CPSC 413: Assignment 1 Solutions

1. (a) *Show that if $f = O(g)$, then $g = \Omega(f)$.*

   If $f = O(g)$, then there is some n_0 and $c > 0$ such that for all $n \geq n_0$, $f(n) \leq cg(n)$. Dividing both sides by c then gives $g(n) \geq \frac{1}{c} f(n)$ for all $n_0 \geq n$, so $g = \Omega(f)$.

   (b) *Recall that $n! = n(n-1)(n-2)\ldots(2)(1)$. Show that $n! = \Omega(2^n)$.*

   Consider

$$\begin{aligned} 2^n &= 2 \cdot 2 \cdot 2 \ldots 2 \cdot 2 \cdot 2 \cdot 2 \text{ (n times)}, \\ &= 2 \cdot 2 \cdot 2 \ldots 4 \cdot 2 \cdot 2, \text{ (n-1 numbers)} \\ &\leq n \cdot n-1 \cdot n-2 \ldots 4 \cdot 3 \cdot 2 \\ &= n! \text{ (so long as } n \geq 4). \end{aligned}$$

   Thus, for $n \geq 4$, $2^n \leq n!$, so $n! = \Omega(2^n)$.

   (c) *How does $n!$ compare to n^n? Which is Big-O of the either? Prove your claim.*

   We claim that $n! = O(n^n)$. Indeed,

$$n! = n(n-1)(n-2)\ldots(3)(2)(1) \leq n(n)(n)\ldots(n)(n)(n) = n^n$$

   so $n! = O(n^n)$.

2. *A toy company has n different toys $t_1, \ldots, t_n$ it could make, each with an associated profit P_i. The company has m machines, and each toy has an associated machine M_i on which it needs to be made. Their next shipment can hold $k \leq n$ toys. Each machine can only make one toy before they must be shipped out. Give an $O(n \log n)$ algorithm that determines which toys the company should make so as to maximize their total profit.*

   Our algorithm does the following:

   - sorts the toys by profit, with the highest profit first,

- adds the highest profit toy, keeping track of which machine it is made on,
- looks for the next highest profit toy which has not been made on a previous machine, adding it,
- continue until k toys have been assigned.

Sorting the toys takes $O(n \log n)$, and then assigning the toys $O(n)$, so the total running time is $O(n \log n)$.

To show that this solution is optimal, let S be any choice of k toys different than our solution A. Let i be the first point where the solutions differ. Then, by definition of our algorithm, the profit of A_i must be at least as large as the profit of S_i, since they have the same set of toys they can select from. Thus, exchanging A_i for S_i will lead to a solution that is as least as good as the original solution S. Thus, if S is optimal, A must be as well.

3. *Over the past several weeks, a bank has noticed that someone is attempting to hack into their computer system. They have n different times when they believe someone was trying to change their data: times t_i. However, this time is approximate, so for each time t_i, they only know that the incident occured somewhere between $t_i - e_i$ and $t_i + e_i$. Now, the company has found someone they believe is responsible, and have a record of when they were accessing the system, at times $x_1, x_2, \ldots x_n$. There are many thousands of these incidents, and the company needs to find out if each access x_i from this individual matches up with a unique incident t_j. So, they need to match each x_i to some unique span of time $[t_j - e_j, t_j + e_j]$. Give an algorithm that can determine if such a match exists in $O(n^2)$ time.*

We first mention two possible greedy algorithms which do not work. One possibility is to order the t_i, order the x_i, then simply see if the first x_i matches up with the first t_i, and so on. Return false if they do not match up in this order. Consider an example where $t_1 = 6, t_2 = 5$, $e_1 = 1$, $e_2 = 10$, $x_1 = 1$, $x_2 = 6$. In this case, the algorithm returns false, since the earliest t_i does not match with the earliest x_i. However, there is a solution: match x_1 with t_2 and x_2 with t_1. So this algorithm is incorrect.

