Algorithms Assignment 3 Solutions

1. There is a row of n items, numbered from 1 to n. Each item has an integer value: item i has value $A[i]$, where $A[1..n]$ is an array. You wish to pick some of the items but you cannot pick two adjacent items (that is, you cannot pick both items i and $i+1$ for *any* i). Subject to this constraint, you wish to maximise the total sum of values of the items you have picked.

   (a) [**2 marks**] Provide a counterexample to show that it is not always optimal to pick the largest item.

   (b) [**3 marks**] Provide a counterexample to show that the optimal solution may require selecting a combination of odd and even numbered elements.

   (c) [**10 marks**] Give an $O(n)$ algorithm that determines this maximum total sum of values.

   Solution:

   (a) A possible solution is $[9, 10, 9]$, where it is more optimal to pick the two 9s than the 10.

   (b) A possible solution is $[10, 1, 1, 10]$, where it is optimal to pick the two 10s.

   (c) We use DP. We assume that values of all items are positive numbers.
   Subproblem(i): Among items $1 \ldots i$ pick a subset of maximal total value, denoted by $M(i)$, which satisfies the constraint.
   Recursion: $M(1) = A[1]$; $M(2) = \begin{cases} A[2], & if \;\; A[2] > A[1] \\ A[1], & otherwise. \end{cases}$;
   $M(i) = \max(A[i] + M(i-2), M(i-1))$ for all $i > 2$.

   Solution to the original problem: is given by $M(n)$.

2. [**15 marks**] A company is organising a retreat for its employees. The organisers of the retreat want to keep costs low, and know the cost of inviting each employee. Note that this may be different for different employees.

   The employees are organised into a strict hierarchy, i.e. a tree T rooted at the CEO. There is one restriction, though, on the guest list to the retreat: due to successful union bargaining, it has been established that for every employee x, if x's immediate supervisor (parent in the tree) is *not* attending the retreat, then x *must* attend the retreat. Give an algorithm that finds the minimum total cost of hosting the retreat.

   Solution: By two simultaneous recursions.

   Subproblem(r): For the subtree $T(r)$, rooted at an employee r, find the cost $\text{INC}(r)$ of the cheapest retreat for the employees from $T(r)$ which must include

r, as well as the cost EXC(r) of the cheapest retreat which does not include r.

Recursion: let the children of r be $v_1, \dots, v_k$; then

$$\mathrm{EXC}(r) = \sum_{i=1}^{k} \mathrm{INC}(v_i);$$
$$\mathrm{INC}(r) = \sum_{i=1}^{k} \min(\mathrm{INC}(v_i), \mathrm{EXC}(v_i)).$$

Solution to the original problem: min(INC(ceo), EXC(ceo) where ceo is the CEO of the company (i.e., the root of the entire company employees tree T).

3. [**15 marks**] We say that a sequence of Roman letters A occurs as a subsequence of a sequence of Roman letters B if we can obtain A by deleting some of the symbols of B. Design an algorithm which for every two sequences A and B gives the number of different occurrences of A in B, i.e., the number of ways one can delete some of the symbols of B to get A. For example, the sequence ba has three occurrences in the sequence baba: **ba**ba, **b**ab**a**, ba**ba**.

   Solution: Let $l(A)$ and $l(B)$ be the length of sequence A (sequence B, respectively). For all $1 \leq i \leq l(A)$ and all $1 \leq j \leq l(B)$ we consider the following subproblems:

   Subproblem(i,j): "Find the number of ways $N(i, j)$ one can delete some of the symbols of the subsequence $B(1..j)$ to obtain the subsequence $A(1..i)$."

   Recursion:

$$N(i, j) = \begin{cases} N(i, j-1) & \text{if } \; A[i] \neq B[j] \\ N(i-1, j-1) + N(i, j-1) & \text{if } \; A[i] = B[j]; \end{cases}$$

   Thus, if $A[i] \neq B[j]$ then every way to obtain $A[1..i]$ from $B[1..j]$ will involve deleting $B[j]$; if $A[i] = B[j]$ then we have two options: we either match $A[i]$ to $B[j]$ and then delete symbols from $B[1..j-1]$ to obtain $A[1..i-1]$ or we first delete $B[j]$ and then delete symbols from $B[1..j-1]$ to obtain the entire subsequence $A[1..i]$.

   NOTE: The rest of the solutions were written by one of the tutors and might be done in a different style and using a different terminology then what you are used to; in particular, a "state" is just what we called a (generalized) subproblem. Also, the recursion usually proceeds in a "backward manner", for example by operating on suffixes of arrays rather than prefixes; this means that we recursively obtain a solution for a suffix $A[i..n]$ from the solution for the shorter suffix $A[i+1..n]$, but this does not make a big difference and you also get to see an alternative (well, somewhat odd) way of doing things. Sorry I really had no time to "translate" the rest of the solutions into the form you are more familiar with.

