directly write a dynamic programming solution, whose recurrence is extremely similar to that of the longest common subsequence. A much simpler way to solve this problem is to simply relate it to the longest common subsequence problem. Observe that the shortest common supersequence must contain the longest common subsequence of a and b, and hence its length is just

$$|\text{SCS}(a,b)| = n + m - |\text{LCS}(a,b)|.$$

We can solve the longest common subsequence problem in $O(nm)$ time, hence we can also solve the shortest common supersequence problem in $O(nm)$ time.

To solve this problem directly with dynamic programming, we use similar observations that we made for the longest common subsequence problem, and write the subproblems:

$$\texttt{DP}[i,j] = \{\text{The length of the shortest common supersequence of } a[1..i] \text{ and } b[1..j]\}$$

for all $0 \le i \le n, 0 \le j \le m$. The recurrence is then given by

$$\texttt{DP}[i,j] = \begin{cases} i & \text{if } j = 0, \\ j & \text{if } i = 0, \\ 1 + \texttt{DP}[i-1, j-1] & \text{if } a[i] = b[j], \\ 1 + \min(\texttt{DP}[i-1, j],\ \texttt{DP}[i, j-1]), & \text{otherwise.} \end{cases}$$

We have $O(nm)$ subproblems, each of which can be evaluated in constant time, hence this algorithm will take $O(nm)$ time.

Problem 14. (Advanced) Suppose that I give you a string S of length n and a list L consisting of m words with length at most n. Devise a dynamic programming algorithm to determine the minimum number of strings from L that must be concatenated to form the string S. You may use a word from L multiple times. Your algorithm should run in $O(n^2 m)$ time.

Solution

Consider a particular suffix of the string S, say $S[i..n]$ for some $1 \le i \le n$. S needs to be made of the concatenation of words from L, so in particular, we need to try to make some prefix of $S[i..n]$ out of a word from L. Let's just try all of the possible words and see which ones fit. If we pick a word, say $w \in L$, then we need to make the remaining string $S[i+|w|..n]$ in as few words as possible. This illuminates the substructure of the problem. Let's write the subproblems

$$\texttt{DP}[i] = \{\text{The minimum number of words needed to form the suffix } S[i..n]\},$$

for $0 \le i \le n$. For each suffix, we can simply try each word w, see if it correctly matches the characters $S[i..i+|w|-1]$ and if so, recursively find the minimum number of words to form what remains.

$$\texttt{DP}[i] = \begin{cases} 0 & \text{if } i = n, \\ \infty & \text{if no word } w \in L \text{ is a prefix of } S[i..n] \\ 1 + \min\limits_{\substack{w \in L \\ w \text{ prefix of } S[i..n]}} \texttt{DP}[i+|w|] & \text{otherwise.} \end{cases}$$

The answer is the value of $\texttt{DP}[1]$. We have $O(n)$ subproblems, and for each of them we try all m words and perform a string comparison which takes $O(n)$ time. In total, this brings us to $O(n^2 m)$ time.

Problem 15. (Advanced) Given three strings A, B, and C of length n, m, and $n+m$ respectively, determine whether C can be formed by interleaving the characters of A and B. In other words, determine whether or not A and B can be found as disjoint subsequences of C. Your algorithm should run in $O(nm)$ time.

Solution

Suppose that we choose to use the first character of A as the first character of C. It is then possible to complete the string if and only if the strings $A[2..n]$ and $B[1..m]$ can be interleaved to form the string $C[2..n+m]$. Otherwise, if we use the first character of B, then we must interleave $A[1..n]$ and $B[2..m]$ to form $C[2..n+m]$. Let's just try both choices and see if either work. This suggests the following subproblems

$$\texttt{DP}[i, j, k] = \{\text{Is it possible to interleave } A[i..n] \text{ and } B[j..m] \text{ to form } C[k..n+m]?\},$$

for $1 \le i \le n, 1 \le j \le m, 1 \le k \le n+m$. This is not incorrect, but it is inefficient as it requires $O(nm(n+m))$ subproblems. We can notice that the third parameter of the subproblems k is redundant, since if we are up to characters i and j in A and B respectively, then we are necessarily up to position $i+j-1$ in C. A better set of subproblems is therefore

$$\texttt{DP}[i, j] = \{\text{Is it possible to interleave } A[i..n] \text{ and } B[j..m] \text{ to form } C[i+j-1..n+m]\},$$

for $1 \le i \le n+1, 1 \le j \le m+1$. The recurrence then simply tries both choices, to use a character from A or a character from B and sees if either work.

$$\texttt{DP}[i, j] = \begin{cases} \textbf{True} & \text{if } i = n+1 \text{ and } j = m+1, \\ \texttt{DP}[i+1, j] \textbf{ or } \texttt{DP}[i, j+1] & \text{if } A[i] = C[i+j-1] \text{ and } B[j] = C[i+j-1], \\ \texttt{DP}[i+1, j] & \text{if } A[i] = C[i+j-1], \\ \texttt{DP}[i, j+1] & \text{if } B[j] = C[i+j-1], \\ \textbf{False} & \text{otherwise.} \end{cases}$$

It is possible to successfully interleave the strings if $\texttt{DP}[1,1]$ is **True**. We have $O(nm)$ subproblems and each can be evaluated in constant time, hence this solution runs in $O(nm)$ time.