Another possibility is to try and match each x_i with its closest t_i. This also does not always return a correct output. Consider the example where $t_1 = 6, t_2 = 5$, $e_1 = 10$, $e_2 = 1$, $x_1 = 1$, $x_2 = 6$. Matching the closest x_i with the closest t_i tries to match x_2 with t_1, but then cannot match x_1 with t_2. However, there is a solution: x_1 with t_1, and x_2 with t_2.

A greedy algorithm which does work is to go through x_i's in order, assigning it to an interval which overlaps it for which $t_j + e_j$ is minimized. Let us write T_j for the interval $[t_j - e_j, t_j + e_j]$. Then our algorithm is:

- order the x_i's,
- let I be the set of all intervals,
- for each x_i,
  - find the intervals in I which overlap x_i ($O(n)$ time),
  - if none exist, return false,
  - otherwise, find the interval T_j for which $t_j + e_j$ is minimized (breaking ties arbitrarily, $O(n)$ time),
  - remove T_j from the list I,
- endfor,
- return true.

Note that the ordering of x_i's takes $O(n \log n)$ time, and the for loop runs at most n times, with $O(n)$ operations in each loop, so the total cost of the for loop is $O(n^2)$. Thus, the total running time is $O(n^2)$.

To determine correctness, first note that if our algorithm returns correct, then it has found a solution, as the only time it assigns intervals to a point is if they overlap that point.

Conversely, we must show that if there is a solution, then our algorithm finds it. Suppose there is a solution S which assigns every point to some intervals. We will use an exchange argument to show that

S can be modified so as to still be a solution, while also giving the same intervals and points as our algorithm. Suppose that x_i is the first time our algorithm A does something different than S (this might be the first point). If A returns false at this point, then there must be no interval left that overlaps x_i. But this is a contradiction, since S is a solution, so there must be some interval remaining that overlaps x_i.

On the other hand, A could simply assign a different interval to x_i than S does. Suppose A assigns x_i interval T_j, while S assigns it interval T_k. We want to change S so as to assign x_i the interval T_j, but we now must find a point to assign to interval T_k. Since S is a solution, there is some other point x_l which it assigns to T_j, so x_l is in T_j. Since S is the same as A before x_i, $x_i \leq x_l$. Moreover, by definition of our algorithm, we know $t_j + e_j \leq t_k + e_k$, and x_i is in the interval T_k. Thus, we have

$$t_k - e_k \leq x_i \leq x_l \leq t_j + e_j \leq t_k + e_k.$$

Thus, x_l is in the interval T_k, so we can assign x_l to T_k and still have a valid solution.

Thus, if there is a solution S, then we can modify S until it runs the same as A. Thus if there is a solution, then A returns correct.

4. *A number of developers are hoping to develop the land at the edge of a circular lake. Each has requested to develop some portion of the edge of the lake. Design an algorithm that runs in polynomial time and returns the largest set of development requests so that no two overlap.*

Clearly, this problem is quite similar to the interval scheduling problem we did in class (call this the linear scheduling problem). The only difference is that there may not be a point at which there are no overlapping intervals. Thus, to modify this problem, we consider n different instances of the linear scheduling problem, and take the best of those.

Consider the following algorithm:

- for each request n,

- remove all requests that overlap with the starting point of that request, giving a new set of requets R_n,
- the set R_n has no overlap at that starting point, so it is an instance of the linear interval scheduling problem. Find the solution to this in $O(n \log n)$ time, giving a number of requests t_n.
- Return the maximum of the t_n's.

Since we run the linear scheduling problem algorithm n times, the total running time is $O(n^2 \log n)$.

To check correctness, let O be an optimal solution, and let I_k be some request in this optimal solution. Since O is a solution, no interval in O overlaps the starting point of I_k. Thus, O is also a solution to the problem R_k. By definition of our algorithm, our algorithm takes the maximum of the solutions to each R_k, so our algorithm is as least as good as this optimal solution, and is thus itself optimal.

