CS218 Design and analysis of algorithms **Jan-Apr 2023**

Exercises (No submission)

Lecture 1 (Jan 3) Binary search and variants

- Given two sorted integer arrays (with all distinct numbers in them) of size n, we want to find the median of the union of the two arrays. Can you find it by accessing only $O(\log n)$ entries in the two arrays.
- Given an array of n integers and an integer S, find a pair of integers in the array whose sum is S. Can you do it in $O(n \log n)$ time?
- Let $f\colon \mathbb{R} \to \mathbb{R}$ be a convex function that is promised to have a minimizing point. The function $f(x)$ and its derivative $f'(x)$ are not given explicitly, but via oracle access. That is, you can give any point $x \in \mathbb{R}$ and the oracle will give the values of $f(x)$ and $f'(x)$. How many queries do you need you find the point minimizing $f(x)$?
- Given an integer a, check if it is of the form b^k for some (unknown) integers b and $k > 1$. Can you do this in time $O(\log^c a)$ for some constant c?
- You are given a list of landholdings, say $a_1, a_2, \ldots, a_n$ (in acres, can be zero). We want to give land to everyone who has less than f acres and bring them up to f acres. For this, we need to take away land from everyone who has more than c acres, bring them down to c acres, and redistribute the obtained land. (1) What is the highest value of f that can be feasible? (2) For a chosen value of f, find the right value of c?
- We saw two algorithms for finding square root of an integer, Babylonian method (Newton-Raphson method) and Binary search. Which one do you think is faster?

Lecture 2 (Jan 6) Reducing to a subproblem, divide and conquer

- Suppose there is a trader for a particular commodity, whose license allows them to buy it only once and sell it only once during a season. Naturally, the commodity can be sold only after it is bought. The price of the commodity fluctuates every day. The trader wants to compute the maximum profit they could have made in the last season.

  Design an $O(n)$-time algorithm, where the input is the list of prices $\{p_1, p_2, \ldots, p_n\}$ for the n days in the last season and the output is the maximum possible profit. Assume addition, subtraction, comparison etc are unit cost. Sample input: Prices: 70, 100, 140, 40, 60, 90, 120, 30, 60.

  Output: Max profit: 80
- **Celebrity.** There is a party with n people, among them there is 1 celebrity. A celebrity is someone who is known to everyone, but she does not know anyone. You need to identify the celebrity by only asking the following kind of queries: ask the ith person if they know the j person. Can you do this in $O(n)$ queries?
- **Faster Fibonacci.** Assuming unit cost multiplications/additions, can you compute the nth Fibonacci number in $O(\log n)$ operations? Try to convert the problem into computing nth power of a 2×2 matrix.
- **Non-dominated points.** For the problem of computing non-dominated set of points, suppose as the first step, we sort the points in increasing order of x-coordinates. After sorting, can you find the non-dominated points in $O(n)$ time. Write a pseudocode.

- **Coverage by posters.** You have a rectangular notice board where each day a new rectangular poster appears (possibly overlapping the older ones). Each poster has its top edge sitting on the top edge of the notice board. You want to compute the **total area** of the notice board hidden by the posters.

  Let the top left corner of the notice board be $(0, 0)$ and the bottom right corner be (p, q). Let the i-th poster have its four corners located at $(\alpha_i, 0), (\beta_i, 0), (\alpha_i, d_i), (\beta_i, d_i)$. The notice board currently has n posters and you are given numbers α_i, β_i, d_i for each $1 \leq i \leq n$. Can you compute the area of the notice board hidden by the posters in $O(n \log n)$ time?

  Hint: Try divide and conquer. Remember that you might need to compute more information in the subproblem than what is actually asked.

- **Majority Fingerprints.** You are given a collection of n fingerprints. You are also told that more than $(n/2)$ of these are identical to each other. You are only given access to an equality test, which takes two fingerprints and tells whether they are identical or not. Using this equality test, can you find out the one with more than $n/2$ copies in $O(n \log n)$ time?

- **Significant inversions.** In an array A of integers, a pair is called a significant inversion if $i < j$ and $A[i] > 2A[j]$. Design an $O(n \log n)$ time algorithm to find the number of significant inversions.

