Name: Jahan Kuruvilla Cherian

Discussion 1D
4:00pm – 6:00pm
TA: Arman

CS 180 – Homework 4

Exercise 1 – Page 246

Since we are only allowed to query the database, and we need to complete the algorithm in logarithmic time, we can make start by trying an approach similar to a binary search wherein we eliminate halves of each database per query.

We can start by comparing the two medians of the databases. If $m = \left\lceil\frac{n}{2}\right\rceil$ then *A(m)* and *B(m)* are the two medians of the respective databases. If *A(m) < B(m)* then by the definition of the median – the middle value in a data set – we can say that *B(m)* is greater than the first *m* elements in A and that *B(m)* is greater than the m-1 elements in itself, making it the $2m^{th}$ element in the combined dataset. Thus let us focus only on [0,k] elements in B and call it ***B***. If we perform a similar analysis on *A* noticing that the first $\left\lfloor\frac{n}{2}\right\rfloor$ elements of A are less than *B(m)* and thus less than the last *n-m+1* elements of *B*, and are less than the last $\left\lceil\frac{n}{2}\right\rceil$ elements of *A* which means they are by definition less than at least $n - m + 1 + \left\lceil\frac{n}{2}\right\rceil = n + 1$ elements of the combined database. This means that these lower elements of *A* are less than the median and so we can ignore them focusing only on the upper half ***A*** – $[\left\lfloor\frac{n}{2}\right\rfloor + 1, n]$.

Since we ignore the $\left\lfloor\frac{n}{2}\right\rfloor$ elements less than the median and same number of elements greater than the median, and so the median of the remaining set is the same as the median of the original set of elements, and so we can repeat this process recursively removing halves of each side of the two databases until we get to the combined median of the two. We can define the algorithm as follows:

```
Def median(n, astart, bstart):
    mid = $\left\lceil\frac{n}{2}\right\rceil$
    If n == 1 then return min(A(astart+mid), B(bstart+mid))
    If A(astart+mid) < B(bstart + mid):
        Return median(mid, astart+$\left\lfloor\frac{n}{2}\right\rfloor$, bstart)
    Else:
```

Return median(mid, astart, bstart + $\left\lfloor \frac{n}{2} \right\rfloor$*)*

We start from the starts of both databases as astart and bstart = 0. Because we are essentially dividing the overall database into halves each time and conquering those datasets, we see that the number of queries to evaluate the function is logarithmic. That is if $q(n)$is the number of queries, then $q(n) = q\left(\left\lceil \frac{n}{2} \right\rceil\right) + 2 \to 2\lceil \log(n) \rceil$.

And thus we have a logarithmic algorithmic to find the median of two combined databases of unique elements using divide and conquer recursively.

Exercise 4 – Page 247

Based on the given equations, we can define a convolution to generate a simpler equation as follows, we take one vector $v_i = (q_1, q_2, \dots, q_n)$ and $v_j = (\frac{1}{n^2}, \frac{1}{(n-1)^2}, \dots, 1, 0, -1, \dots, -\frac{1}{n^2})$ where we see that the difference of $(j - i)$ can take on a maximum of n and a minimum of -n, if j is n and i is 0, and the latter for vice versa. Thus we get the convolution of the form: $\sum_{i<j} \frac{q_i}{(j-i)^2} + \sum_{i>j} \frac{-q_i}{(j-i)^2}$. Once we calculate this convolution we just multiply by Cq_j to get F_j, which is an O(n) process for all j terms. Now we show the convolution computation is O(nlogn) as shown in the book.

Essentially there are two ways to represent polynomials. We have a coefficient representation as $A(x) = a_0 + a_1x + a_2x^2 + \dots + a_{n-1}x^{n-1}$. However, the convolution of two such representations would use O(n^2), but it's evaluation would be O(n) using Horner's method. Then we have the point-value representation using the fundamental theorem of algebra by Gauss, where a degree n polynomial with complex coefficients has n complex roots, which leads to a corollary that a degree n-1 polynomial A(x) is specified by evaluation at n distinct values of x giving us $A(x) = (x_0, y_0), \dots, (x_{n-1}, y_{n-1})$ in two dimensional space. The multiplication of this would be O(n) but its evaluation would require Lagrange's formula which is an O(n^2) operation. Thus we want a tradeoff between the two. The conversion between the two methods just use a matrix multiplication which is still too expensive. So what we say is that we can instead break our polynomial up by dividing and conquering into odd and even sets such that $A(x)_{even} = a_0 + a_2x^2 + a_4x^4 + \dots + a_{2n}x^{2n}\ and\ A(x)_{odd} = a_1x + a_3x^3 + \dots + a_{2n-1}x^{2n-1} \to A(x) = A_{even}(x^2) + A_{odd}(x^2)$. The Discrete Fourier Transform states the key ides that given a polynomial $A(x) = a_0 + a_1x + a_2x^2 + \dots + a_{n-1}x^{n-1}$, we can evaluate it at n distinct points $x_0, \dots, x_{n-1};\ where\ x_k = \omega^k$ where ω is the nth root of unity which is a complex number $x\ such\ that\ x^n = 1$. Thus the Fast Fourier Transform states that we can evaluate at the roots of unity $\omega^0, \omega^1, \dots, \omega^{n-1}$. So the divide is breaking it up into odd and even functions and then the conquer is evaluating at the $\frac{1}{2}n^{th}$ roots of unity which thus evaluates a $n-1$ degree polynomial in O(nlogn) steps, thus solving the convolution in O(nlogn).

