Section 5: Solutions

1. Lots of fun, with a normal sleep schedule

You are planning your social calendar for the month. For each day, you can choose to go do a social event or stay in and catch-up on sleep. If you go to a social event, you will enjoy yourself. But you can only go out for two consecutive days – if you go to a social event 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 social events, and have assigned each day an (integer) numerical happiness score (and you know you get 0 enjoyment from staying in and catching up on sleep). You have an array $H[]$ which gives the happiness you would get by going out each day. Your goal is to maximize the sum of the happinesses for the days you do go out, while not going out for more than two consecutive days.

1.1. Read and understand the problem

Read the problem and answer the usual quick-check-questions

- Are any words in the problem technical terms? Do you know them all?
- What is the input type?
- What is the output type?

Solution:

Technical terms: "consecutive" means in a row
Input: int[]
Output: int (the maximum sum of happinesses)

1.2. Generate Examples

Generate at least two examples along with their correct answers. It often helps at this point to ask yourself "what would a greedy algorithm be?" and design a counter-example for that algorithm **Solution:**

$[2, 2, 1, 2, 2, 1, 2, 2]$ has a maximum happiness sum of 12
$[10, 8, 15, 9, 3, 11, 12, 13]$ has a maximum happiness sum of 59

1.3. Design and justify a recursive solution

Since we know we're writing DP algorithms this week, we're going to make these steps a bit more specific to the DP process.

(a) Formulate the problem recursively – what are you looking for (in English!!), and what parameters will you need as you're doing the calculation?

(b) Write a recurrence for solving the problem you defined in the last part (the recurrence is for the answer, **not** the running time).

(c) What is your final answer (e.g. what parameters for the recurrence do you need? Is it a single value or the max/min of a set of values?)?

(d) Give a brief justification for why your recurrence is correct. You do not need a formal inductive proof, but your intuition will likely resemble one.

Solution:

(a) As usual we consider how to deal with the last day assuming that we have solutions to problems involving the previous days. To know whether we can include the last day we need to know how many consecutive days have been included immediately prior (0, 1, or 2). Therefore we define OPT(j, k) to be the most points we can earn from days/indices $1..j$ (inclusive) where we have taken k consecutive days at the right end of the subproblem for $k = 0, 1, 2$. (e.g. if $k = 2$ then we have included elements j and $j - 1$ but not element $j - 2$). Per the problem, we only allow $k \in \{0, 1, 2\}$ and $j \in \{0, 1, ..., n\}$.

(b)

$$\text{OPT}(j,0) = \begin{cases} 0 & \text{if } j = 0 \\ \max(OPT(j-1,0), OPT(j-1,1), OPT(j-1,2)) & \text{if } j \geq 1 \end{cases}$$

$$\text{OPT}(j,1) = \begin{cases} -\infty & \text{if } j = 0 \\ A[j] + \text{OPT}(j-1,0) & \text{if } j \geq 1 \end{cases}$$

$$\text{OPT}(j,2) = \begin{cases} -\infty & \text{if } j \leq 1 \\ A[j] + \text{OPT}(j-1,1) & \text{if } j \geq 2 \end{cases}$$

Note that we could put the last two together as:

$$\text{For } k = 1, 2, \text{ we have } OPT(j,k) = \begin{cases} -\infty & \text{if } j < k \\ A[j] + \text{OPT}(j-1,k-1) & \text{if } j \geq k \end{cases}$$

(c) $\max(\text{OPT}(n, 0), \text{OPT}(n, 1), \text{OPT}(n, 2))$.

(d) For OPT$(j, 0)$, if $j = 0$ there are no elements to include and the empty solution with value 0 is the best one can do. Otherwise, $j \geq 1$ and we need to skip element j. Any one of the options for the first $j - 1$ elements is possible, so we should select the best of the 3 options which are given by OPT$(j - 1, 0)$, OPT$(j - 1, 1)$ and OPT$(j - 1, 2)$.

For OPT$(j, 1)$, we must include both $A[j]$ but not $A[j - 1]$. This is impossible for $j = 0$ since there is no such element so we encode this impossibility with value $-\infty$ so that it can never be part of a maximum value. Otherwise, we include $A[j]$ to the most points among $1, ..., j - 1$ where we do not include $A[j - 1]$, which is the definition of OPT$(j - 1, 0)$.

For OPT$(j, 2)$, we must include both $A[j]$ and $A[j - 1]$ but not $A[j - 2]$. This is impossible if $j = 0$ or $j = 1$ since there are no such elements so we encode this impossibility with value $-\infty$ so that it can never be part of a maximum value. The optimal must inlcude $A[j]$ can observe that the remaining condition of including $A[j - 1]$ but not $A[j - 2]$ is precisely the condition covered by OPT$(j - 1, 1)$.