5. *The police are looking for houses which have particularly large electricity consumption. To simplify the problem, imagine that they are investigating houses which are laid out on an $n \times n$ grid. Each house on the grid has some electricity consumption, $e(i, j)$. The police consider the house suspicious if it has electricity consumption equal to or greater than each of its vertical and horizontal neighbours. Design an algorithm that runs in $O(n)$ time and returns the location of a suspicious house.*

This is a difficult problem. I'll build up to the solution by first considering two ideas which do not work (but intuitively work well) then modifying the second idea to produce a correct solution. If you just want to skip to the solution without explanation, it is in the middle of page 8.

Let us call a suspicious house a "local maximum" to emphasize that a suspicious house is larger than or equal to each of its neighbours. One solution (which does not run in $O(n)$ time) is to start at an arbitrary point in the grid (say, in the corner), and compare that electricity consumption to all houses adjacent to it. If it is at least as large as all adjacent points, then it is a local maximum, and we are done. Otherwise, there must be some house adjacent to it with strictly greater energy consumption. Move to that house, and test all its neighbours. Continuing in this way, our algorithm must terminate, since each house considered has electricity consumption strictly greater than the last, and we only have a finite number of houses to look through. Thus, this algorithm will eventually terminate, and when it does, it has found a local maximum. Call this algorithm "local search". The only problem with local search is that it in particular cases, it may have to consider most of the $O(n^2)$ points in the grid. However, we want to run in $O(n)$ time.

Instead, we must find a way to focus on one of the four $n/2 \times n/2$ quarters of the grid, and find a solution there. Since we are looking for a total run time of $O(n)$, we can spend $O(n)$ time determining which of the four of these subregions to consider, since, as we showed in class, an algorithm that looks at one subproblem of size $n/2$ while only spending $O(n)$ time before and after the recursion has a total running time of $O(n)$. We will give an idea of how to do this, show that there is a small problem with the idea, then modify the idea to give a correct solution.

The idea is the following: we begin by finding the maximum value in the middle column of the grid. Call this point v. We then test the left and right neighbours of v. If they are both less than or equal to v,then v is a local maximum, and we are done. Otherwise, there is some house to the left or right with strictly greater energy consumption; call this point m.

We claim that there is a local maximum somewhere in the half of the grid containing m. Indeed, imagine running the local search starting from m. Since $m > v$, v has the largest value in the middle column, and the local search always finds houses with strictly larger consumption, a local search from m will never consider values in the middle column. So, the local maximum it finds is somewhere in the half containing m.

Thus, we may as well restrict our attention to the half of the grid containing m. Now consider all the houses in the middle row of this half, and suppose w has the largest consumption of these houses. There are two cases.

If $m > w$, then we claim there is a local maximum somewhere in the quarter containing m. Again, if we ran the local search from m, then since the local search always takes larger values and $m > v, w$ (which are the maximums in their column/row) we would never consider values in the middle row or middle column. So, there is a local maximum for the whole grid somewhere in this quarter; we recursively run the algorithm on this quarter.

The other case is when $w \geq m$. In this case, we check the two houses above and below w. If both of these are less than or equal to w, w is a local maximum and we are done. Otherwise, there is some point k adjacent to w where $k > w$. Running a local search from k would never consider the middle row or middle column, since $k > w \geq m > v$, and w and v are the maximum in the middle row/column. Thus, there is a local maximum for the grid somewhere in this quarter; we recusively run the algorithm on this quarter.

Unfortunately, there is a subtle problem with the above idea. Suppose one of our recursive calls returned a value that was a local maximum for its subregion. There is no guarantee that this local maximum is actually a local maximum for the entire grid - if the local maximum for the subregion was on the border of the subregion, then there may be a point outside the subregion, but adjacent to this point, that is greater than this point.

However, by the arguments above, we still know that there is a local maximum somewhere in the subregion that *is* a local maximum for the whole grid. The problem is that our algorithm may return "false" local maximums that are not this local maximum. We need some way to exclude these false local maximums from consideration.

