Week 5 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, psuedocode may be provided where it illustrates a particular useful concept.

Assessed Preparation

Problem 1. Implement the solution to the coin change problem described in the lectures.

(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. 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.

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

(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 psuedocode 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

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

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 $\texttt{DP}[1] = c_1$. We leverage the optimal substructure to write the recurrence

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

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

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

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

Tutorial Problems

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

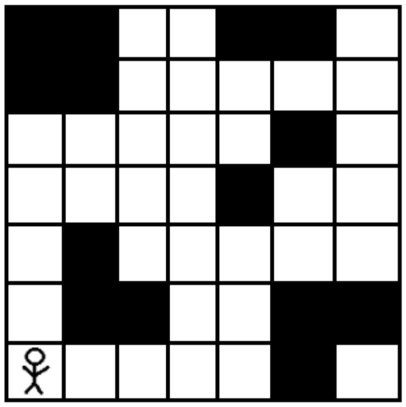
To reconstruct the optimal set of houses, we make the observation that in a range of houses $[1..i]$, house i is optimal to sell to if $\texttt{DP}[i] > \texttt{DP}[i-1]$. Why is this true? If $\texttt{DP}[i] > \texttt{DP}[i-1]$, then since $\texttt{DP}[i] = \max(\texttt{DP}[i-1], \texttt{DP}[i-2] + c_i) > \texttt{DP}[i-1]$, it must be true that $\texttt{DP}[i] = \texttt{DP}[i-2] + c_i > \texttt{DP}[i-1]$. 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 $\texttt{DP}[0..n] = 0$
    $\texttt{DP}[1] = c_1$
    for i = 1 to $n$ do
        $\texttt{DP}[i] = \max(\texttt{DP}[i-1], \texttt{DP}[i-2] + c_i)$
    end for
    Set houses = []
    Set $i = n$
    while $i > 0$ do
```

```
        if DP[$i$] > 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 $\texttt{DP}[i, j]$, the following subproblems:

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

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

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

The value of $\texttt{DP}[1,1]$ 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:

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

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:

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

The optimal solution is the value of $\texttt{DP}[1,1]$. 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 $a_1, a_2, ..., a_n$ of length n. Devise a dynamic programming algorithm that finds the maximum sum of all subarrays of a. 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. Note that if the sum in some interval $a[i..j]$ is non-negative, and $\text{sum}(a[i..j]) + a[j+1]$ is non-negative, then it is always beneficial to add element $j+1$ into the sum. If $a[i..j]$ is negative, then we are better off simply starting with $a[j+1]$ on its own. Using this observation, lets write the subproblems

$$\texttt{DP}[i] = \{\text{The maximum possible subarray sum of elements } a[1..i]\}.$$

for $0 \le i \le n$. We can use the observation above to write the recurrence

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

Note that we write $\max(a[i], 0)$ since if $a[i]$ is negative, then it is better to take nothing at all (the empty subarray) than $a[i]$ on its own. The solution is the maximum value of $\texttt{DP}[i]$ over all i. 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!

Supplementary Problems

Problem 9. 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 greater 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 $1 \le i \le n+1$ and $0 \le L_1, L_2 \le L$.

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 $1 \le i \le n+1$ and $0 \le L_1 \le L$. 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 10. 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 11. Suppose you have n jobs where each job has a start time, a finish time and a profit value associated with it. A valid subset of the jobs is a subset of jobs such that no two jobs overlap. Your goal is to find a valid subset of jobs with maximum profit value. Below is an example.

Suppose you have following jobs where the first two values in each job represent start and finish time and the third value represents the profit of the job.

```
Job1 = [1, 10, 50]
Job2 = [8, 12, 100]
Job3 = [10, 45, 100]
Job4 = [18, 55, 150]
Job5 = [50, 60, 50]
```

The optimal schedule is Job2 and Job4 with total profit of 250. Note that Job1, Job3, Job5 is a valid schedule but its total profit is 200. Your goal is to write a dynamic programming solution to find the total profit of the optimal schedule.