1.4. Write and Analyze the Dynamic Program

(a) Describe the set of parameters for the subproblems in the recursive calls for your algorithm and how you could store their solutions. **Solution:**

All possible pairs (j, k) where $j = 0, \ldots, n$ and $k = 0, 1, 2$, which could be stored in an $(n + 1) \times 3$ array.

(b) Describe a computation order for those subproblems that allows an iterative solution. **Solution:**

Initial values for $(0,0)$, $(0,1)$ and $(0,2)$ Outer loop j from 1 to n
Compute values for $(j,0)$, $(j,1)$ and $(j,2)$.

(c) Write the pseudocode for an iterative algorithm

Solution:

```
OPT[0,0]= 0; OPT[0,1] = -ϖ; OPT[0,2]= -ϖ
for j = 1 to n do
    OPT[j,0] = max(OPT[j-1,0],OPT[j-1,1],OPT[j-1,2])
    OPT[j,1] = A[j]+OPT[j-1,0]
    if j==1 then
        OPT[j,2]= -ϖ
    else
        OPT[j,2] = A[j]+OPT[j-1,1]
return max(OPT[n,0],OPT[n,1],OPT[n,2])
```

(d) State and justify the running time of your iterative solution

Solution:

In each of the n iteations, we check at most 3 entries, and we have 3 entries to fill, so our total running time is $\mathcal{O}(n)$.

More Problems!

2. Longest Common Subsequence

Given two strings s with length m and t with length n, find the length of their longest common subsequence. A subsequence is a (possibly non-contiguous) substring.

Examples:

Input s="backs", t="arches": the longest common subsequence is "acs", so the output should be 3.

Input s="skaters", t="hated": the longest common subsequence is "ate", so the output should be 3.

2.1. Define and justify a recursive solution

(a) Formulate the problem recursively – what are you looking for (in English!!), and what parameters will you need as you're doing the calculation? **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) Write a recurrence for solving the problem you defined in the last part (the recurrence is for the answer, **not** the running time). **Solution:**

$$\text{OPT}(i,j) = \begin{cases} 1 + \text{OPT}(i-1, j-1) & \text{if } s_i = t_j \\ \max(\text{OPT}(i-1,j), \text{OPT}(i,j-1)) & \text{if } s_i \neq t_j \\ 0 & \text{if } i < 1 \text{ or } j < 1 \end{cases}$$

(c) What is your final answer (e.g. what parameters for the recurrence do you need? Is it a single value or the max/min of a set of values?)? **Solution:**

OPT(n, m).

(d) Give a brief justification for why your recurrence is correct. You do not need a formal inductive proof, but your intuition will likely resemble one. **Solution:**

For the first case, the character at index i in s is equal to the character at index j in t, so we want to add this character to our longest common subsequence. So, we'll add 1 to the longest common subsequence we found up until we reached these characters, which is OPT$(i-1, j-1)$.

For the second case, the character at index i in s is not equal to the character at index j in t, so together they can't be part of the longest common subsequence. So then we'll look at a smaller chunk in either s or t, we'll compare the character at index i in s to the character at index $j-1$ in t, or we'll compare the character at index $i-1$ in s to the character at index j in t.

For the third case, either i or j is out of bounds which means we're not looking at any characters in either s or t, so we just return 0.

2.2. Write and Analyze the Dynamic Program

(a) Describe the set of parameters for the subproblems in the recursive calls for your algorithm and how you could 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 $(m+1) \times (n+1)$.

(b) Describe a computation order for those subproblems that allows an iterative solution. **Solution:**

Outer loop i from 0 to m.
Inner loop j from 0 to n.
    Compute answer for (i, j)

(c) Write the pseudocode for an iterative algorithm

```
for i=0 to m
   OPT[i,0] = 0
for j=0 to n
   OPT[0,j] = 0
for i=1 to n
    for j=1 to n
       OPT[i,j] = max(OPT[j-1,i],OPT[j,i-1])
       if s[i] == t[j] && then
           OPT[i,j] = max(OPT[i,j],OPT[i-1,j-1]+1)
return(OPT[m,n])
```

(d) State and justify the running time of an iterative solution. **Solution:**

Creating entry i, j requires checking at most 2 recursive calls, which each require $\mathcal{O}(1)$ time. Since we have nm entries, we need $\mathcal{O}(nm)$ time.

