Section 5: Solutions

1. Going to parties

You are planning your calendar for n days, where every day, there is a party that you can go to. Every day, you can choose to go to that party or stay in and catch-up on sleep. If you party, you will enjoy yourself. But you can only party for two consecutive days — if you party three days in a row, you'll fall too far behind on sleep and miss class. Luckily, you have an excellent social sense, so you know exactly how much you will enjoy any of the parties, and you have assigned each day a positive integer happiness score in an array $H[1\,..\,n]$. You get 0 happiness if you do not go to the party. Maximize the sum of your happiness for these n days, while not going out for more than two consecutive days.

(a) Write a summary for this problem.

Solution:

Input: An array of positive integers $H[1\,..\,n]$.
Output: The maximum value of $\sum_{i \in S} H[i]$ over all $S \subseteq \{1, \dots, n\}$ with no three consecutive indices.

Dynamic programming is difficult because examples rarely help until you know your subproblems. You need to abstractly analyze the problem to determine subproblems first.

You've seen a strategy in lecture before. Here it is:

1. Let OPT(j) = the optimal solution on inputs up to j.
2. Divide OPT(j) into cases based on what to do with the jth element.
3. For each case, can you use OPT$(j-1)$ to handle it? Why or why not?
4. If you could not handle it, what additional information would help you? What kinds of subproblems are these?

(b) Now you try. Say how to adapt each step to the party problem at hand.

(i) Let OPT(j) = the optimal solution on inputs up to j.

Solution:

Let OPT(j) = the maximum happiness you can get on days $1\,..\,j$.

(ii) Divide OPT(j) into cases based on what to do with the jth element.

Solution:

On day j, you will either sleep or party.

(iii) For each case, can you use OPT$(j-1)$ to handle it? Why or why not?

Solution:

If you sleep, you happiness will be OPT$(j-1)$. However, if you party, there is no way to use OPT$(j-1)$, because maybe OPT$(j-1)$ called for partying both yesterday and the day before.

(iv) If you could not handle it, what additional information would help you? What kinds of subproblems are these?

Solution:

Keep track of the last time you slept (equivalently, how many days you've been partying). This way, you could choose to only party if you slept within the last 2 days.

Two equivalent ways to implement this:

- Let OPT(j, s) = maximum happiness from days $1\,..\,j$, assuming you slept s days ago, for $s = 0, 1, 2$.
- Look back to OPT$(j-2)$ if you last slept on day $j-1$, or look back to OPT$(j-3)$ if you last slept on day $j-2$.

Both ways involve only prefixes as subproblems.

(c) Now that you know what form you want your subproblems to take, try this example to flesh out the details and convince yourself that it works.

[9, 2, 3, 8, 6, 6, 4, 7, 1]

Solution:

If using OPT(j, s):

$H[j]$	9	2	3	8	6	6	4	7	1
OPT$(j, 0)$	0	9	11	12	20	25	26	30	36
OPT$(j, 1)$	9	2	12	19	18	26	29	33	31
OPT$(j, 2)$	9	11	5	20	25	24	30	36	34

If using OPT(j):

$H[j]$	9	2	3	8	6	6	4	7	1
OPT(j)	9	11	12	20	25	26	30	36	36

(d) Write the recurrence relation. Don't forget the base cases.

Solution:

If using OPT(j, s), the recursive case is:

$$\begin{aligned}
\text{OPT}(j,0) &= \max(\text{OPT}(j-1,0), \text{OPT}(j-1,1), \text{OPT}(j-1,2))\\
\text{OPT}(j,1) &= \text{OPT}(j-1,0) + H[j]\\
\text{OPT}(j,2) &= \text{OPT}(j-2,0) + H[j-1] + H[j]
\end{aligned}$$

