HW5 - S16

Handed out 5-3 Chris Troiano 5-10, beg. of class

In all problems address the following:

(a) What subproblems are you solving?

(b) How are the subproblems related?

(c) Give a recurrence - what is the precise meaning of the entries of the recurrence?

(d) What table are you filling in, and in what order? ow is the table initialized?

(e) An arrow diagram for the recurrence.

(f) Time bound with justification.

1. KT, Ch6, #1, p. 315
   Solution:

   (a) $7-8-9-10-6$
   The following algorithm will pick:
   $8+10=18$
   The optimal algorithm will pick:
   $7+9+6=22$

   (b) $12-8-2-3-12-1-1-15$
   The following algorithm will pick:
   odd: $12+2+12+1=27$
   even: $8+3+1+15=27$
   The optimal algorithm will pick:
   $12+2+12+15=41$

   (c)
```
S[]
for i = 0 to n
        if (i == 0) S[0] = 0
        else if (i == 1) S[1] = $v_1$
        else S[i] = max (S[i-1], $v_i$ + S[i-2])
return S[]
```

   i. The subproblems that we're solving is deciding whether the v_{i-1}, or the $v_{i-2}\&v_i$ nodes are in the maximum independent set.

   ii. The subproblems are related because they build on each other. If we choose a node, the neighbor nodes are automatically taken out of the maximum independent set.

   iii.

$$opt(i) = \begin{cases} 0, & \text{if i == 0} \\ v_1, & \text{if i == 1} \\ max(v_{i-1}, v_i + v_{i-2}), & \text{otherwise} \end{cases} \quad (1)$$

If i == 0 and i == 1, are the two initialization cases.
The weight of nothing is 0.
The independent max set of a single node, must be that node.
If $i > 1$, we take the max between $v_i + v_{i-2}$ and v_{i-1}, this line solves the problem of finding the independent max set.

iv. Using the example graph:
$1 - 8 - 6 - 3 - 6$
We are filling in a 1D array where the indexes represent the number of nodes we take into consideration. The values of array are filled out from left to right with the total weight of the maximum independent set up to the i^{th} node. It is initialized with the first two indexes, this is important because the max function needs to be able to see $i - 2$ indexes back.

v. The arrow diagram:

vi. After initialization, which is 2 O(1) operations, we iterate through every node and do an O(1) comparison. This gives us an O(n) algorithm.

2. KT, Ch6, #4, p. 315 **Solution:**

(a) NY: 55 57 33 = 135
SF: 56 56 22 = 134
correct: SF all 3 months = 134
incorrect algorithm: NY $\Rightarrow$ SF + M$\Rightarrow$ SF = 143

(b) NY: 10 30 30 100
SF: 12 15 100 30
NY $\Rightarrow$ SF $\Rightarrow$ NY $\Rightarrow$ SF
This example moves each month because cost(stay in the current city) > cost(moving + rent in other city) for each month.

(c)
```
ASF[1] = $S_i$
ANY[1] = $N_i$
for i = 2 to n
        ASF[i] = $S_i$ + min (ASF[i-1], ANY[i-1] + M)
        ANY[i] = $N_i$ + min (ANY[i-1], ASF[i-1] + M)
return min (ASF[n], ANY[n])
```

i. Each iteration we determine if it is cheaper to stay in the current city, or to move, in the i-1 entry.

ii. The subproblems are related because we must check to see if the previous month would have been cheaper if we moved.

iii.

$$opt(i) = \begin{cases} ASF[i] = S_i, & \text{if i == 1} \\ ANY[i] = N_i, & \text{if i == 1} \\ ASF[i] = S_i + min(ASF[i-1], ANY[i-1] + M), & \text{otherwise} \\ ANY[i] = N_i + min(ANY[i-1], ASF[i-1] + M) & \text{otherwise} \end{cases} \tag{2}$$

We initialize both arrays with their respective city's monthly rent. This is necessary because the rest of the recurrence relies on an $i - 1$ comparison.
We always add the respective city to its array, but we add on the optimal cost either staying in the city or moving in the previous month.

iv. Each index $>$ 1 is the result of staying in the respective city, having followed the optimal path the previous months. The table is initialized with the first month of their respective arrays. This is important because we rely on i-1 comparisons each iteration.

v. The arrow diagram:

vi. Initializing both arrays are done in O(1). Then we iterate from i $>$ 1 to n, we add the current month and make an O(1) comparison. This is done n times, thus the algorithm is O(n).

3. A *contiguous* subsequence of a list S is a subsequence made up of consecutive elements of S. For instance if S is

$$5, 15, -30, 10, -5, 40, 10,$$

then 15,-30 is a contiguous subsequence but 5,15,40 is not. Give a linear-time algorithm (i.e. $O(n)$ where n is the length of the sequence S)
for the following task:

Input: A list of numbers $S = a_1, a_2, \ldots, a_n$.
Output: The largest sum you can get by summing a contiguous subsequence of S
(A subsequence of length zero has sum zero).