4. You are playing a game on an $n \times n$ grid of cells. Each cell contains a single integer: the cell in row r, column c contains the integer $A[r][c]$, where $A[1..n][1..n]$ is a two-dimensional array of integers. You start at the top-left cell $(1, 1)$ and will move to the bottom-right cell (n, n). At each step, you can only move to the cell *immediately* below or to the right. Specifically, from cell (r, c) you can move to $(r + 1, c)$ or $(r, c + 1)$ only. Your score is equal to the sum of the integers among all cells you have passed through: you would like to maximize this score.

   (a) **[10 marks]** Give an $O(n^2)$ algorithm that computes the maximum possible score.

   (b) **[5 marks]** Describe how to extend your algorithm from part (a) to also output any path that produces this maximum possible score. This path should be output as a series of `D` (down) and `R` moves, for instance `DDRDRR` is a valid path on a 4×4 grid. The overall time complexity of the algorithm should still be $O(n^2)$.

   (c) **[5 marks]** Suppose as input you are given the array $A[1..n][1..n]$ in *read-only* memory. This means that you can freely read its values, but cannot modify it in any way.

   Design an algorithm that runs in $O(n^2)$ time that outputs any path with maximum possible score, just as in part (b). However, your algorithm may only use $O(n\sqrt{n})$ additional memory on top of the input array. This additional memory can be freely read from or written to.

   Solution:

   (a) Our DP state corresponds to a cell (r, c) and asks, what is the maximum score achievable, assuming you start at this cell and end at (n, n). Our recurrence is as follows:

   - If we are at (r, c) we cannot move any further;
   - If $r = n$, we can only move right to $(r, c + 1)$;
   - If $c = n$, we can only move down to $(r + 1, c)$;
   - Otherwise, we have a choice of moving either right or down.

   In any case, we count the value of the current square. The recurrence can be written thus:

$$dp(r,c) = A[r][c] + \begin{cases} 0, & \text{if } r = n \text{ and } c = n \\ dp(r+1,c), & \text{if } r < n \text{ and } c = n \\ dp(r,c+1), & \text{if } r = n \text{ and } c < n \\ \max(dp(r+1,c), dp(r,c+1)), & \text{if } r < n \text{ and } c < n \end{cases}$$

Our final answer is then $dp(1,1)$. There are $O(n^2)$ states, and the recurrence is $O(1)$, giving and $O(n^2)$ time algorithm.

(b) Some possible answers, any will do:

- When we compute our DP from Part (a), we simply note which of the two directions gave us the better result. After we finish, we can then start at $(1,1)$, and follow the noted direction at each step.
- We first compute the DP from Part (a), and let $answer = dp(1,1)$. We start at $(1,1)$. Suppose at each step we are at (r,c). We must determine whether we should proceed down or right. If $r = n$ or $c = n$, our next move is forced. Otherwise, we pick whichever of $(r+1,c)$ and $(r,c+1)$ gives us a score of $answer - A[r][c]$. We then update $answer = answer - A[r][c]$, our position, and output either `D` or `R`, accordingly. We repeat this until we reach (n,n).
- As in the previous solution, except we simply compare which of down and right gives us a better answer.

Clearly the runtime is dominated by the DP (since constructing the path is done in $O(n)$, so the time complexity remains $O(n^2)$.

(c) Here are two solutions, one is more optimal than the other but both use at most $O(n\sqrt{n})$ additional memory.

The DP value is the answer $dp(r,c)$ we computed in part (a), which is the maximum score we can achieve if we start in (r,c) and finish at (n,n).

Both solutions require the observation that we can compute the maximum score attainable in $O(n^2)$ time and only $O(n)$ memory: this follows from the fact that we only need to keep two rows of DP values at a time, since we only ever refer to values in the current row or the next row.

- Let X be some positive integer. We perform our DP as in Part (a), keeping only two rows at at time. This uses $O(n)$ memory. However, for every Xth row (row $X, 2X, \ldots$), we will store in our additional memory the DP values for that row. This uses $O(n^2/X)$ memory and $O(n^2)$ time.

  We perform a second pass of DP, X rows at a time. This time, we store all the DP values we compute. We first repeat the same DP step for rows $[1..X-1]$, using the stored DP values for row X when we are computing row $X-1$. By using the technique in part (b), we can output the optimal path for the first X rows. This first block of X rows takes $O(nX)$ time to perform the DP and output the path.