Problem 16. Given an $n \times n$ matrix A, your task is to find the submatrix with the maximum possible sum. A submatrix of a matrix A is a matrix consisting of the elements of some contiguous set of rows and columns of A.

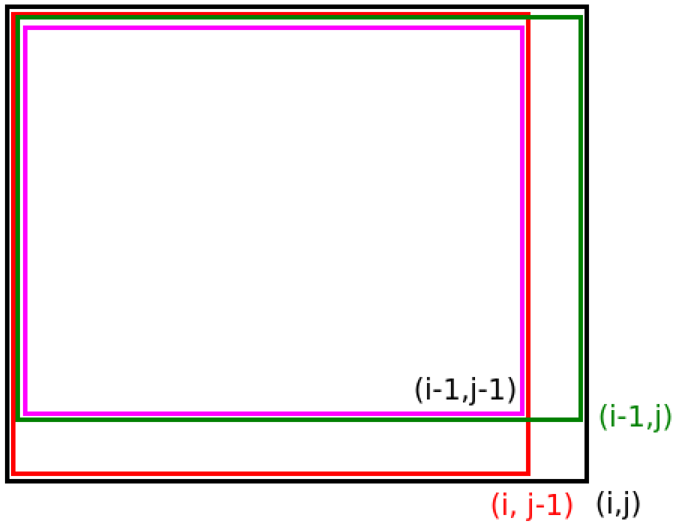
1. Devise a dynamic programming algorithm that runs in $O(n^4)$ time.
2. **(Advanced)** Devise an algorithm that runs in $O(n^3)$ time. [Hint: Use your solution to Problem 8(b)]

Solution

The solution to Part 1 is pretty much the same as the solution to Problem 8, but with more cases. We note that we can compute the sum of a submatrix $(1,1)$ to (i,j) by the following

$$\text{sum}(A[1..i][1..j]) = A[i][j] + \text{sum}(A[1..i-1][1..j]) + \text{sum}(A[1..i][1..j-1]) - \text{sum}(A[1..i-1][1..j-1]).$$

Draw a diagram and you will quickly see why this works.



Hence we write the subproblems:

$$\texttt{DP}[i,j] = \text{sum}(A[1..i][1..j])$$

for $0 \le i, j \le n$. The recurrence comes from the equation above, giving us

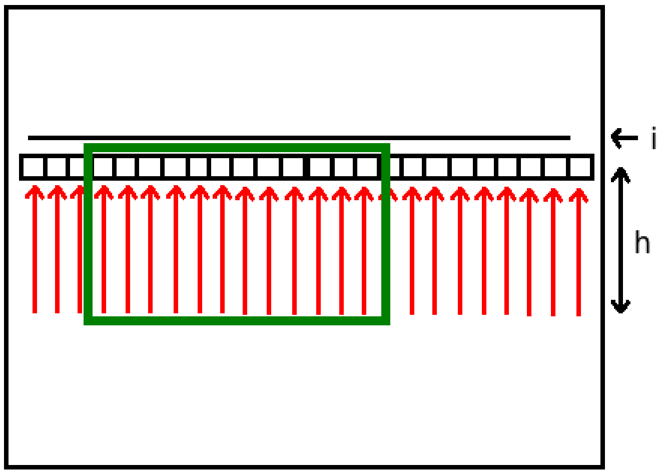
$$\texttt{DP}[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ A[i][j] + \texttt{DP}[i-1,j] + \texttt{DP}[i,j-1] - \texttt{DP}[i-1,j-1] & \text{otherwise.} \end{cases}$$

We have $O(n^2)$ subproblems and each takes constant time to solve, so computing this takes $O(n^2)$ time. We can then try all $O(n^4)$ submatrices $A[i..j][k..l]$, and compute their sums in constant time by using the fact that

$$\begin{aligned} \text{sum}(A[i..j][k..l]) &= \text{sum}(A[1..j][1..l] - A[1..i-1][1..l] - A[1..j][a..j-1] + A[i-1][j-1]) \\ &= \texttt{DP}[j,l] - \texttt{DP}[i-1,l] - \texttt{DP}[i,j-1] + \texttt{DP}[i-1,j-1]. \end{aligned}$$

Draw a diagram like the one above to see why this works. We evaluate $O(n^4)$ sums, each in constant time, and hence this approach takes $O(n^2 + n^4) = O(n^4)$ time.