3. k Minimum Disjoint Subarrays, Each with Target Sum t

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

Examples:

Input $k = 4$, $t = 2$, and the array $[1, 1, 8, 2, 3, 2, 1, 1, 2]$: the four smallest disjoint subarrays whose elements each sum to 2 are $[[1, 1], 8, [2], 3, [2], 1, 1, [2]]$, so the output should be 5.

Input $k = 2$, $t = 5$, and the array $[2, 2, 2, 2]$: there aren't two subarrays whose elements each sum to 5, so the output should be ∞.

Input $k = 3$, $t = 4$, and the array $[1, 3, 1, 2, 1, 4, 2, 2]$: the three smallest disjoint subarrays whose elements each sum to 4 are $[[1, 3], 1, 2, 1, [4], [2, 2]]$ so the output should be 5.

3.1. Design and justify a recursive solution

(a) Formulate the problem recursively – what are you looking for (in English!!), and what parameters will you need as you're doing the calculation? **Solution:**

Let OPT(i, j) be the minimum sum of the lengths of j disjoint subarrays whose elements each sum to t that are found in the array from $1..i$ inclusive.
The parameter i ranges from 0 to n, and the parameter j ranges from 0 to k.

(b) Write a recurrence for solving the problem you defined in the last part (the recurrence is for the answer, **not** the running time). **Solution:**

$$\text{OPT}(i,j) = \begin{cases} \min(\text{OPT}(i-1,j), \text{OPT}(i-m,j-1)+m) & \text{if } i>0,\ j>0,\ \text{and } \exists m[\sum_{x=i-m+1}^{i} A[x] = t] \\ \text{OPT}(i-1,j) & \text{if } i>0,\ j>0 \text{ if no such } m \text{ exists} \\ 0 & \text{if } j=0 \\ \infty & \text{otherwise} \end{cases}$$

(c) What is your final answer (e.g. what parameters for the recurrence do you need? Is it a single value or the max/min of a set of values?)? **Solution:**

OPT(n, k).

(d) Give a brief justification for why your recurrence is correct. You do not need a formal inductive proof, but your intuition will likely resemble one. **Solution:**

For the first case, the element at index i can be the right-most element in a subarray of length m whose elements sum to the target t, so that subarray can be one of the j subarrays we consider. We either want to include the subarray ending with element i, in which case we add m to the minimum sum of elements in $j-1$ subarrays ending at index $i-m$, or we don't want to include the subarray ending with element i, in which case we just take the minimum sum of elements in j subarrays up to index $i-1$. To determine which is better, we take the minimum of these two options.

For the second case, there is no subarray whose elements sum to t that ends with the element at index i, so our only option is to just take the minimum sum of elements in j subarrays up to index $i-1$.

In the third case, $j = 0$, which means we want the minimum length of 0 subarrays, which is necessarily 0.

Otherwise, one of our indices is out of bounds or there are less than j subarrays from $1..i$ whose elements sum to t, so we put in a placeholder of ∞ so that as soon as we have enough valid subarrays, that number of elements will be the minimum.

3.2. Write and Analyze the Dynamic Program

(a) Describe the set of parameters for the subproblems in the recursive calls for your algorithm and how you could store their solutions. **Solution:**

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

(b) Describe a computation order for those subproblems that allows an iterative solution. **Solution:**

Outer loop i from 0 to n.
Inner loop j from 0 to k.
    Compute solution for (i, j).

(c) Write the pseudocode for an iterative algorithm

```
for i=0 to n
    OPT[i,0] = 0
for j=0 to k
    OPT[0,j] = ∞
for i=1 to n
    for j=1 to k
        sum = A[i]
        m = 1
        while sum < t && m < i do
            m = m + 1
            sum = sum + A[i-m+1]
        OPT[i,j] = OPT[i-1,j]
        if sum == t then
            OPT[i,j] = min(OPT[i,j],OPT[i-m,j-1]+m)
return OPT[m,n]
```

(d) State and justify the running time of an iterative solution. **Solution:**

Creating entry i, j requires 2 recursive calls. For one of the recursive calls we need to check if an m exists so that the element at index i can be the last element in a subarray whose elements sum to the target

t. As all elements are positive, we can have at most t elements in each subarray, so you have to check at most t possibilities for m, requiring $\mathcal{O}(t)$ time. This means each entry requires $\mathcal{O}(t)$ time overall. All other recursive calls take $\mathcal{O}(1)$ time. Since we have nk entries, we need $\mathcal{O}(nkt)$ time.

Alternatively, we can upper bound the number of m we need to check by n, thus giving us $\mathcal{O}(n^2k)$, which might be better if t is large.