  We then *discard* the memory used to perform this second pass on the first block, and repeat with the next block of rows $X, \ldots, 2X-1$. Thus, we use $O(nX + n^2/X)$ memory at any point in time. In total, we take $O(n^2)$ time for the original DP, and we spend $O(nX)$ time in the second pass for each of n/X blocks of X rows. Hence, we spend

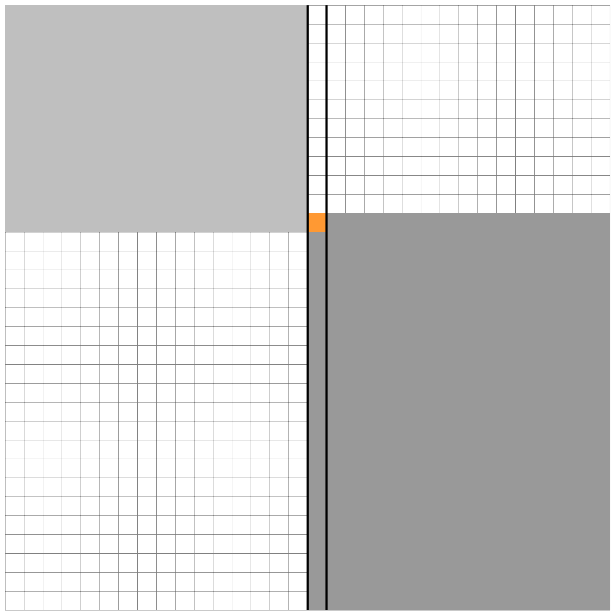

Figure 1: Divide and conquer to output the path. The orange square in the middle column lies on some optimal path. We recurse separately into the two gray regions, which together have roughly $n^2/2$ cells.

$O(n^2)$ time in total. If we choose $X = \sqrt{n}$, we see that this algorithm has $O(n\sqrt{n})$ memory usage, as required.

- There is a better solution, using only $O(n)$ extra memory. Suppose we have an identical DP, but instead of computing the maximum score of any down/right path to (n, n) from each square, it computes the maximum score of any up/left path to $(1, 1)$ from each square. It is easy to see this is an identical problem, so both can be solved in $O(n^2)$ with only $O(n)$ extra memory.

  Critically, we observe that a cell is on *some* optimal path from $(1, 1)$ to (n, n) if and only if the sum of these DP values, (ensuring the cell's value is counted precisely once) is equal to the *answer* computed in part (a).

  To output the path, we use divide-and-conquer over the columns. First, we perform our DP in both directions, storing the maximal score for each direction from every cell in column $n/2$. This takes $O(n^2)$ time, and $O(n)$ memory, particularly because we only wish to persist the results of a single column. We then select the *topmost* cell in this middle column that appears in some optimal path. This then separates the grid into quadrants, where we recurse on the top-left and bottom-right quadrants (see 4c).

  Recursively performing this process will give a list of cells on the path (there are only $2n + 1 = O(n)$ of these). Once we have these, we can finally output the desired path.

Since the total number of rows between the quadrants is still n, but the number of columns has halved, there remains (roughly) $n^2/2$ cells in total in the recurrence. Solving this recurrence gives a time complexity of $O(n^2)$. Since we only process one recursive call at a time, and each call uses at most $O(n)$ memory, our solution uses $O(n^2)$ time and $O(n)$ additional memory, which fits well within the $O(n\sqrt{n})$ bound required.

5. [**15 marks**] Assume that you are given a network flow graph G with a source s, a sink t and two other distinct vertices u and v. Design an algorithm which returns a smallest capacity cut among all cuts for which the vertex u is in the same side of the cut as the source s and vertex v is in the same side as the sink t and which runs in polynomial time.

   Solution

   We modify the flow network by adding two edges: $s \to u$ and $v \to t$, both of *infinite capacity*. We then run our standard Max Flow - Min-Cut Algorithm on the modified network, finding a minimum $s - t$ cut. Another option is to use a supersource connected to s and u and a supersink connected to t and v by edges of infinite capacity.

6. (EXTRA CREDIT!) At a certain theme park there are n rides, numbered from 1 to n. Before you can go on ride i you must pay its cost $C[i]$ dollars, plus a deposit, $D[i]$ dollars. This means you must have at least $C[i] + D[i]$ dollars to go on the ride. Once you have finished the ride, $D[i]$ dollars are returned to you, as long as you did not damage the ride!

   Careful Cheryl never damages any rides, and currently has T dollars on her. Cheryl would like to know the maximum number of *different* rides she can go on. She can go on rides in *any order* she chooses.

   You may assume $C[1..n]$ and $D[1..n]$ are both positive integer arrays of length n.

   (a) [**5 marks**] Prove that, if an amount of money T is enough for Cheryl to go on some subset S of all the rides in some order, then she can go on the same subset S of rides in a non-increasing order of deposit $D[i]$ (breaking ties arbitrarily).