To solve this problem faster, we make use of the $O(n)$ algorithm for the 1-D case from Problem 8(b). We note that the maximum sum subarray must begin at some particular row, and have some particular height. Suppose that the maximum sum submatrix begins at row i and has height h.



For a given, fixed value of i and h, we can compute the maximum sum submatrix by compressing the h rows from i to $i+h-1$ into a single row, and then running the 1-D algorithm from Problem 8(b). There are n rows to try, and for each height we try, we can extend the previously compressed row by the next row in $O(n)$ time. There are $O(n)$ possible heights, and each can be evaluated in $O(n)$ time with the 1-D algorithm, hence this approach solves the 2-D problem in $O(n^3)$.

This approach also generalises and solves the 3-D problem in $O(n^5)$, and in general, the k-D problem in $O(n^{2k-1})$. To solve the k-D problem, we try all possible starting rows and heights, and compress it into a $(k-1)$-D problem and recursively solve that, until we solve the base case, the 1-D problem in $O(n)$.

Problem 17. Consider the problem of nicely justifying text on a page. Given a sequence of word lengths $l_1, l_2, \ldots, l_n$ and a page width w, we wish to figure out the best way to justify the words on a page of width w. We define the quality of a line containing the words from index i to j inclusive by the following scoring function

$$\texttt{score}[i..j] = \begin{cases} \infty & \text{if } \operatorname{sum}(l[i..j]) > w \\ (w - \operatorname{sum}(l[i..j]))^3 & \text{otherwise} \end{cases}$$

The aim is to split the words into lines such that the total score of all the lines is minimised. Devise a dynamic programming algorithm that computes the minimum possible score. Implement your solution in Python and extend it such that it also prints the optimal justification for a given list of words. Your algorithm should run in $O(n^2)$ time to compute the minimum score, and $O(T)$ time to print the justified words, where $T = \operatorname{sum}(l[1..n])$.

Solution

Suppose that we have already chosen how to justify the first $i-1$ words on previous lines, and are now considering how to justify words i onwards. Our choices are to include just one word on this line, two words on this line, three words, and so on until the line becomes too long and the length exceeds w. Let's just try all of these options recursively. Once we have selected the words that will go on the current line, we must arrange the remainder of the words optimally. We therefore have the following subproblems:

$$\texttt{DP}[i] = \{\text{The minimum score required to justify the words } [i, n]\}$$

for $1 \le i \le n+1$. Our solution will then simply try all possibilities for the current line and recursively justify the following line, like so.

$$\texttt{DP}[i] = \begin{cases} 0 & \text{if } i = n+1, \\ \min\limits_{\substack{i \le j \le n \\ \operatorname{sum}(l[i..j]) \le w}} \left((w - \operatorname{sum}(l[i..j]))^3 + \texttt{DP}[j+1]\right) & \text{otherwise.} \end{cases}$$

We do not want to compute sum($l[i..j]$) naively at each step, or our algorithm will take $O(n^3)$ time, so we must be careful to compute the sum incrementally as we consider each additional word on this line. Alternatively, we could precompute the partial sums sum($l[1..j]$) and use those to compute the sums in constant time.

The solution is the value of DP[1]. We have $O(n)$ subproblems and take at most $O(n)$ time to evaluate each of them, hence this algorithm runs in $O(n^2)$ time.

To print the justified words, we simply backtrack through the words until we find a line whose score plus the remaining score is the optimal score according to the DP table. We repeat this until we have printed all of the words. A pseudocode implementation might look something like this. Assuming that the DP table has already been computed and that we have the means to compute the sums in constant time:

```
function JUSTIFY(words[1..$n$], $l[1..n]$, DP[1..$n+1$], $w$)
    Set $i = 1$
    Set best_score = DP[$n+1$]
    while $i \le n$ do
        Set $j = i$
        while $j \le n$ and $(w - \text{sum}(l[i..j]))^3 + \text{DP}[j+1] \neq$ best_score do
            $j = j + 1$
        end while
        print words[$i..j$]
        best_score = best_score - $(w - \text{sum}(l[i..j]))^3$
        $i = j$
    end while
end function
```

Problem 18. Along a particular river, there are $2n$ houses, exactly n of them on each side of the river. The owner of each house has a friend in one of the houses on the other side of the river. Each person is friends with exactly one other person. The house owners are tired of having to swim across the river to meet their friends each day, so they are going to connect some of the houses by bridges. Each person would like to have a bridge that connects their house directly to their friend's house, but this is not always possible since some bridges would have to cross. What is the maximum number of bridges that can be built between friends' houses without any of them crossing? Devise a dynamic programming algorithm to solve this problem. Your algorithm should run in $O(n^2)$ time.