Hint: For each end point $j \in \{1, 2, \ldots, n\}$, find the highest sum value $H(j)$ obtainable by a contiguous subsequence ending at j.

(a) For the above example sequence, find the highest sum $H(j)$ for each value of j.

(b) Give a recurrence for the $H(j)$ quantity? How is $H(j)$ initialize?

(c) Give an algorithm for returning the contiguous subsequence with the largest sum.

Solution:

(a) The highest sum for $H(j)$:

(b)

$$H(j) = \begin{cases} max(a_1, 0), & \text{if j == 1} \\ max(a_j + H(j-1), a_j) & \text{otherwise} \end{cases} \quad (3)$$

$H(j)$ is initialized by getting the max between the first value and 0, from here, the recurrence is good to go.

(c)
```
M = 1
for j = 1 to n
        A[j] = H(j)
        if (A[j] > A [j-1]) M = j
S[]
G = A[M]
while G $\geq$ 0
        G = G - $a_m$
        S.insert $a_m$
        M–
return S[]
```

i. After running the $H(j)$ algorithm on every index, the subproblem is taking the max of the $H(i)$ and $opt(i-1)$. This will keep the largest sum in the $opt(i)$ answer.

ii. They are related because we need to check to see if the previous entry of the opt array, is larger than the current value in the array.

iii.

$$opt(i) = \begin{cases} H(1), & \text{if i == 1} \\ max(H(i), opt(i-1)) & \text{otherwise} \end{cases} \tag{4}$$

The initialization is the $H(1)$ when i == 1 because if we have only 1 element in the set, the contiguous subsequence is either the empty set, or that single element.
The contiguous subset is found with the max line. We choose between the current $H(i)$ value, or we use the previous opt value.

iv. After filling out the $H(j)$ table, we go from left to right comparing the current $H(1)$ value with the previous $opt(i-1)$ value, and take the max. It is initialized with the H(1).

v. The arrow diagram:

vi. The opt recurrence is O(n) because we just run the H(j) algorithm for every element in the set and compare it to the previous opt value. This gives us O(n) because we must iterate through the whole list, but when we add a new value it is O(1) because it is just a comparison and an addition, so we have 2n O(1) comparisons. Thus, the recurrence is O(n). The algorithm is not though, this is because I needed to do some extra math to extract the actual elements in the subsequence. The opt recurrence just returns the maximum sum for each index j.

4. A rabbit wants to go accross a narrow wooden bridge. It can do either a short hop of 2 feet or a long hop of 3 feet. Unfortunately, the bridge is old and has holes in it, that the rabbit has to jump over. For simplicity, suppose that the bridge is divided into n one-foot-long segments, each solid or with a hole in it. You are given a binary array $A[1..n]$, where the 1's indicate where the holes are in the bridge: if $A[i] = 1$, then the i-th segment has a hole, so the rabbit cannot land on it. If $A[i] = 0$, then the segment is solid, and therefore safe for the rabbit. The rabbit starts on segment 1 and ends on segment n. We assume that $A[1] = A[n] = 0$. Give an $O(n)$ algorithm for finding:

(a) the number of ways that the rabbit can go across the bridge,

(b) minimum number of hops that the rabbit would need to go across (if possible),

(c) maximum number of hops that the rabbit could use (note that it cannot go backwards).

Can the rabbit use a greedy approach to minimize/maximize the number of hops? Explain.
Solution:

(a)

$$f(i) = \begin{cases} 1, & \text{if i == 1} \\ N/A, & \text{if A[i] == 1} \\ N/A, & \text{if i < 3} \\ f(i-2) + f(i-3) & \text{otherwise} \end{cases} \tag{5}$$

i. We are trying to find the number of ways across the bridge based on the i-2 and i-3 bridge sectors.

ii. They are related because we must look at the previous decision, to find the current position to jump.

iii. The recurrence can be found at a) above.
We initialize the array with a 1 if the rabbit is on the first sector.
If the A[i] == 1, this denotes a hole and the rabbit cannot go there. I used a "N/A" because if I used a number, the f(i) would mistake it for a valid entry on account of the fact I am not leveraging a min or max function in this recurrence.
The rabbit cannot jump a single foot, therefore the i_2 index will always be "N/A". Otherwise we count whether it is possible to do a 2 foot jump or a three foot jump depending on the i-2 and i-3 indexes.

iv. We're filling out a 1D array from left to right. It is initialized with A[1] = 1, that is, there is one way to get to the 1^{st} foot of the bridge.

v. The arrow diagram:

vi. All of the algorithms are O(n) because we must iterate through the whole array and check the i-2 and i-3 indexes to find the value of the i^{th} index. To add another value to the array, we do two O(1) comparisons n times. Thus the algorithms are O(n).