It's nice to also notice OPT$(j, 2)$ = OPT$(j-1, 1) + H[j]$. By avoiding looking back 2 days, it makes the pseudocode easier later on, and simplifies the base cases too. There are many equivalent options for base cases, and if you use this, you can delete the last line from each:

$$\begin{aligned}
\text{OPT}(0,s) &= 0\\
\text{OPT}(1,2) &= H[1]
\end{aligned}
\qquad
\begin{aligned}
\text{OPT}(0,s) &= 0\\
\text{OPT}(-1,0) &= 0
\end{aligned}
\qquad
\begin{aligned}
\text{OPT}(1,0) &= 0\\
\text{OPT}(1,1) &= H[1]\\
\text{OPT}(1,2) &= H[1]\\
\text{OPT}(2,2) &= H[1] + H[2]
\end{aligned}$$

Note that the first two are much shorter and simpler — take advantage of the ability to use 0 as the base case. Note, however, that it's not always true that OPT$(0) = 0$, though it will usually be true whenever optimizing for a sum or total of something, but you should just think it through.

If using OPT(j), the recursive case is:

$$\text{OPT}(j) = \max(\text{OPT}(j-1), \text{OPT}(j-2) + H[j], \text{OPT}(j-3) + H[j-1] + H[j]).$$

Again, many equivalent options for the base cases:

$\text{OPT}(0) = 0$	$\text{OPT}(-2) = 0$	$\text{OPT}(1) = H[1]$
$\text{OPT}(1) = H[1]$	$\text{OPT}(-1) = 0$	$\text{OPT}(2) = H[1] + H[2]$
$\text{OPT}(2) = H[1] + H[2]$	$\text{OPT}(0) = 0$	$\text{OPT}(3) = \max(H[1] + H[2], H[1] + H[3], H[2] + H[3])$

In dynamic programming, the pseudocode will end up being a fairly direct translation of the recurrence, so we'll do the proof first. In this class, we will focus on just proving the recursive case. A complete formal proof is, of course, induction.

(e) Prove your recurrence to be correct.

Solution:

Cases based on when we last slept. We must have slept within the last 2 days, otherwise the schedule violates the excessive partying condition.

Case 1: Sleep today. We get nothing today. All schedules on $1\,..\,j-1$ are compatible with sleeping today, so the maximum happiness we can get in this case is OPT$(j-1)$.

Case 2: Slept yesterday. Then we have no reason not to party today, and all schedules on $1\,..\,j-2$ are compatible with sleeping yesterday, so OPT$(j-1) + H[j]$.

Case 3: Slept 2 days ago. Then we had no reason not to party yesterday, nor today, and all schedules on $1\,..\,j-3$ are compatible, thus OPT$(j-3) + H[j-1] + H[j]$.

Overall, the maximum of these three cases is the maximum possible overall happiness.

Now, on to the implementation details. Even though we're use a recurrence relation, do not call your function recursively! Calling the function recursively can lead to blowing up the running time, so we need to consider how to remember the solutions to subproblems.

(f) (i) State the parameters for your subproblems and what kind of structure you will use to store them.

Solution:

From now on, these solutions will only cover the case of the OPT(j) solution with base cases for 0, 1, and 2.

Parameters are j from 0 to n. We will store OPT in an array of length $n+1$.

(ii) Describe the order for evaluating your subproblems.

Solution:

We will evalute the base cases, then each OPT(j) for j from 3 to n.

(g) Write the pseudocode for your iterative algorithm.

Solution:

```
1: Let OPT be a zero-indexed array of length $n+1$ initialized to anything.
2: Let OPT[0] $\leftarrow 0$, OPT[1] $\leftarrow H[1]$, and OPT[2] $\leftarrow H[1] + H[2]$.
3: for $j \leftarrow 3$ to $n$ do
4:     OPT[$j$] $\leftarrow \max(\text{OPT}[j-1], \text{OPT}[j-2] + H[j], \text{OPT}[j-3] + H[j-1] + H[j])$.
5: output OPT[$n$].
```

(h) What is the running time of your algorithm?

Solution:

It is one for loop with constant work per iteration, so $O(n)$.