Solution

Let's focus on the bottom bank of the river, and say we have an array $f[1..n]$, which denotes the position 1 to n of the house of the i^{it} person's friend on the top bank. What we would like to do is for each house, to choose whether we should build the bridge connected to that house or not, and then simply try every possibility recursively. In order to decide whether it is possible to build a bridge however, we need to know which bridges have been built before, otherwise they might overlap. We could try storing all subsets of bridges as our subproblems, but this would yield exponential complexity, so we need something better.

Instead, suppose we evaluate the houses on the bottom from left to right. Rather than remembering every single bridge that has been built so far, all we really need to know is the furthest to the right that any bridge on the top side has reached. If the bridge desired by person i goes to a position less than this, then it is not possible to build this bridge, or it would overlap, otherwise it is. So, let's make our subproblems:

$$\text{DP}[i,j] = \begin{cases} \text{The maximum number of bridges that can be built from house } i \text{ onwards if} \\ \text{the furthest to the right bridge on the top bank so far extends to house } j \end{cases}$$

for $1 \le i \le n+1$, and $0 \le j \le n$. At each position i, we check whether it is possible to build the bridge or not. This is possible if and only if $f[i] > j$. If it is possible, we try to build that bridge and then recursively ask how many more bridges can be built. Otherwise, we do not attempt to build this bridge and simply move on to house $i+1$.

$$\texttt{DP}[i,j] = \begin{cases} 0 & \text{if } i = n+1, \\ \max(1+\texttt{DP}[i+1, f[i]],\ \texttt{DP}[i+1,j]) & \text{if } f[i] > j, \\ \texttt{DP}[i+1,j] & \text{otherwise.} \end{cases}$$

The solution is the value of $\texttt{DP}[1,0]$. We have $O(n^2)$ subproblems and each of them is evaluated in constant time, hence the time complexity of this solution is $O(n^2)$.

Problem 19. You have access to a set of n different types of cardboard boxes. Each type of box is rectangular, with a particular length, width, and height. Your goal is to build a tower of boxes that is as tall as possible. For stability purposes, you can only stack a box on top of another if the dimensions of the base of the top box are strictly smaller than the dimensions of the top of the lower box. You may rotate the boxes in any way that you wish, and may use as many of the same type of box as necessary (but each layer can only have one box). For example, you can stack a box with a 2×3 base on top of a box with a 4×3 base (by rotating the box so that $2 < 3$ and $3 < 4$), but you cannot stack a 2×3 box on top of a 3×3 box. Write a dynamic programming algorithm that computes the height of the tallest possible tower. Your algorithm should run in $O(n^2)$ time.

Solution

To simplify things a little bit, let's start by making six copies of every box, one for each possible orientation. This way, we do not have to think about rotating the boxes, and the complexity only increases by a constant, as we have $6n$ boxes. Note that in this formulation, a box will never be repeated, since the base lengths must be strictly decreasing. For $1 \le i \le 6n$, box i is described by three parameters, (w_i, l_i, h_i), the width, length, and height. Suppose that we currently have a stack of boxes in which box i is on top. We can subsequently use any box j such that $w_j < w_i$ and $l_j < l_i$ (we do not have to think about rotations since we have a copy of each box for every possible rotation) and the height will increase by h_j. We should just try every possible box and take the one that results in the tallest tower. So, let's write the subproblems:

$$\texttt{DP}[i] = \{\text{The maximum possible tower height of a tower with box } i \text{ at the bottom}\}$$

for $1 \le i \le 6n$. With each box as the base, we simply try every possible box to be the next box, resulting in a recurrence like this

$$\texttt{DP}[i] = h_i + \begin{cases} 0 & \text{if there is no box with } w_j < w_i \text{ and } l_j < l_i, \\ \max\limits_{\substack{1 \le j \le 6n \\ w_j < w_i \text{ and } l_j < l_i}} \texttt{DP}[j] & \text{otherwise.} \end{cases}$$

The solution is the maximum value of $\texttt{DP}[i]$ for all $1 \le i \le 6n$. Since we have $O(n)$ subproblems and each of them takes $O(n)$ time to evaluate, this solves the problem in $O(n^2)$ time.

Problem 20. Given a sequence of n integers $a_1, a_2, ..., a_n$, and two integers k and d, determine whether it is possible to partition the integers a_i into groups such that each group contains at least k items, and for each pair of items a_i, a_j in the same group, $|a_i - a_j| \le d$. Come up with a dynamic programming algorithm for this problem.

(a) Find an algorithm that runs in $O(n^2)$.

(b) **(Advanced)** Find an algorithm that runs in $O(n\log(n))$.

(c) **(Advanced)** Assume that the sequence (a_i) is sorted. Find an algorithm that runs in $O(n)$.