   (b) [**10 marks**] Hence, design an $O(n \log n + nT)$ algorithm to determine the maximum number of different rides Cheryl can go on.

   (c) [**2 marks**] Does an $O(n \log n + nT)$ algorithm for this problem run in polynomial time? Why or why not?

   (d) [**5 marks**] Design an algorithm for the same task, but which runs in time $O(n^2)$.

Solution:

(a) First, consider the case when $n = 2$ and suppose that $D[1] < D[2]$ and that Cheryl can go on ride 1 then ride 2, in that order, starting with T dollars. We know that $T \geq C[1] + D[1]$ (since Cheryl can go on the first ride), and that $T - C[1] \geq C[2] + D[2]$ (since she can go on the second ride after the first).

Since all costs are positive, we know that $T \geq C[2] + D[2]$. Additionally, since $D[1] < D[2]$, we know that $T - C[1] \geq C[2] + D[1]$, so $T - C[2] \geq C[1] + D[1]$. Hence, Cheryl would also be able to go on ride 2 then ride 1.

We now extend this idea for sets of rides of arbitrary size. Suppose that $S = \{s_1, \ldots, s_k\}$ and that Cheryl is able to go on all of these rides in the order $s_1, \ldots, s_k$. Suppose that there is some $1 \leq j < k$ so that $D[s_j] < D[s_{j+1}]$: one always exists if $D[s_1], \ldots, D[s_k]$ is not in non-increasing order. Then, based on our previous argument, we can see that Cheryl would also be able to go on the rides in the order $s_1, \ldots, s_{j-1}, s_{j+1}, s_j, s_{j+2}, \ldots, s_k$. By Bubble Sort, we can repeat this process until we have ordered the rides in non-increasing order of deposit, with Cheryl still being able to go on all the rides at each step. This gives the result we require.

(b) The property proven in (a) gives rise to the following algorithm. Sort all the rides in non-decreasing order of deposit (breaking ties arbitrarily), and assume that the rides are renumbered 1 to n in this order.

Then, we have the following DP: our state is some suffix $[i..n]$ of rides in this order, and the amount of money we have left (an integer t between 0 and T, inclusive). For each state, we seek the maximum number of rides in this suffix we can take, asserting that we may only take rides in this prescribed order.

If we have an empty suffix of rides, we can go on 0 rides: this is our base case. Otherwise, if we choose whether or not to go on the ride, noting that if our current amount of money is less than the cost plus the deposit of the ride, we cannot go on it. The recurrence can be written thus:

$$dp(i,t) = \begin{cases} 0, & \text{if } i = n+1 \\ dp(i+1,t), & \text{if } t < C[i] + D[i] \\ \max(dp(i+1,t), dp(i+1,t-C[i]) + 1), & \text{if } t \geq C[i] + D[i] \end{cases}$$

The final answer is $dp(1,T)$. Sorting takes $O(n \log n)$ time. Our DP has $O(nT)$ states and an $O(1)$ recurrence, giving an $O(n \log n + nT)$ algorithm, as required.

(c) No. The value of T is proportional to $\log_2 X$, where X is the size of the input, rather than X itself.

Optional: Hence, this algorithm runs in pseudo-polynomial time, but exponential time to the size of the input.

(d) We keep the sorted order we used in Part (b) and observe that as the amount of money we have increases, our answer (the maximum number of rides we can take in the order) cannot decrease.

We formulate a different, but similar DP: our state is a suffix $[i..n]$ of rides in the order, and the number of the remaining rides g we intend to take (our goal). For each state, we compute the minimum amount of money we need to achieve our goal.

Our base case is when our suffix is empty. If we have 0 remaining rides, then we require 0 dollars, otherwise, our goal is impossible, and so we require ∞ dollars. Otherwise, we have a choice. If we take this ride we require the larger of:

- The cost of this ride plus its deposit
- The cost of this ride plus the minimum amount required to fulfill the remaining rides.

Alternatively, we can reject this ride. This gives the following recurrence:

$$dp(i,g) = \begin{cases} 0, & \text{if } i = n+1 \text{ and } g = 0 \\ \infty, & \text{if } i = n+1 \text{ and } g > 0 \\ \min(\max(C[i]+D[i], C[i]+dp(i+1,g-1)), dp(i+1,g)), \\ & \text{if } i < n+1 \end{cases}$$

To find our final answer, we simply find the largest g such that $dp(1,g) \leq T$: this is the maximum number of rides we can go on among all n rides that we have enough money for.

Sorting takes $O(n \log n)$. Our DP has $O(n^2)$ states, and an $O(1)$ recurrence so the time complexity of our algorithm is $O(n^2)$, as required.