Lastly, you will sometimes be asked to return the optimal object, rather than the optimal value. However, doing it naively can cost you. You've seen this in lecture already, but a brief review:

- A first idea might be to track the optimal object at each j, instead of the optimal value. In today's problem, this will use $O(n^2)$ total time and space, very bad!
- Thus, we usually try to backtrack to find the optimal object after finding the optimal value. You may need to leave some hints for yourself to know where to go.

(i) How would you modify the algorithm if you were asked to return the optimal party schedule?

Solution:

While processing each day j, additionally rememebr an arrow pointing to which OPT$(j-k)$ the optimal solution uses (for some $k = 1, 2, 3$). Note that an arrow pointing to OPT$(j-k)$ means we slept $k-1$ days ago. At the end of the algorithm, follow the arrows back to collect the set of days we slept.

The following problems will not be covered in section, but may be useful to think about.
We recommend trying them by yourself first. Solutions will be posted in the evening.

2. Longest common subsequence

Given two strings, $s = s_1 \dots s_m$ with length m and $t = t_1 \dots t_n$ with length n, find the length of their longest common subsequence. (A subsequence may not be contiguous. That is, one finds a subsequence by taking any subset of the indices, and putting together the letters at those indices in their original order.)

Here are a few examples:

- **Input**: $s =$ `backs`, $t =$ `arches`
  Solution: The longest common subsequence is `acs`, so the output should be 3.
- **Input**: $s =$ `skaters`, $t =$ `hated`
  Solution: The longest common subsequence is `ate`, so the output should be 3.

Solution:

Here is a sample chain of thought that might lead you to the following solution:

- Let OPT$(i, j) =$ the longest common subsequence between $s_1 \dots s_i$ and $t_1 \dots t_j$. Based on the first example from section, I will consider what to do with the last characters s_i and t_j.
- There are two cases. Either they are paired in the longest common subsequence, or they are not.
  - If s_i and t_j are not paired, then the two immediately previous instances, OPT$(i, j-1)$ and OPT$(i-1, j)$ are sufficient to determine the longest common subsequence. (In problems with 2 arrays, these two are the standard thing to look at first, analogous to OPT$(j-1)$ in a 1 array problem.)
  - If s_i and t_j are paired, then OPT$(i, j-1)$ cannot be used because it is possible that s_i was already paired in OPT$(i, j-1)$, and similarly for OPT$(i-1, j)$.

    Thus we need to remove both s_i and t_j from consideration. Thus, we should look at OPT$(i-1, j-1)$.

At this point, even if you are not entirely certain that OPT$(i-1, j-1)$ is the right thing to look at or why it works, you can try to manually compute an example as we did in section. Then you will discover the right recurrence.

Note: We are not particularly interested in seeing your chain of thought on homeworks or exams. In particular, *you should not use this chain of thought as a proof of correctness.* This is written for your reference. Please only answer the questions asked as shown below, and read the solution to understand how proofs are different from chains of thought.

(a) Define, in English, the quantities that you will use for your recursive solution.

Solution:

Let OPT(i, j) be the longest common subsequence between elements $1 \,..\, i$ in s and $1 \,..\, j$ in t. The parameter i ranges from 0 to m, and the parameter j ranges from 0 to n (so we have our base cases in our memoization table for ease of calculation).

(b) Given a recurrence relation for the quantities you have defined.

Solution:

The recursive case is:

$$\text{OPT}(i,j) = \begin{cases} 1 + \text{OPT}(i-1, j-1) & \text{if } s_i = t_j \\ \max(\text{OPT}(i-1,j), \text{OPT}(i, j-1)) & \text{if } s_i \neq t_j \end{cases}$$

The base cases are:

$$\text{OPT}(i, 0) = 0$$
$$\text{OPT}(0, j) = 0$$

(c) Argue for the correctness of your recurrence relation.

Solution:

For the first case of $s_i = t_j$, we are asserting that is always optimal to pair s_i with t_j. To prove this:

- If s_i and t_j were both unpaired, the solution would not be optimal because we could just pair them.
- If s_i is unpaired and t_j is paired with s_k, where $k < i$, another valid solution with the same length would be to leave s_k unpaired and pair s_i with t_j. (Note that because t_j is last, s_k must also have been the last paired character of s, so the pairs continue to respect the order.)
- If t_j is unpaired but s_i is paired, the case is similar.