Solution

Given the constraint that all elements in the same group must be close to each other (within d of each other), we should intuitively see that, if there is a valid solution, then it possible to obtain a valid solution in which groups are formed of contiguous segments of elements in sorted order. Let's quickly prove this. Suppose that there exists a solution containing at least two groups, say G_1 and G_2 such that $\min(G_2) < \max(G_1)$ and $\min(G_1) < \max(G_2)$. Then we can swap $\min(G_1)$ and $\max(G_2)$ and the groups will still be valid. This process could be repeated until there are no such groups remaining, at which point we can order the groups such that

$$\min(G_1) \le \max(G_1) \le \min(G_2) \le \max(G_2) \le \min(G_3) \le \max(G_3) \le \min(G_4)...$$

Thus we obtain a valid solution in which the elements are partitioned into groups of contiguous elements in sorted order. So, our first step is to sort the elements of a, which can be done in $O(n\log(n))$ time. To form a valid group starting with a particular element a_i, we should select elements up to some $j \ge i+k-1$ such that $a_j - a_i \le d$, and such that the remaining items from a_{j+1} onwards can be correctly grouped. Let's simply try all possibilities recursively and see if any of them work. Our subproblems will therefore be:

$$\texttt{DP}[i] = \{\text{Is it possible to group } a[i..n] \text{ into valid groups?}\}$$

for $1 \le i \le n+1$. The recurrence simply tries every possibility and checks whether any of them are valid.

$$\texttt{DP}[i] = \begin{cases} \textbf{True} & \text{if } i = n+1, \\ \bigvee\limits_{\substack{j \ge i+k-1 \\ a[j]-a[i] \le d}} \texttt{DP}[j+1] & \text{otherwise.} \end{cases}$$

The notation $\vee_{j\ge i+k-1}$ denotes a logical **or** operator over values of j such that $i+k-1 \le j \le n$ and $a[j]-a[i] \le d$. The problem is feasible if DP[1] is **True**. We take $O(n\log(n))$ to sort, then since we have $O(n)$ subproblems and each one can be evaluated in $O(n)$ time, this solution takes $O(n^2)$ time.

To speed this up, we need a faster way of checking whether any value of $\texttt{DP}[j+1]$ is true for j such that $j \ge i+k-1$ and $a[j]-a[i] \le d$. Since the sequence a is sorted in order, we can use a binary search to find the largest value of j, say j' such that $a[j'] - a[i] \le d$. Now the problem is to find whether between two indices $i+k \le j \le j'+1$, there exists any $\texttt{DP}[j]$ that are **True**. Let's solve this by solving the problem bottom up from $i = n+1$ to $i = 1$, and then for each value of i such that $\texttt{DP}[i]$ is **True**, storing i in a balanced binary search tree. We can then query the binary search tree for $i+k$ and $j'+1$ and determine whether there are any elements between them in $O(\log(n))$ time. Now the solution sorts in $O(n\log(n))$, then for each of $O(n)$ subproblems performs a binary search and a balanced binary search tree lookup, hence the total time complexity is $O(n\log(n))$.

Finally, suppose the sequence a is already sorted. We can further speed up our solution by eliminating the binary search and the balanced binary search tree. Instead of performing a binary search to locate j', we observe that the value of j' is monotonic, that is it will never decrease if we increase i, and will never increase if we decrease i. So, let's just remember the value of j', and when we reduce i by one, we check to see whether $a[j'] - a[i] > d$, and if so, we decrease j', otherwise we leave it. Since j' will only ever move from n to 1, this takes $O(n)$ time in total, with no binary searching required. Finally, we need to be able to check whether there is a **True** $\texttt{DP}[j]$ in the range $i+k \le j \le j'+1$. To do this, we store the partial sums of $\texttt{DP}[j]$, ie. we store

$$\texttt{PS}[j] = \sum_{k=j}^{n+1} \begin{cases} 1 & \text{if } \texttt{DP}[k] \text{ is } \textbf{True}, \\ 0 & \text{otherwise.} \end{cases}$$

which can be computed as we fill in DP. Using the partial sums, we know that there is a **True** value in the range $[l, r]$ if and only if $\texttt{PS}[r] - \texttt{PS [l-1]} > 0$. Taken all together, we solve $O(n)$ subproblems each in constant time plus $O(n)$ extra work in total to move the j' pointer. This yields an $O(n)$ time solution.

Problem 21. (Advanced) Given an unparenthesised boolean expression containing n literals (ie. an expression containing n symbols T (**True**) or F (**False**), each separated by $\wedge$ (and), or $\vee$ (or)), there are many ways to add parenthesis to the expression to change the order in which it is evaluated. For example, given the expression $T \vee T \wedge F$, there are two orders in which we could evaluate the operands:

$$(T \vee T) \wedge F, \qquad \text{or} \qquad T \vee (T \wedge F).$$

Our goal is to count the number of different orders of evaluating the expression such that it evaluates to **True**. In the example above, of the two orders, the second evaluate to **True**, so the answer is 1. Devise a dynamic programming algorithm to solve this problem in $O(n^3)$ time.