Lecture 3 (Jan 10) Integer Multiplication

- Let us try to apply the divide and conquer approach on the integer multiplication problem. Suppose we want to multiply two n-bit integers a and b. Write them as

  $$a = a_1 2^{n/2} + a_0$$

  $$b = b_1 2^{n/2} + b_0$$

  The product of the two integers can be written as

  $$ab = a_1 b_1 2^n + (a_1 b_0 + a_0 b_1) 2^{n/2} + a_0 b_0.$$

  Can you compute these three terms $a_1 b_1$, $a_1 b_0 + a_0 b_1$, $a_0 b_0$, using only **three multiplications of** $n/2$ **bit integers** and a few additions/subtractions? If yes, then we will get an $O(n^{1.58})$ time algorithm.

- Can you find square of an n-bit integer a, using squaring subroutine on **five** $n/3$ bit integers and a few additions/subtractions? What's the running time you get?

- Can you find square of an n-bit integer a, using squaring subroutine on **2k-1** integers with n/k bits and a some additions/subtractions? What's the running time you get? What if you take k as something like $n/2$? Does that give you a really fast algorithm?

- **Matrix Multiplication.** Let us say we have 8 numbers $a_1, a_2, a_3, a_4, b_1, b_2, b_3, b_4$ and we consider these **seven** expressions.

  $$p_1 = (a_1 + a_4)(b_1 + b_4), \quad p_2 = (a_3 + a_4)b_1, \quad p_3 = a_1(b_2 - b_4), \quad p_4 = a_4(b_3 - b_1)$$
  $$p_5 = (a_1 + a_2)b_4, \quad p_6 = (a_3 - a_1)(b_1 + b_2), \quad p_7 = (a_2 - a_4)(b_3 + b_4)$$

  (a) Compute the following four sums. This will be helpful later.

  $$p_1 + p_4 - p_5 + p_7, \quad p_3 + p_5, \quad p_2 + p_4, \quad p_1 - p_2 + p_3 + p_6$$

  Now, we want to apply divide and conquer technique to matrix multiplication. Let A and B be two $n \times n$ matrices, and we want to compute their product $C = A \times B$. The naive algorithm for this will take $O(n^3)$ arithmetic operations. We want to significantly improve this using divide and conquer.

A natural way to split any matrix can be this:

$$A = \begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix},$$

where each A_i is an $n/2 \times n/2$ matrix.

(b) Can you express the product matrix C, in terms of A_1, A_2, A_3, A_4 and B_1, B_2, B_3, B_4.

(c) Design an algorithm for matrix multiplication using divide and conquer which takes $O(\mathbf{7}^{\log_2 n}) = O(n^{\log_2 7}) = O(n^{2.81})$ time.

Lecture 4 (Jan 13) Polynomial Multiplication/Convolution

- Can you apply Karatsuba's trick on polynomial multiplication and get a runtime bound better than $O(d^2)$ for degree d polynomials.
- Given probability distributions of two (discrete) random variables, we want to compute the probability distribution for the sum of the two random variables. Do you see how convolution can be used to compute this distribution.
- **Puzzle.** There is a resource that is shared by n people. We want that the resource should be accessible if and only if at least k out of the n people come together to access it. Can you device a scheme to distribute the password/secret key that achieves this? Hint: different representations of a polynomial.

Lecture 5 (Jan 17) Discrete Fourier transform

We saw an algorithm that evaluates a degree d polynomial on dth roots of unity and take $O(d \log d)$ time (also known as discrete Fourier transform). The plan was to use this for faster polynomial multiplication. Recall the high level idea of polynomial multiplication ($p(x) \times q(x)$):

- evaluate the two given polynomials over sufficiently many points, say $p(\alpha_1), p(\alpha_2), \ldots,$ and $q(\alpha_1), q(\alpha_2), \ldots,$
- do point-wise multiplication of the evaluations of the two polynomials, that is,

$$p(\alpha_1) \times q(\alpha_1), p(\alpha_2) \times q(\alpha_2), \ldots,$$

- the above are evaluations of the product polynomial. In the last step, we compute its coefficients from the evaluations.

We skipped this last part, given evaluations of a polynomial how to get back the coefficients? In other words, how to do inverse of discrete Fourier transform. It turns out that inverse can also be expressed as discrete Fourier transform. The below exercise will give the idea.