This covers all other cases. With s_i and t_j paired, the remaining problem is maximized by OPT$(i-1, j-1)$, which our s_i-t_j pair is compatible with.

For the second case of $s_i \neq t_j$, we cannot pair them. Because OPT$(i-1, j)$ and OPT$(i, j-1)$ cover all instances where s_i and t_j are not paired to each other, it is correct to take their maximum in this case.

For the base case, when either string is empty, the longest common subsequence must be the empty string.

(d) Describe the parameters for the subproblems in your recursion and how you will store their solutions.

Solution:

There are subproblems for $i = 0, \ldots, m$ and $j = 0, \ldots n$ which could be stored in a (2D) array of size $(m+1) \times (n+1)$.

(e) In what order can you evaluate them iteratively?

Solution:

Many orders are possible. Here are some options (either would be okay, other choices may also work):

- All base cases, then outer loop on i from 1 to m, and inner loop j from 1 to n.
- All base cases, then outer loop on j from 1 to n, and inner loop i from 1 to m.

(f) Write the pseudocode for your iterative algorithm.

Solution:

```
1: Initialize OPT$[i,0] \leftarrow 0$ and OPT$[0,j] \leftarrow 0$ for all $0 \le i \le m$ and $0 \le j \le n$.
2: for $i \leftarrow 1$ to $m$ do
3:     for $j \leftarrow 1$ to $n$ do
4:         if $s_i = t_j$ then
5:             OPT$[i,j] \leftarrow 1 +$ OPT$[i-1, j-1]$
6:         else
7:             OPT$[i,j] \leftarrow \max($OPT$[i-1,j]$, OPT$[i,j-1])$
8: output OPT$[m,n]$
```

(g) Analyze the running time of your algorithm.

Solution:

Two nested loops with m and n iterations, respectively, and constant work per iteration, thus $O(mn)$.

3. Short disjoint subarrays

You are given an array of positive integers $A[1..n]$ and an positive integer target t. You need to find the minimum total length of k disjoint (non-overlapping) contiguous subarrays of A that each have a sum of their elements equal to the target. If there are not k such subarrays, return ∞.

Here are a few examples:

- **Input**: $k = 4$, $t = 2$, and the array $[1, 1, 8, 2, 3, 2, 1, 1, 2]$
  Solution: We could do $[[1, 1], 8, [2], 3, [2], 1, 1, [2]]$, so the output should be 5. (not the only solution)
- **Input**: $k = 2$, $t = 4$, and the array $[2, 1, 2, 2, 2]$
  Solution: No two disjoint, contiguous subarrays each have sum 4, so the output should be ∞.
- **Input**: $k = 3$, $t = 4$, and the array $[1, 3, 1, 2, 1, 4, 2, 2]$
  Solution: We could do $[[1, 3], 1, 2, 1, [4], [2, 2]]$, so the output should be 5.

Solution:

Here is a sample chain of thought that might lead you to the following solution:

- Let OPT(j) = the shortest total length of k disjoint contiguous subarrays of $A[1..j]$ that each sum to t. Based on the first example from section, I will consider what do with the last element $A[j]$.
- There are two cases. Either $A[j]$ is in one of the subarrays, or it is not.
  - If $A[j]$ is not in one of the subarrays, we are simply looking for OPT$(j-1)$.
  - If $A[j]$ is in one of the subarrays, we cannot use OPT$(j-1)$ because OPT$(j-1)$ already has k subarrays

that each sum to t, and there is no room to add another $A[j]$.

Since $A[j]$ is the last element of the array we're looking at, if it is included in the optimal solution, it must be the end of a subarray whose sum is t. To fix the above issue, we would like our subproblem to be compatible with adding the entire subarray containing $A[j]$.