Solution

Given a boolean expression consisting of the literals $e_1, e_2, ..., e_n$, and operators $\circ_1, \circ_2, ..., \circ_{n-1}$, there are many ways in which it could be evaluated. The key observation to make is that there will always be an operand that is evaluated last. We can force a particular operand to be evaluated last by parenthesising like so:

$$(e_1 \circ_1 e_2 ... e_{k-1} \circ_{k-1} e_k) \underbrace{\circ_k}_{\text{will be evaluated last}} (e_{k+1} \circ_{k+1} e_{k+2} ... e_{n-1} \circ_{n-1} e_n).$$

For a particular expression, we can try all of the operands as being the final operand, and then count the resulting number of situations in which the expression evaluates to **True**. Suppose that the left hand side of the operand evaluates to **True** in T_1 cases, and to false in F_1 cases, and that the right hand side of the operand evaluates to **True** in T_2 cases, and to false in F_2 cases. Then the total number of ways that the expression evaluates to **True** in all possible cases is

$$\begin{cases} T_1 T_2 & \text{if the operand is } \wedge, \\ T_1 T_2 + T_1 F_2 + F_1 T_2 & \text{if the operand is } \vee. \end{cases}$$

and the total number of ways that the expression evaluates to **False** in all possible cases is

$$\begin{cases} T_1 F_2 + F_1 T_2 + F_1 F_2 & \text{if the operand is } \wedge, \\ F_1 F_2 & \text{if the operand is } \vee. \end{cases}$$

The key quantities that we need are therefore the number of ways that each particular subexpression evalutes to **True** or **False**. Let's write the following subproblems:

$$\texttt{DP}[i, j, v] = \{\text{The number of ways in which the subexpression } e[i..j] \text{ evalutes to } v\}$$

for $1 \le i \le j \le n$ and $v =$ **True** or **False**. The base cases are easy, a subexpression of size one is **True** if its literal is T, or **False** otherwise. Otherwise, we use the relations above to evaluate the number of possible ways in which the expression evaluates to **True** or **False** for every possible choice of final operand and add them all up.

We will write the recurrence in three parts since the entire thing will not fit in the width of a page. The base cases can be expressed as

$$\texttt{DP}[i, j, v] = \begin{cases} 1 & \text{if } i = j \text{ and } e[i] = v, \\ 0 & \text{if } i = j \text{ and } e[i] \ne v, \end{cases}$$

For $v =$ **True**, we have

$$\texttt{DP}[i, j, T] = \sum_{k=i}^{j-1} \begin{cases} \texttt{DP}[i, k, T]\texttt{DP}[k+1, j, T] & \text{if } \circ_k = \wedge, \\ \texttt{DP}[i, k, T]\texttt{DP}[k+1, j, T] + \texttt{DP}[i, k, T]\texttt{DP}[k+1, j, F] + \texttt{DP}[i, k, F]\texttt{DP}[k+1, j, T] & \text{if } \circ_k = \vee, \end{cases}$$

Finally, for $v = \textbf{False}$, we have

$$\texttt{DP}[i,j,F] = \sum_{k=i}^{j-1} \begin{cases} \texttt{DP}[i,k,T]\texttt{DP}[k+1,j,F] + \texttt{DP}[i,k,F]\texttt{DP}[k+1,j,T] + \texttt{DP}[i,k,F]\texttt{DP}[k+1,j,F] & \text{if } \circ_k = \wedge, \\ \texttt{DP}[i,k,F]\texttt{DP}[k+1,j,F] & \text{if } \circ_k = \vee, \end{cases}$$

The solution to the problem is the value of $\texttt{DP}[1,n]$. Since we have $O(n^2)$ subproblems and each of them required $O(n)$ time to evaluate, the time complexity of this solution is $O(n^3)$.

Problem 22. (Advanced) Consider a distributed computer network consisting of n computers. The computers are connected together such that there is exactly one path in the network between any pair of computers (in other words, the network is a tree). One of the computers contains an important message that needs to be sent to every other computer. In a single *round*, each computer that knows the message can send a copy of the message to one of the computers that it is connected to. Write a dynamic programming algorithm that computes the minimum number of rounds required to distribute the message to every computer if done optimally. Your algorithm should run in $O(n)$ time.

Solution

Let's visualise the tree as a rooted tree such that the root node is the computer that initially contains the message. In this setting, the message will always be sent down the tree from parent to child. Suppose node v currently has the message and needs to decide who to send it to such that all of its descendants receive the message in the minimal number of rounds. Its options are all of its children, and since they must all receive the message eventually, the problem is to decide what order the children should receive the message in.

