CS 577-2: Introduction to Algorithms Fall 2020

Solutions to the Midterm 1 Practice Problems

Instructor: Dieter van Melkebeek

Problem 1

We are going to use divide-and-conquer. We need to determine how to divide the problem into some number of subproblems. We can do this by dividing the set of points into two equal-size subsets based on their x-coordinates. The first subset will contain points with x-coordinates less than or equal to median, and the other will contain points with x-coordinates greater than the median. Call these sets P_L and P_R. We will recursively compute the set of undominated points for each of these subsets and then combine these two subsets in such a way that we end up with the set of undominated points for the whole set.

Let the two computed undominated sets be U_L and U_R. When combining the two subsets, the key insight is that all elements in U_R have x-coordinates greater than all elements in U_L, which gives us information about which elements of each set should end up in U, the complete set of undominated points:

- Points in U_R are not dominated by any points in U_L, since their x-coordinates are all greater than the x-coordinates of all the elements in U_L.
- A point in U_L is not dominated by any point in U_R if and only if its y-coordinate is greater than the maximum y-coordinate in U_R.

Since we can calculate the maximum y-coordinate of U_R in a linear scan, we can combine the solutions of the subproblems to produce the complete solution U in linear time.

Algorithm Our algorithm makes use of the following procedure, which can be implemented in linear time using the linear-time algorithm for finding the median.

SplitHalves(P)

Input: A nonempty set of points P
Output: Two sets P_L and P_R which partition P, and where:

- $|P_L| = \lfloor |P|/2 \rfloor$
- $|P_R| = \lceil |P|/2 \rceil$
- $(x, y) \in P_L, (x', y') \in P_R \Rightarrow x < x'$

The complete algorithm is as follows:

Correctness We will argue correctness by induction on the size of P.

Claim 1. *Given a nonempty set of points P, the algorithm returns the set of undominated points in P.*

Algorithm 2 UNDOMINATED(P)

Input: A nonempty set of points P
Output: The set of undominated points in P

```
1: procedure UNDOMINATED(P)
2:     if |P| = 1 then
3:         Let (x, y) be the sole element in P
4:         return (P, y)
5:     else
6:         (P_L, P_R) ← split P into halves
7:         (U_L) ← UNDOMINATED(P_L)
8:         (U_R) ← UNDOMINATED(P_R)
9:         U ← U_R
10:        maxy_R ← compute the largest y-coordinate of a point in U_R
11:        for each element (x, y) in U_L do
12:            if y > maxy_R then
13:                add (x, y) to U
14:        return (U)
```

Proof. By induction on the size of P.

If $|P| = 1$, then the sole element is undominated.

Now assume the claim is true when $|P| \leq n$. Let P be a set of size $n+1$. We make two recursive calls with sets of size $\lceil \frac{n+1}{2} \rceil \leq n$ and $\lfloor \frac{n+1}{2} \rfloor \leq n$. By the induction hypothesis, these recursive calls return the set of undominated points for their respective sets. The set of undominated points is a subset of $U_L \cup U_R$, and a point in U_L (resp. U_R) is dominated by an element of P if and only if it is dominated by an element of U_R (resp. U_L). Every point in U_R has x-coordinate larger than every point in U_L, so no point in U_R is dominated by a point in U_L. A point in U_L is dominated by a point in U_R if and only if its y-coordinate is at most the largest y-coordinate of a point in U_R. It follows that U contains precisely the undominated points of P. □

Complexity We use the recursion tree method. Each non-leaf node in the recursion tree has two children, and the input size n shrinks by half with each level of recursion, so the tree has the same shape as MERGESORT. At any internal node in the tree, the non-recursive work done is the call to SPLITHALVES plus some linear amount of work. The leaves do constant work. As in the analysis of MERGESORT, it follows that the total work done is $O(n \log n)$.

Note that since our algorithm takes $O(n \log n)$ time anyways, we could have sorted the points by their x-coordinate at the beginning of our algorithm and then split our points into their two subsets directly without using linear-time selection as in SPLITHALVES.

Problem 2

Let $G = (V, E)$ be the given graph where V contains the n intersections and E the m road segments. Consider the digraph $G' = (V', E')$, where V' contains two copies of V. We include in E' all road segments between the vertices of the first copy oriented in the uphill direction, and similarly all road segments between the vertices of the second copy oriented in the downhill direction. Moreover,

we introduce a "bridge" edge from every vertex in the first copy to its corresponding vertex in the second copy. The bridge edges have weight zero, and are meant to provide the option of switching from uphill to downhill at any intersection. Paths in G from home to school that go uphill and then downhill exactly correspond to paths in G' that go from the home vertex h in the first copy to the school vertex s in the second copy, and corresponding paths have the same length in G and in G'. In this way, we have recast the original problem into finding the shortest path in a digraph.