1. Your algorithm should run in $O(n^2)$ time
2. **(Advanced)** Your algorithm should run in $O(n\log(n))$ time

Solution

Suppose we decide to take a particular job, Job i, denoted by $[s_i, f_i, w_i]$, where s_i = the start time, f_i = the finishing time, and w_i = the profit of the job. This rules out doing any job that overlaps the time $[s_i, f_i]$. Keeping track of all possible non-overlapping jobs would require tracking all subsets of jobs, which would require $O(2^n)$ subproblems, far too many! Instead, we should work in order, selecting the job that we will do first, then the job that we will do second, and so on. This way, all we need to know at any particular point is what time the last job we took finished at. This means that there are only $O(n)$ things to track, which is fine. If we choose to perform job i, then we should subsequently perform the most profitable set of jobs that start at or after f_i. So let's write the subproblems:

$$\texttt{DP}[i] = \{\text{The maximum profit achievable by taking non-overlapping jobs that start at or after } f_i\},$$

for $1 \le i \le n$. For each subproblem, we can simply look at all jobs, and try those that start after f_i, in other words, all jobs j such that $s_j \ge f_i$. This yields the recurrence

$$\texttt{DP}[i] = \begin{cases} 0 & \text{if there are no jobs } j \text{ with } s_j \ge f_i \\ \max\limits_{\substack{1\le j\le n \\ s_j \ge f_i}} (w_j + \texttt{DP}[j]) & \text{otherwise.} \end{cases}$$

The answer is then the maximum value of $w_i + \texttt{DP}[i]$ for all $1 \le i \le n$, ie. we try all possible first jobs. We have $O(n)$ subproblems, each of which takes $O(n)$ time to evaluate, hence this solutions runs in $O(n^2)$ time.

We can speed this up by removing some redundancy in our recurrence. Note that for every subproblem $\texttt{DP}[i]$, we try all subsequent jobs as the next job. This is rather redundant, since the jobs that come after us will also try the jobs that come after them, and so on. So really, all we should be trying is the job that comes immediately after us, since the rest will be covered transitively by the future jobs.

So, let's rework our ideas a bit. Let's sort the jobs by order of start time, and say we are currently considering job i. Then we just need to decide whether to do this job or not. If we decide not to do this job, we should consider the next job, ie. Job $i+1$. If we do decide to take Job i, then we should subsequently consider the earliest job that starts at or after time f_i. Since the jobs are sorted, we can quickly locate such a job with a binary search. So, we have the following subproblems

$$\texttt{DP}[i] = \begin{cases} \text{The maximum profit achieveable by taking non-overlapping jobs} \\ \text{from Job } i \text{ onwards, where the jobs are sorted by start time.} \end{cases}$$

for $1 \le i \le n+1$. At each subproblem, we consider performing Job i, and either do not, in which case we move on to considering Job $i+1$, or we perform it and then consider the earliest subsequent job that comes after f_i.

$$\texttt{DP}[i] = \begin{cases} 0 & \text{if } i = n+1 \\ \max \begin{cases} \texttt{DP}[i+1], \\ w_i + \texttt{DP}[j] \quad \text{where } j = \min\limits_{k>i}\{s_k \ge f_i\} \end{cases} & \text{otherwise,} \end{cases}$$

where we locate $j = \min\limits_{k>i}\{s_k \ge f_i\}$ with a binary search, or return $n+1$ if no such job can be found (if Job i finishes after all jobs can start). The answer to the problem is the value of $\texttt{DP}[1]$. This solutions sorts the input, which takes $O(n\log(n))$, and has $O(n)$ subproblems, each of which can be computed in $O(\log(n))$ time (the time required for binary search), hence this solution works in $O(n\log(n))$ 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 be not used. 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, lets 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 psuedocode 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
```

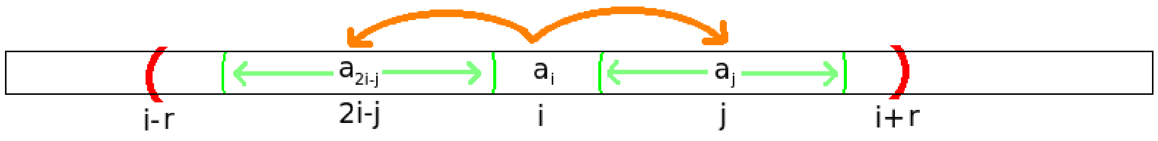
(c) Longest palindromic substring in $O(n)$

To solve this problem in $O(n)$ requires significantly more effort. First, we make the observation that without loss of generality, we can seek only 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. Assume from here that without loss of generality, n is odd.

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 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 psuedocode.

```
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 $w = \min(\texttt{DP}[2i-j], i+r-j)$ **if** $j < i+r$ **else** $0$
            **while** $j - w \geq 1$ **and** $j + w \leq n$ **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**
            $\texttt{DP}[i] = w$
            $\texttt{answer} = \max(\texttt{answer}, 2w+1)$
        **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 $\texttt{DP}[i]$, 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.