For each of v's children, we can recursively compute the number of rounds that it will take for it to send the message to all of its descendants. If v's children in the order that we send to take $r_1, r_2, r_3, \ldots r_k$ rounds to distribute the message to their descendants, then the time taken to have the message to all of v's descendent is

$$\max(1+r_1, 2+r_2, 3+r_3, \ldots, k+r_k),$$

which is smallest when the sequence r is sorted largest to smallest. We should therefore order the children by the number of rounds they require to distribute the message. Putting this together, we have the subproblems:

$$\texttt{DP}[v] = \begin{cases} \text{The minimum number of rounds required to send} \\ \text{the message from } v \text{ to all of its descendants} \end{cases}$$

for all $v \in V$, where V are the vertices of the tree. Let us denote by $n_C(v)$, the number of children of the node v, and by C_v, the set of all the children of v. The recurrence can then be written as

$$\texttt{DP}[v] = \begin{cases} 0 & \text{if } v \text{ is a leaf,} \\ \max\limits_{\substack{1 \le i \le n_C(v) \\ C_v \text{ sorted by } \texttt{DP}[C_v[i]] \\ \text{in descending order}}} (i + \texttt{DP}[C_v[i]]) & \text{otherwise.} \end{cases}$$

The solution is the value of $\texttt{DP}[r]$ where r is the node that initially contains the message. We have $O(n)$ subproblems, and each of them performs $O(n_C(v))$ work, where $n_C(v)$ is the number of children of the node. The sum of all $n_C(v)$ is just the number of nodes, ie. n, hence we perform $O(n)$ work in total. We must also consider the cost to sort the children. Using a standard fast sorting algorithm, that would make the complexity $O(n\log(n))$, but since the maximum number of rounds in bounded by the number of nodes in the tree, we can use counting sort to sort in $O(n)$, hence the total time complexity is $O(n)$.

Problem 23. (Advanced) You wake up early in the morning and step outside to head to your favourite class (Algorithms and Data Structures of course), only to find that it is raining! Consider the path from your home to class to be a straight line from position 0 to position C. There are n segments on which it is currently raining.

Each segment can be described by an interval $[l, r]$ such that it is raining at all points between l and r inclusive. Conveniently, along the way there are m umbrellas on the ground that you may pick up. Each umbrella has a particular location and a particular weight. You can only move through the rain if you are currently carrying an umbrella (you do not want your laptop to get wet). However, since the umbrellas are heavy, you would like to carry them for as little time as possible. In particular, if you carry an umbrella of weight w for a distance of d, you will use up $w \times d$ energy. You may put down an umbrella at any time, or swap your current umbrella with another umbrella (including while in the rain, if an umbrella is located in the rain). Given the location of the rain and the locations and weights of the umbrellas, what is the minimum amount of energy that you must expend to get to class without getting wet? Assume that all of the raining segments do not intersect.

(a) Write a dynamic programming algorithm that runs in $O((n+m)^2)$ time.

(b) **(Extra advanced)** Write a dynamic programming algorithm that runs in $O((n+m)\log(n+m))$ time.

Solution

Let's consider the "important" points along the interval $[0, C]$. The important points are $0, C$ and those at which we have to make a decision, which are the points containing an umbrella and the ends of rainy segments. These are important since we may have to make the following decisions:

- If there is an umbrella at this point, should I take it?
- If this is the end of a rainy segment, should I drop my current umbrella?

This gives us a total of $n+m+2$ important points. At each point with an umbrella, we should consider all of the places where we could drop it. These are the ends of rainy segments, and the locations of other umbrellas which we could swap for it. So, let's sort the important points in ascending order, call them $p[1..(n+m+2)]$ and write the following subproblems:

$$\texttt{DP}[i] = \{\text{The minimum energy needed to make it from point } i \text{ (location } p[i]) \text{ to class}\}$$

for $1 \le i \le n+m+2$. We will assume that for each subproblem, you begin without an umbrella. At point i, we can decide to move forward without an umbrella to point $i+1$ provided that there is no rainy segment in between them. We can simply loop over all rainy segments to check this. If there is an umbrella at point i, then we can try taking it, and if so, decide where to drop it. If we decide to carry an umbrella from point i to point j, then we will expend

$$w(p[j] - p[i])$$

energy, where w is the weight of the umbrella. We will then try to move forward from position j, so our total minimum energy expenditure will be $w(p[j]-p[i]) + \texttt{DP}[j]$. Trying every option and selecting the best one yields the following recurrence:

$$\texttt{DP}[i] = \min \begin{cases} \texttt{DP}[i+1] & \text{if not raining in } [p[i], p[i+1]], \\ \min\limits_{j>i} \left(w(p[j]-p[i]) + \texttt{DP}[j] \right) & \text{if } p[i] \text{ has an umbrella of weight } w, \\ \infty & \text{otherwise} \end{cases}$$

with the base case $\texttt{DP}[n+m+2] = 0$. The answer is the value of $\texttt{DP}[1]$. We have $O(n+m)$ subproblems and each one takes $O(n+m)$ time to evaluate, hence this solution takes $O((n+m)^2)$ time.

