CS 577-2: Introduction to Algorithms Fall 2019

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 undominating 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.

So if we can keep track of the maximum y-coordinate of U_R, we can combine the solutions of the subproblems to produce the complete solution U in linear time. We will facilitate this by having each recursive call of the algorithm keep track of the maximum y-coordinate as well as the set of undominated points.

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.

Algorithm 1 SPLIT(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:

Algorithm 2 UNDOMINATED(P)

Input: A nonempty set of points P

Output: The set of undominated points in P, along with the maximum y-coordinate of the 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, maxy_L) ← UNDOMINATED(P_L)
8:          (U_R, maxy_R) ← UNDOMINATED(P_R)
9:          U ← U_R
10:         for each element (x, y) in U_L do
11:             if y > maxy_R then
12:                 add (x, y) to U
13:         Compute the largest y-coordinate of a point in U; call it maxy
14:         return (U, maxy)
```

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 and the maximum y-coordinate of the undominated points in P.*

Proof. By induction on the size of P.

If $|P| = 1$, then the sole element is undominated and the element's y-coordinate is the maximum undominated y-coordinate.

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. This implies that $maxy$ is the largest y-coordinate among the undominated points. □

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 SPLIT 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 $O(n \log n)$.

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'. 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

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

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

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

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

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

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.

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