- Let $P(x) = a_0 + a_1 x + \cdots + a_{d-1} x^{d-1}$ be a degree $d-1$ polynomial. Let the dth roots of unity be $\omega^0, \omega^1, \ldots, \omega^{d-1}$. Let the evaluations of $P(x)$ on the dth roots of unity be $e_0, e_2, \ldots, e_{d-1}$. That is, for each $0 \leq i \leq d-1$

  $$e_i = P(\omega^i) = a_0 + a_1 \omega^i + \cdots + a_{d-1} \omega^{(d-1)i}.$$

  Define a new polynomial $Q(y)$ as $e_0 + e_1 x + \cdots + e_{d-1} x^{d-1}$. Prove that for each $0 \leq i \leq d-1$

  $$Q(\omega^{-i}) = e_0 + e_1 \omega^{-i} + \cdots + e_{d-1} \omega^{-(d-1)i} = d a_i.$$

  Hint: Useful facts to prove, $\omega^{-i} = \omega^{d-i}$, For any $1 \leq i \leq d-1$, $\sum_{j=0}^{d-1} \omega^{ij} = 0$.

- **Implementation of FFT.** It is possible to do an iterative implementation of FFT, which is "in-place". That is, we can just work on the input array of length d and need only constant size extra memory. Can you think about such an implementation?

4. You've been working with some physicists who need to study, as part of their experimental design, the interactions among large numbers of very small charged particles. Basically, their setup works as follows. They have an inert lattice structure, and they use this for placing charged particles at regular spacing along a straight line. Thus we can model their structure as consisting of the points $\{1, 2, 3, \ldots, n\}$ on the real line; and at each of these points j, they have a particle with charge q_j. (Each charge can be either positive or negative.)

They want to study the total force on each particle, by measuring it and then comparing it to a computational prediction. This computational part is where they need your help. The total net force on particle j, by Coulomb's Law, is equal to

$$F_j = \sum_{i<j} \frac{Cq_iq_j}{(j-i)^2} - \sum_{i>j} \frac{Cq_iq_j}{(j-i)^2}$$

They've written the following simple program to compute F_j for all j:

```
For j = 1, 2, ..., n
  Initialize F_j to 0
  For i = 1, 2, ..., n
    If i < j then
      Add (C q_i q_j)/((j - i)^2) to F_j
    Else if i > j then
      Add -(C q_i q_j)/((j - i)^2) to F_j
    Endif
  Endfor
  Output F_j
Endfor
```

It's not hard to analyze the running time of this program: each invocation of the inner loop, over i, takes $O(n)$ time, and this inner loop is invoked $O(n)$ times total, so the overall running time is $O(n^2)$.

The trouble is, for the large values of n they're working with, the program takes several minutes to run. On the other hand, their experimental setup is optimized so that they can throw down n particles, perform the measurements, and be ready to handle n more particles within a few seconds. So they'd really like it if there were a way to compute all the forces F_j much more quickly, so as to keep up with the rate of the experiment.

Help them out by designing an algorithm that computes all the forces F_j in $O(n \log n)$ time.

Figure 1: Kleinberg Tardos: Divide and Conquer Chapter 5, Exercise 4.

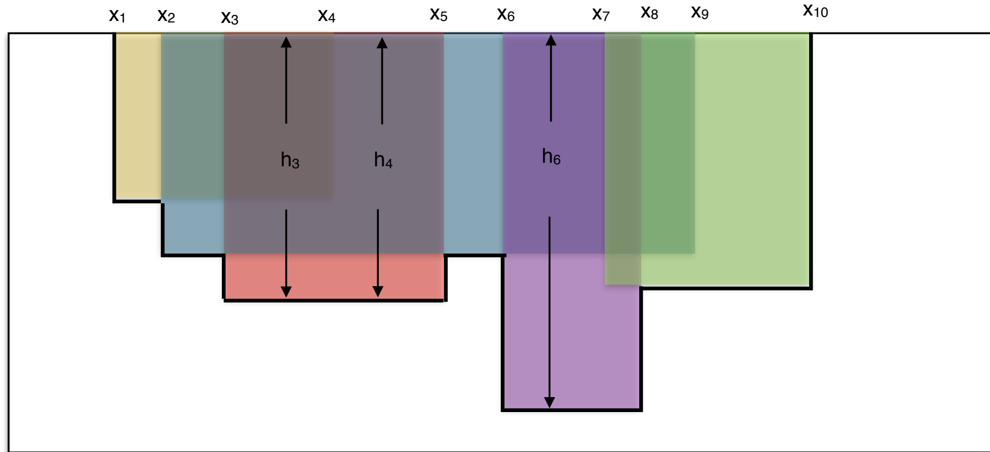

Figure 2: Showing area covered by 5 posters.

Lecture 6 (Jan 21) Problems discussion

Here is one more exercise on divide and conquer.