(b)

$$minHop(i) = \begin{cases} 0, & \text{if i == 1} \\ \infty, & \text{if f(i) == N/A} \\ min(1 + f(i-2), 1 + f(i-3)) & \text{otherwise} \end{cases} \tag{6}$$

i. We are trying to find the minimum number of hops that the rabbit can take to get across the bridge by checking the i-2 and i-3 bridge sectors using the $f(i)$ recurrence and adding one to count the total hops taken, and then taking the min.

ii. They are related because we must look at the previous decision, to find the current position to jump.

iii. The recurrence can be found at b) above.
We initialize the array with a 0 on the first index because the rabbit didn't use any jumps to get to the first bridge sector.
If f(i) returns a "N/A" then there is a hole and we cannot count the jump. We return ∞ because the min function will not return that unless there are no moves possible.
We add one to the number that $f(i)$ returns so that we can count the number of hops instead of just the possible ways across and then take the min to find the minimum number of hops it takes to get across.

iv. We're filling out a 1D array from left to right. It is initialized with A[1] = 0, that is, the rabbit didn't use any hops to get to 1^{st} foot of the bridge.

v. The arrow diagram:

vi. All of the algorithms are O(n) because we must iterate through the whole array and check the i-2 and i-3 indexes to find the value of the i^{th} index. To add another value to the array, we do two O(1) comparisons n times. Thus the algorithms are O(n).

(c)

$$maxHop(i) = \begin{cases} 0, & \text{if i == 1} \\ -\infty, & \text{if f(i) == N/A} \\ max(1 + f(i-2), 1 + f(i-3)) & \text{otherwise} \end{cases} \tag{7}$$

i. We are trying to find the maximum number of hops that the rabbit can take to get across the bridge by checking the i-2 and i-3 bridge sectors using the $f(i)$ recurrence and adding one to count the total hops taken. We then take the max.

ii. They are related because we must look at the previous decision, to find the current position to jump.

iii. The recurrence can be found at c) above.
We initialize the array with a 0 on the first index because the rabbit didn't use any jumps to get to the first bridge sector.
If f(i) returns a "N/A" then there is a hole and we cannot count the jump. We return $-\infty$ because the max function will not return that unless there are no moves possible for that index.
We add one to the number that $f(i)$ returns so that we can count the number of hops instead of just the possible ways across and then take the max to find the maximum number of hops the rabbit can take to get across.

iv. We're filling out a 1D array from left to right. It is initialized with A[1] = 0, that is, the rabbit didn't use any hops to get to 1^{st} foot of the bridge.

v. The arrow diagram:

vi. All of the algorithms are O(n) because we must iterate through the whole array and check the i-2 and i-3 indexes to find the value of the i^{th} index. To add another value to the array, we do two O(1) comparisons n times. Thus the algorithms are O(n).

The rabbit problem cannot use the greedy approach because it doesn't have the ability to use previous solutions to find the optimal one. The greedy approach might take as many 2 foot hops as possible to find the max number of jumps, but strand itself and come up with a result that says it is impossible to cross the bridge, where if it took one 3 hop jump previously, it would have found the optimal solution. The greedy approach does not have this ability, and this is why we must use a dynamic programming solution.

5. Consider the following variant of the segmented least squares problem. Design a dynamic programming algorithm for computing is the minimum total error achievable by partitioning n points into k segments. Use the $e(i, j)$ variable used in the version given in the slides.

   Make sure the get the best running time as a function of n and k by pre-computing various statistics?

   How does the time compare against the original variant where each new segment incurs an additional cost of c?

   How can you recover the optimum segmentation?

Solution:

$$opt(n,k) = \begin{cases} 0, & \text{if n == 0} \\ min((e_i, n) + opt(i-1, k-1)) & \text{for } 1 \leq i < n \text{ and } k > 0 \end{cases} \tag{8}$$

The time is similar to the original variant. Instead of using a c, to discourage adding a bunch of segments, we just specify the number of segments that we want with variable k. This gives us a time complexity of O(kn) = O(n).
The optimum segmentation can be recovered by replacing the k with c.

(a) The subproblem we're solving is decrementing k

(b) The subproblems are related because we need to check the value of k before proceeding, and the value of k depends on the previous iteration.

(c)

$$opt(n,k) = \begin{cases} 0, & \text{if n == 0} \\ min((e_i, n) + opt(i-1, k-1)) & \text{for } 1 \leq i < n \text{ and } k > 0 \end{cases} \tag{9}$$

There is no initialization for this recurrence, because we start at n and work down through the points. The base case is when k == 0, but this will happen on the last iteration instead of the initial one.

(d) We're filling out the table from right to left and the table is not initialized.

(e) The arrow diagram:

(f) We iterate through every point k times, thus our algorithm runs in O(kn) = O(n).