Let $A[s+1]$ be the start of that subarray. The subproblem cannot use $A[s+1\,..\,j]$, so we should look at OPT(s). However, we also need to look for one less subarray. Thus, we define OPT(j,K) = the shortest total length of K disjoint contiguous subarrays of $A[1\,..\,j]$ that each sum to t, and actually look at OPT$(s, K-1)$.

Note: We are not particularly interested in seeing your chain of thought on homeworks or exams. In particular, *you should not use this chain of thought as a proof of correctness.* This is written for your reference. Please only answer the questions asked as shown below, and read the solution to understand how proofs are different from chains of thought.

(a) Define, in English, the quantities that you will use for your recursive solution.

Solution:

Let OPT(j,K) = the shortest total length of K disjoint contiguous subarrays of $A[1\,..\,j]$ that each sum to t. The parameter j ranges from 0 to n, and the parameter K ranges from 0 to k.

(b) Given a recurrence relation for the quantities you have defined.

Solution:

Given j, let $A[s+1]$ be the start of the subarray that ends at $A[j]$ and has sum t, if such an index s exists. The recursive case is:

$$\text{OPT}(j,K) = \begin{cases} \min(\text{OPT}(j-1,K), \text{OPT}(s,K-1) + (j-s)) & \text{if } s \text{ exists} \\ \text{OPT}(j-1,K) & \text{if } s \text{ does not exist} \end{cases}$$

The base cases are

$$\text{OPT}(j,0) = 0$$
$$\text{OPT}(0,K) = \infty \text{ for all } K \neq 0$$

(c) Argue for the correctness of your recurrence relation.

Solution:

In the first case, where s exists, the optimal solution has (a priori) two choices. One choice is to take $A[s+1\,..\,j]$ as one of the subarrays, in which case we are compatible with all solutions in OPT$(s, K-1)$, and end up using length $j-s$. If this choice is not taken, we are compatible with all solutions in OPT$(j-1,K)$. To determine which is better, we take the minimum of these two options.

In the second case, where s does not exist, the optimal solution cannot use $A[j]$ in the solution, and must use a solution covered by OPT$(j-1,K)$.

For the base cases, when $K=0$, we are looking for no subarrays, and this is achievable with taking nothing. When $j=0$ and K is at least 1, because we are told that the target t is positive, it is impossible to have a subarray of nothing sum to t. The value ∞ is both the value we were asked to output, and a value that is compatible with the min operation that we take in the first case.

(d) Describe the parameters for the subproblems in your recursion and how you will store their solutions.

Solution:

There are subproblems for each j from 0 to n and K from 0 to k. We need a (2D) array of size $(n+1)\times(k+1)$ to store them.

(e) In what order can you evaluate them iteratively?

Solution:

Many orders are possible. Here are some options (either would be okay, other choices may also work):

- All base cases, then outer loop on j from 1 to n, and inner loop K from 1 to k.
- All base cases, then outer loop on K from 1 to k, and inner loop j from 1 to n.

(f) Write the pseudocode for your iterative algorithm.

Solution:

```
Initialize OPT$[j,0] \leftarrow 0$ and OPT$[0,K] \leftarrow \infty$ for all $0 \le j \le n$ and $1 \le K \le k$.
for $j \leftarrow 1$ to $n$ do
    for $K \leftarrow 1$ to $k$ do
        Loop down from $j$ to find $s$ such that $A[s+1\,..\,j]$ has sum exactly $t$, if such $s$ exists.
        if $s$ exists then
            OPT$[j,K] \leftarrow$ min(OPT$[j-1,K]$, OPT$[s,K-1]+(j-s)$)
        else
            OPT$[j,K] \leftarrow$ OPT$[j-1,K]$)
output OPT$[n,k]$
```

(g) Analyze the running time of your algorithm.

Solution:

Two nested loops with n and K iterations, respectively, and $O(\min(n,t))$ work per iteration to search for s (since each entry of A is positive, i.e. at least 1, we will search for at most t iterations, or until we hit the end of the array which takes n iterations). Thus, $O(nK\min(n,t))$