Exercise 4 – Page 315

a.) If the moving cost *M = 10*, int[] NY = {1, 4, 1} and int[] SF = {20, 1, 20} as contiguous arrays, then the optimal plan would be sum([NY, NY, NY]) + M = 6 since M would be 0 for no move, but the greedy algorithm given would be sum([NY, SF, NY]) + M = 13.

b.) Any plan wherein the cost of living exceeds the moving cost*3 and is interleaved between the two cities as in the following: M = 10, int[] NY = {1, 50, 1, 50} and int[] SF = {50, 1, 50, 1} then to have the optimal plan we would have to move each time the cost is about to hit 50 that is [NY, SF, NY, SF] which will lead to a cost of 34. If no move was made then the cost would be at least >= 51, which is clearly not optimal.

c.) If the optimal plan ends in NY then it will pay $N_n + \min(opt_N(n-1), opt_S(n-1) + M)$ and similar for SF except with the roles reversed $S_n + \min(opt_s(n-1), opt_N(n-1) + M$. Basically saying that we know a plan will end in either one of those cities and so we have the cost as the cost of the last month and the cost of the previous n-1 months in that same city or in the different city with the moving cost, thus following a top down dynamic programming approach. Thus we have the following linear runtime (O(n)) algorithm:

```
def optimalPlan(n, M, nList, SList):
        let $opt_N(0)$, $opt_S(0)$ = 0,0
        For i in range n:
                $opt_N(i) = N_i + \min(opt_N(i-1), opt_S(i-1) + M)$
                $opt_S(i) = S_i + \min(opt_s(i-1), opt_N(i-1) + M$
        return min($opt_N(n)$, $opt_S(n)$)
```

Exercise 8 – Page 319

a.) If we just use a modified version of the example from the book as follows:

i	*1*	*2*	*3*	*4*
x_i	1	10	10	2
f(i)	1	2	4	8

The algorithm would set off the EMP at times 2 and 4 leaving 4 robots destroyed, whereas activating at 3 and 4 would leave 5 destroyed, showing the algorithm is not the most optimal.

b.) If the input of robots ends at some x_j *in a sequence* $x_1, \ldots, x_j$ then it is best to activate the EMP at the best solution upto some point j or not. If we define a optimal function $opt(j,k)$ which is the best solution for the j steps through to n given the EMP has been charging for *k* steps, then we have the following: $opt(j,k) = \max(\min\left(x_j, f(k)\right) + opt(j+1,1), opt(j+1,k+1))$. Thus we have the following:

```
def scheduleEMP(x_1, …, x_j):
    let opt(n,k) = $\min\left(x_j, f(k)\right)$
    for j in reversed range n:
        for k in range n:
            $opt(j,k) = \max(\min\left(x_j, f(k)\right) + opt(j+1,1), opt(j+1,k+1))$
    return opt(1,1)
```

The runtime for this algorithm is O(1) per entry leading to a O(n^2) algorithm and is a top down dynamic programming approach.

Problem 5 from website

We start by looking at the very base case that is of n = 0, that is there are no lines. We define our output R_n to correspond to the number of regions based on n lines. Thus in the base case we have only 1 region by definition that is the plane itself. If we have 1 line then we have 2 regions. If we now look at the *nth* line then the number of regions increases by *r* if and only if the line splits *r* of the old regions. We can split *r* of the old regions if and only if the *nth* line hits the existing lines on the plane in *r-1* points. Since two non-parallel straight lines can intersect at most one point, so the new line can intersect the n-1 old ones in at most n-1 different points, thus $r \le n \rightarrow R_n \le R_{n-1} + n$. But we also see that it is possible to place this line such that it is not parallel to any other line and thus intersects all n-1 lines and it does not go through any existing intersection point thus $R_n \geq R_{n-1} + n$. Thus we have the following recurrence $R_n = R_{n-1} + n$. Using this relation let us assume $R_n = \frac{n(n+1)}{2} + 1$ and then proceed to prove by induction.

Base Cases:

$$R_0 = \frac{0(0+1)}{2} + 1 = 1$$

$$R_1 = \frac{1(1+1)}{2} + 1 = 2$$

$$R_2 = \frac{2(2+1)}{2} + 1 = 4$$

Induction Hypothesis:

Let us assume that $R_k = \frac{k(k+1)}{2} + 1$ gives us the number of regions produced by k lines, and so let us try to prove this true for k+1.

Induction Step:

$$R_{k+1} = R_k + k + 1 = \frac{k(k+1)}{2} + k + 1 = \frac{k(k+1) + 2k + 1}{2} + 1 = \frac{k^2 + 3k + 2}{2} + 1$$
$$= \frac{(k+1)(k+2)}{2} + 1 = \frac{(k+1)((k+1)+1)}{2} + 1$$

Thus by assuming the case to be true for *k* lines we show that the case is true for *k+1* lines, and in accordance with our base cases, by the principle of induction we have shown that the number of regions produced by n lines in an infinite plane is given by $R_n = \frac{n(n+1)}{2} + 1 \; \forall n \in \mathbb{N}$.