For digraphs without weights on the edges, finding a shortest path can be done in linear time with a breadth-first search. In G', however, the edges are weighted, so breadth-first search won't work. Instead, we observe that G' is actually *acyclic* (i.e., G' is a DAG). Indeed, every path through G' consists of a (possibly empty) sequence of steps in the first copy of G, possibly a step to the second copy of G, and then possibly some steps there. Within the first copy of G, the elevation increases with each step, so a cycle cannot be formed; similarly, the elevation decreases with each step in the second copy of G. So there can be no cycles at all.

The acyclicity permits us to compute shortest paths in linear time using dynamic programming. We first find an order of the vertices $v_1, \ldots, v_{2n}$ so that edges only exist from "left to right", i.e., from some v_i to some v_j where $i < j$. Such an order is called a topological order, and can be found in linear time using topological sort.[1]

Next, for each v_i, we define $\mathrm{OPT}(v_i)$ to be the shortest length of a path from v_i to s. We are interested in $\mathrm{OPT}(h)$. For the base case, suppose that s is v_r. For each $i > r$, $\mathrm{OPT}(v_i) = +\infty$, since there are no paths back to s. $\mathrm{OPT}(s) = 0$, clearly. For other vertices ($i < r$), $\mathrm{OPT}(v_i)$ is the minimum of $\ell(v_i, v_j) + \mathrm{OPT}(v_j)$ over all v_j such that there is an edge from v_i to v_j, and $\ell(v_i, v_j)$ is the weight of that edge. Note that such edges only exist for $j > i$, so this recurrence is well-defined.

Such a recurrence can be implemented recursively with memoization. The local work to compute $\mathrm{OPT}(v)$ is bounded by a constant plus the number of edges that leave v. It follows that the total work done is linear in the size of G' (recall from single-source shortest path done in class that this can be implemented in $O(n + m)$ time). Combined with the work to compute the topological sort (linear), the overall running time is linear.

Problem 3

Since both players are playing optimally, Alice will choose the move that would maximize her *overall* gain, and Bob on the other hand, will choose the move that would minimize Alice's *overall* gain.

Let $A(i, j)$ be the optimal gain for Alice when it is Alice's turn and the numbers remaining are $a_i, a_{i+1}, \ldots, a_j$. Here Alice only has two choices: (1) pick a_i or (2) pick a_j. Alice will choose the one with larger overall gain.

(1) Alice picks a_i: Bob will have $a_{i+1}, a_{i+2}, \ldots, a_j$ to choose from. Here Bob also has two choices: pick a_{i+1} or a_j. If Bob chooses a_{i+1}, then Alice's gain would be $A(i+2, j)$; and if Bob chooses a_j, Alice's gain would be $A(i+1, j-1)$. Bob will choose the one that could minimize Alice's gain, and thus Alice will get the minimum of $A(i+2, j)$ and $A(i+1, j-1)$ because of Bob's strategy, thus yielding an overall gain of

$$a_i + \min\{A(i+2, j), A(i+1, j-1)\}$$

[1] See section 6 of the scribe notes on graph primitives.

in this case.

(2) Alice picks a_j: by a similar argument Alice would achieve an overall gain of

$$a_j + \min\{A(i, j-2), A(i+1, j-1)\}.$$

Between the two choices (1) and (2), Alice will choose the larger one. Therefore, we have

$$\begin{aligned} A(i,j) = \max\{&a_i + \min\{A(i+2, j), A(i+1, j-1)\}, \\ &a_j + \min\{A(i, j-2), A(i+1, j-1)\}\}. \end{aligned}$$

The base cases are $A(i,i) = a_i$ and $A(i, i+1) = \max\{a_i, a_{i+1}\}$, where Alice only has 1 or 2 numbers to choose from. One can use an $n \times n$ table to compute all values of $A(i,j)$ and return $A(1,n)$ as the final answer for Alice in time $O(n^2)$. The final gain for Bob will then be

$$\sum_{i=1}^{n} a_i - A(1,n).$$

Note the space complexity can be further reduced to $O(n)$ by only keeping the previous two diagonals.

Problem 4

Let a_i indicate the i-th number, and $\circ_{i,i+1}$ be the operator between the i-th and the $(i+1)$-th number. For $i \leq j$, define $OPT(i,j)$ to be the set of numbers that can be produced from the subexpression from the i-th number to the j-th number modulo m. We want to know if 0 is in $OPT(1,n)$.