- Given n distinct points in 2-D with integer co-ordinates, we want to find all pairs of points which are separated by a distance of at most 2. Design an $O(n \log n)$ algorithm for this. Assume basic arithmetic operations can be done in constant time.

  Hint: Try divide and conquer. One possible way to divide the set of points into two parts: draw a verticle line and take left and right sides of that line. Remember that sometimes in the subproblem, you need to compute more stuff than what is asked for in the original problem.

We discussed the posters problem (listed in Lecture 2), for which multiple approaches were suggested to get time $O(n \log n)$. Below is the summary of four approaches that were suggested.

First, note that as you try to design a recursive algorithm, for any subproblem it will not be good enough to compute the value of the total area covered. You will need more information that that. For example, you can compute some description of the whole area that is covered by posters. One way to describe this area is a list of x-intervals and for each interval what is the height of covered area. For example, see Figure 2. We can represent the area covered as a list of pairs: $(x_1, h_1), (x_2, h_2), (x_3, h_3), \ldots,$. This means that between x_1 and x_2 the height covered is h_1, between x_2 and x_3 the height covered is h_2 and so on. Once we compute this representation, the area can be easily computed as $(x_2 - x_1) \times h_1 + (x_3 - x_2) \times h_2 + \cdots$.

- **Approach 1: divide and conquer.** Divide the set of posters into two sets arbitrarily. Suppose we have already computed the description of the areas covered by the two sets. Let these be in the form mentioned above as

  $$(a_1, h_1), (a_2, h_2), (a_3, h_3), \ldots, (a_k, h_k)$$

  and

  $$(b_1, g_1), (b_2, g_2), (b_3, g_3), \ldots, (b_\ell, g_\ell).$$

  Now, we need to *merge* these two to obtain the area covered by the union of the two sets. This will be similar to the merge part of mergesort.

  Let the merged list be L, which is empty initially. We maintain two variables ℓ_1 and ℓ_2 which indicate the height of the current intervals seen in the list 1 and list 2, respectively. Suppose the current pointers are in the two lists are at (a_j, h_j) and (b_i, g_i). Repeat the following till the lists are empty.

- if $a_j < b_j$ then we update ℓ_1 to h_j. Check whether $\ell_2 \leq h_j$. If yes, then we insert (a_j, h_j) in L. If no, that means ℓ_2 will be the height covered at $x = a_j$, hence, we will insert (a_j, ℓ_2) in L. Move the pointer ahead in list 1.
- the other case is similar. if $a_j \geq b_i$ then we update ℓ_2 to g_i. Check whether $\ell_1 \leq g_i$. If yes, then we insert (b_i, g_i) in L. If no, we will insert (b_i, ℓ_1) in L. Move the pointer ahead in list 2.

The time complexity here is $O(n \log n)$ just like mergesort.

- **Approach 2: sorting all x-coordinates.** Put all left and right end points of the posters into one array of length $2n$ and sort them in increasing order. With each end point, we also store the information of which poster it belongs to and whether it's a left or a right end point. The idea is to do sweep from left to right and for each x-coordinate x_0, find out what height that is covered at x_0. Note that the height covered is nothing but the maximum height of any poster that intersects with that $x = x_0$ line (i.e., $\alpha_i \leq x_0 \leq \beta_i$). We don't need to find this height for literally every x-coordinate, but for those where there is a change in the height.

  To compute the maximum height among current posters, we will need to maintain the set of current heights in a data structure S. As we go from left to right,

  - whenever we see a new poster, say we see the left end point α_j, then we insert height d_j into S. Re-compute the maximum among the heights in S, say it's d_{max}. Then add (α_j, d_{max}) on the output list.
  - whenever we leave a poster, say we see the right end point β_j, then we remove the height d_j from S. Re-compute the maximum among the heights in S, say it's d_{max}. Then add (β_j, d_{max}) on the output list.

  At the end, at the output list will be the representation of the covered area, as explained above. It is easy to compute the value of the area covered from there.

  Time complexity: Sorting takes $O(n \log n)$ time. At each iteration the main computations are to add or remove elements from the data structure and to compute the maximum. One can use a balanced binary tree data structure which allows all this operations in $O(\log n)$ time.