In order to speed this up, we need a faster way of detecting whether a path goes through rain and a faster way of evaluating the expression $\min\left(w(p[j]-p[i]) + \texttt{DP}[j]\right)$. The first part is easy. To check whether the path from $p[i]$ to $p[i+1]$ goes through the rain, just have the rainy intervals sorted and perform a binary search. Now for the harder part. Let's rewrite the expression in question as

$$w(p[j]-p[i]) + \texttt{DP}[j] = (p[j]w + \texttt{DP}[j]) - wp[i].$$

Notice that the final term $wp[i]$ is independent of j, and the first term in brackets is a linear equation evaluated at w. We can interpret this equation as being a set of straight lines of the form $p[j]x + \texttt{DP}[j]$, where $p[j]$ and $\texttt{DP}[j]$ are the slope and y-intercept given by the solutions to previous subproblems. The problem in front of us is therefore to find the lowest value on a set of straight lines given a particular x value (in our case, we have $x = w$). This problem can be solved efficiently using a data structure known as the "Convex Hull Trick", which provides $\log(n)$ queries and insertions, where n is the number of lines. We won't describe the data structure here. If interested, you should read `https://wcipeg.com/wiki/Convex_hull_trick`. We solve the subproblems bottom up, and after solving each one, we insert the line $(p[j]x + \texttt{DP}[j])$ into the convex hull trick data structure. Now we can perform the query $\min\left(w(p[j]-p[i])+\texttt{DP}[j]\right)$ in $O(\log(n+m))$ time.

With these improvements, we still have $O(n+m)$ subproblems, but now we solve them in $O(\log(n+m))$ time each, so the total time taken is $O((n+m)\log(n+m))$.

Problem 24. Let us define a *pattern* string as follows. A pattern string P is a string consisting of lowercase letters, stars (*) and full-stops (.). We say that a text string T of lowercase letters *matches* the pattern P if we can form T from P by replacing the full-stops in P with any lowercase character (each full-stop may be replaced with a different character), and replacing the stars by any (possibly empty) string of lowercase characters (each star may be replaced by a different string). Devise a dynamic programming algorithm to determine whether a given text T of length n matches a given pattern P of length m.

(a) Your algorithm should run in $O(n^2 m)$ time.

(b) **(Advanced)** Your algorithm should run in $O(nm)$ time.

Solution

Suppose we are trying to match character i of the text string to character j of the pattern. If $P[j]$ is just a character, we can only do so if $T[i] = P[j]$. If $P[j] =$ ".", then we can match to anything. Otherwise, if $P[j] =$ "$*$", then we can match to nothing, or to as many further characters as we want. We must then match the remaining suffix of T with the remaining suffix of P. The dynamic programming solution should simply try all of these possibilities, and check whether any of them work. Let's write the subproblems as follows:

$$\texttt{DP}[i,j] = \{\text{Can we match } T[i..n] \text{ to the pattern } P[j..m]?\},$$

for $1 \le i \le n+1, \le j \le m+1$. We then have the recurrence

$$\texttt{DP}[i,j] = \begin{cases} \textbf{True} & \text{if } i = n+1 \text{ and } j = m+1, \\ \textbf{False} & \text{if } i = n+1 \text{ and } j \ne m+1, \\ \textbf{False} & \text{if } j = m+1 \text{ and } i \ne n+1, \\ \texttt{DP}[i+1, j+1] & \text{if } T[i] = P[j], \\ \texttt{DP}[i+1, j+1] & \text{if } P[j] = \text{"."}, \\ \bigvee_{k \ge i} \texttt{DP}[k, j+1] & \text{if } P[j] = \text{"}*\text{"}. \end{cases}$$

The notation $\vee_{k\ge i}$ denotes a logical **or** operator over $\texttt{DP}[k, j+1]$ for all values of k such that $i \le k \le n+1$. In other words, if any of $\texttt{DP}[k, j+1]$ are true, then $\texttt{DP}[i,j]$ is true, otherwise it is false. The text T matches the pattern P if $\texttt{DP}[1,1]$ is **True**. We have $O(nm)$ subproblems, and each one can be evaluated in $O(n)$ time, hence this solution runs in $O(n^2 m)$ time.

To speed this up, we can use the same subproblem definition and recurrence, we just need to figure out how to evaluate the logical or $\vee_{k\ge i}\texttt{DP}[k, j+1]$ in constant time. To do so, lets solve the problem in a bottom-up fashion, and specifically solve the subproblems in descending order of i followed by descending order of j. We will also store an extra array, specifically, lets store $w[j]$ for each value of j, denoting whether we have seen a value of i such that $\texttt{DP}[i][j] =$**True** yet. Since we are solving in descending order i then descending order of j, when we are solving subproblem $\texttt{DP}[i][j]$, the value of $w[j+1]$ is precisely whether or not there exists $k \ge i$ such that $\texttt{DP}[k][j+1] =$ **True**, which is precisely what the logical **or** computes. Now we can solve each subproblem in constant time, hence the total time complexity of the solution is $O(nm)$.