We can compute $OPT(i,j)$ using the following recurrence. For two sets of numbers modulo m, M_1 and M_2, and operator $\circ$, define $M_1 \circ M_2$ to be a new set consisting of $(m_1 \circ m_2 \bmod m)$ for each choice of m_1 in M_1 and m_2 in M_2. Then we have

$$OPT(i,j) = \begin{cases} \{a_i \bmod m\} & i = j \\ \bigcup_{i \leq k < j} OPT(i,k) \circ_{k,k+1} OPT(k+1,j) & i < j \end{cases} \tag{1}$$

Time and Space Analysis There are $O(n^2)$ subproblems to solve. For each subproblem, we need to compute the results of $O(n)$ pairs of sets. Each set in a pair contains at most m numbers, so for fixed k, computing $OPT(i,k) \circ_{k,k+1} OPT(k+1,j)$ takes no more than $O(m^2)$ time.[2] The whole algorithm therefore takes $O(n^3 m^2)$ time to compute, which is polynomial.

As for space, there are $O(n^2)$ unique subproblems. Each subproblem takes $O(m)$ space because it stores all possible values for the subexpression modulo m. So our space complexity is $O(n^2 m)$. We cannot improve our space complexity by using the iterative approach and "forgetting" about part of our OPT table along the way because we look back at all split points k for $i \leq k < j$.

[2] There are data structures that allow for this step to be faster, but this analysis is sufficient for this problem.

Solution Retrieval Option 1 In order to retrieve our solution (the ordering of operations such that the result is a multiple of m) recall that we have access to our 2D OPT table. Now we can walk through the array from $OPT(1, n)$ to our base cases, seeing which choice we would have made at each step. For starters, if $OPT(1, n)$ does not include 0, then we can say that there was no possible ordering of operations that resulted in a multiple of m. Otherwise, we will find out our last operation (as well as the values modulo m that the left and right subexpressions need to evaluate to) by looking through all of the options considered in our recurrence. Once we find the last operation, we then recursively compute the last operations used in the left and right subexpressions to yield their values modulo m, continuing until we reach our base cases.

In the general case that we are trying to compute the last operation used for the subexpression from the i-th number to the j-th number such that the subexpression evaluates to r, this means considering all operations $\circ_{k,k+1}$ for $i \leq k < j$ as well as all values in the sets $OPT(i, k)$ and $OPT(k + 1, j)$. Specifically, we need to find a value k such that there exists an $x \in OPT(i, k)$ and a $y \in OPT(k + 1, j)$ where $(x) \circ_{k,k+1} (y) = r$ modulo m. Once we have found such a value k, we can add $\circ_{k,k+1}$ to our ordering of operations. We then compute the last operation used for the subexpression from the i-th number to the k-th number such that the subexpression evaluates to x, and we compute the last operation used for the subexpression from the $(k+1)$-st number to the j-th number such that the subexpression evaluates to y.

The key difference between calculating the time complexity for retrieving the optimal ordering and the time complexity for finding if an ordering is possible is the number of subexpressions we consider. We don't have to consider a subexpression for all $O(n^2)$ pairs $OPT(i, j)$, we only have to consider the subexpressions on the path to our optimal solution. Think of a recursion tree analysis in which each node is a subexpression we consider. The top node corresponds to $OPT(1, n)$ and the n leaf nodes correspond to $OPT(i, i)$. At each level of the recursion tree, we have split our expression into some number of segments. Within each segment, we try a number of split points equal to the length of the segment. So the total number of split points considered at a given level of the recursion tree is $O(n)$. For each split point, the left and right sets each contain at most m numbers, so computing $(x) \circ_{k,k+1} (y)$ takes $O(m^2)$. In all, we do $O(nm^2)$ work per level and there are at most n levels, giving us a time complexity for retrieving the solution of $O(n^2m^2)$.

Solution Retrieval Option 2 Although the time complexity for the retrieval performed above does not affect the overall complexity, we can speed up our approach by keeping track of more information during the computation of the OPT table and using an appropriate data structure. In the cell $OPT(i, j)$ we could store not only the set of possible result values for a subexpression but also the values of k, x, and y that realized each possible result value (k being the split point, x the value of the left subexpression, and y the value of the right subexpression).

In this case, when we walk through the OPT table to retrieve our solution, we only need to look through the possible values for a subexpression, find the one that we want, and say that we used the operator corresponding to $\circ_{k,k+1}$, continuing on to investigate how we could have yielded x from the subexpression $i...k$ and y from the subexpression $(k+1)...j$ in the same manner. This way, we need to access no more than n cells of the OPT table. In each cell we just need to locate the information for the desired result value. If we represent the set of possible values by a characteristic array of length m, we can locate the information in time $O(1)$. Thus, he running time for retrieval becomes $O(n)$. Note that this approach requires the same number of cells for the OPT table, but each cell contains more memory space: $O(\log n + 2 \log m)$ bits instead of 1 bit.