  Alternatively, as one student suggested, one can use a heap, in addition with some tricks. Note that heap does not allow removal of an arbitrary element but only of the maximum element. The suggestion was that we keep a Booelan array B where jth location indicates whether jth poster is gone or is still there. Whenever we see the right end point a poster, say β_j, we need to remove d_j. If d_j is equal to d_{max} then it works fine. But, when d_j is not the max then we just indicate in B that jth poster is gone, but do not actually remove it. Whenever we compute max of the current heights, we check B to see whether the poster with the max height is gone. If it is already gone, then we discard this max after removing and recompute the max. We continue doing that till we get a max such that the corresponding poster is still there. Time complexity remains $O(n \log n)$. While in one iteration, one might need to use the remove operation many times, note that a particular height is inserted and removed only once in the whole procedure.

- **Approach 3: decreasing order of heights.** Sort the posters in decreasing order of heights. Why we choose this order will be clear when one sees the idea. While we process the posters one by one, we will maintain the following information about the posters processed so far: for which all x-coordinates, a nonzero height has been covered by some poster. To be more precise, we will store a list L of x-intervals which is the union of the x-intervals of posters seen so far. For example, if we have seen four posters with

  $$(\alpha_1 = 5, \beta_1 = 9, d_1 = 6), (\alpha_2 = 7, \beta_2 = 10, d_2 = 4), (\alpha_3 = 1, \beta_3 = 4, d_3 = 3), (\alpha_3 = 13, \beta_3 = 14, d_3 = 2).$$

  Then L will be $(1, 4), (5, 10), (13, 14)$. We will also store the total area covered so far, say A.

  When a new poster arrives, say (α_j, β_j, d_j), we (i) compute the area added in the covered region and (ii) update the list L of x-intervals appropriately. Let's first discuss (i). Since the new poster's height

is smaller than all current posters, the new area added is precisely outside the x-intervals in L. We need to compute the total length of the overlap of intervals in L with (α_j, β_j) and subtract it from $\beta_j - \alpha_j$. Multiply this difference with d_j and add it to A. In step (ii), we should remove of all the intervals that have some overlap with (α_j, β_j) and add all of their union with (α_j, β_j).

Let's see an example, Say currently, we have the four posters as mentioned above. Now a fifth poster comes with $(\alpha_5 = 3, \beta_5 = 12, d_5 = 2)$. Note that intervals $(1, 4)$ and $(5, 10)$ have overlap with $(3, 12)$. Their union is simply $(1, 12)$. Hence, the new L will be $(1, 12), (13, 14)$. The total length of the overlap of $(1, 4)$ and $(5, 10)$ with $(3, 12)$ is $1 + 5 = 6$. We subtract this from $12 - 3 = 9$ and get 3. Hence, the new area added is $3 \times 2 = 6$.

Implementaion with an appropriate data structure One possibility is to store the list L of intervals in a balanced binary tree (with the natural order). When a new poster comes say with (α_j, β_j) we do the following: Initialize $N = \beta_j - \alpha_j$. find the first interval in L that overlaps with (α_j, β_j), say it is (p_1, q_1). Find the last interval in L that overlaps with α_j, β_j, say it is (p_2, q_2). Remove all the intervals from L between (p_1, q_1) and (p_2, q_2) (this can be done using the successor function). When we remove an interval, we subtract its length from N (in case of (p_1, q_1) or (p_2, q_2), only subtract the overlap). Finally insert the interval $(\min\{p_1, \alpha_j\}, \max\{q_2, \beta_j\})$ in L. The new area added will be $N \times d_j$.

Time complexity: Each insertion and deletion can be done in $O(\log n)$. There are total n insertions and hence, at most n deletions as well.

- **Approach 4: increasing order of α_js.** Let's sort the posters in increasing order of α_js (left endpoints). We will process the posters one by one and maintain a description of the area covered so far. The description of the covered area will in in the form

$$(x_1, h_1), (x_2, h_2), (x_3, h_3), \ldots,$$

This represents that between x_1 and x_2 the height covered is h_1, between x_2 and x_3 the height covered is h_2 and so on. When a new poster comes, say (α_j, β_j, d_j), we do the following:

  - Find the largest i such that $x_i \le \beta_j$.
  - Find the smallest k such that $x_k \ge \alpha_j$ and $h_k < d_j$.
  - Remove all pairs (x_ℓ, h_ℓ) from $\ell = k$ to $\ell = i$. If $h_{k-1} \ge d_j$ then insert (x_k, d_j), otherwise insert (α_j, d_j) at an appropriate place.

Again, to ensure $O(\log n)$ time insertion and deletions, one can use a balanced binary tree.