Now, in both of the above cases, just before our recursive call, we had a point p in the subregion which was greater than all values on the boundary of the subregion (in the first case, it was m, in the second case, it was k). Thus, what we need to do, in addition to passing the $n/2 \times n/2$ grid to our recusive call, is also pass this point p and its value to the recursive procedure. Now, we know that there is a local maximum somewhere in this subregion that has value at least p. So, when we test the maximum value in the middle column or middle row, we first check whether this maximum is at least as large as p. If it is not, we simply ignore it, and take the half which does contain p, since we now know there is a local maximum there. If it is at least as large as p and is a local maximum in this subgrid, then we can safely return it as a local maximum of the entire grid, since this value is as large as p, which is larger than other values on the boundary of the subgrid with the whole grid. Finally, if it is at least as large as p and is not a local maximum, we proceed in the direction which is larger, and are guaranteed that a search from that point will never cross the line or any other boundary in the entire grid.

The algorithm, in full, is as follows. We actually pass both the point p and its value seperately, to deal with the initial case.

- LocalMax(Input $n \times n$ grid G, point p in the grid, value B)
- let v be a maximum value in the middle column ($O(n)$ time),
- if $e(v) < B$, (then this maximum might produce a false result, so we are going to ignore it)
  - choose the half of the grid with p in it,
  - let w be a maximum value in the middle row of this half,
  - if $e(w) < B$ run LocalMax on the quarter containing B, along with p and B,
  - if $e(w) \geq B$ and w is a local maximum, return w,
  - if $e(w) \geq B$ and w is not a local maximum, choose the quarter with a greater point m, run LocalMax on this quarter, along with m and $e(m)$,
- if $e(v) \geq B$ and v is a local maximum, return v (since it is larger than B, it is a true local maximum for the whole grid),

- if $e(v) \geq B$ and v is not a local maximum,
  - find a larger adjacent point k,
  - choose the half of the grid with k in it,
  - let w be a maximum value in the middle row of this half,
  - if $e(w) < k$ run LocalMax on the quarter containing k, along with k and $e(k)$,
  - if $e(w) \geq B$ and w is a local maximum, return w,
  - if $e(w) \geq B$ and w is not a local maximum, choose the quarter with greater point l, run LocalMax on this quarter, along with l and $e(l)$.

To initialize the program, we simply run LocalMax on the entire grid G, passing in an arbitrary point (say, the top left corner), with value $B = -\infty$. In this way, we are ensured that any points we consider on the first run through the algorithm are larger than B.

To prove that this algorithm is correct, let G_0 be the grid G, G_i be the subgrid after the ith iteration of the algorithm, and p_i be the point chosen in the ith iteration to send to the next iteration.

Claim: For any $n \geq 0$, $e(p_n)$ is larger than the values on the boundary of grid G with the subgrid G_n.

Proof: by induction on n. For $n = 0$, $G_n = G$, so there is no boundary and nothing to prove. Now assume the result is true for n, and consider the point p_{n+1}. By the algorithm, p_{n+1} is chosen so that it is larger than any values on newly-created boundaries. Moreover, also by the algorithm, p_{n+1} is at least as large as p_n, which is, by the induction assumption, larger than any values on the other boundaries. Thus, p_{n+1} is larger than all boundaries with G.

Finally, we can prove that this result returns a local maximum. By definition of the algorithm, if x is returned at stage n, then $x \geq p_n$, and x is a local maximum in the subgrid G_n. By the claim, p_n is strictly larger than any values on the boundary of G_n with G, so x is larger than any values on the boundary of G_n with G, and so is a local maximum in the entire grid.

As mentioned at the beginning, at each stage, the algorithm considers a single subproblem of size $n/2$, and takes $O(n)$ time to determine which subproblem to consider. As done in class, such a divide and conquer algorithm runs in $O(n)$ time.