1 Divide-and-Conquer Convex Hull

Here's another random Divide-And-Conquer algorithm.

Recall, possibly from CS 2, that the convex hull $H(Q)$ of a set of points Q is the smallest convex set containing Q (that is to say that it is the intersection of all convex sets containing Q), where a convex set is any set C where for any $a, b \in C$, the line segment joining a and b is contained entirely in C.

There are a number of equivalent formulations for convexity; if C is a polygon, convexity is equivalent to all of its angles being 'right turns', in the sense that the cross product between the (correctly ordered) vectors which forms this angle is positive.

Also note that if Q is a finite set of points, any convex polygon H containing the points of Q, whose vertices are all points of Q, is necessarily the convex hull of Q, as any convex polygon contained strictly inside of this set would necessarily fail to include one of the vertices of H, and hence some point of Q.

There are a number of ways to compute the convex hull; here is a divide and conquer method.

1.1 Algorithm Idea

Like all divide-and-conquer algorithms, we can imagine dividing this into two subproblems and then merging the results. We can find the convex hull of the points on the left half of the plane and the convex hull of the points on the right half. Then, we can merge these two convex hulls in order to figure find the convex hull of the complete set (the method which we will use to do this will be outlined in the next section).

Figure 1: We begin by dividing the set into two parts.

1.2 Algorithm

```
ConvexHull(Q):
    Sort(Q, x) //Sort the points in Q by their x-coordinate
```

```
A = ConvexHull(Q[0:n/2]) //Find the convex hull of the left side
B = ConvexHull(Q[n/2+1:n]) //Find the convex hull of the right side

return merge(A, B)
```

1.3 Merging

We will let the ordered sequence of points which makes up A as $(a_1, a_2, \ldots a_m)$, and B as $(b_1, b_2, \ldots, b_l)$, where consecutive points in A are connected by an edge, i.e. (a_i, a_{i+1}) and (b_i, b_{i+1}) are connected as an edge for each i, and (a_m, a_1) and (b_1, b_l) are connected. We'll assume that these are stored as doubly linked lists, so we can easily delete and combine elements.

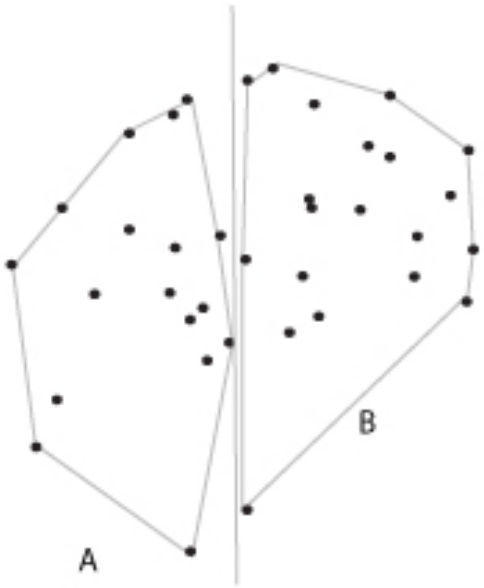

Figure 2: The two convex hulls of either side of the plane.

In order to merge two convex hulls, we will make the following observation: suppose we cyclically reorder the indices so that a_0 and b_0 are those points, in A and B respectively, so that their x-coordinates are closest together (these can be found immediately because our list is sorted by x-coordinate). The line joining them, (a_0, b_0), must be contained entirely in the convex hull, by the definition of convexity. However, this segment is currently not contained in $A \cup B$ because these are the closest points to the dividing line, and therefore, if there were other points which surround that line, they would have to be closer to the dividing line. Then, we can imagine connecting A and B via this line segment (a_0, b_0), i.e. let $H = A \cup B \cup (a_0, b_0)$.

H will not be convex in general, but we can iteratively make H more convex as follows: a polygon fails to be convex iff one of angles made by its edges, say $\angle a_{i+1} a_i, b_j$ or $\angle a_i b_j b_{j+1}$, are not 'right turns', in the sense that the cross products of the corresponding vectors are not positive. All of the angles in A and B are necessarily right turns as they are convex sets, so we have that the only angles which can be a left turn are (a_0, b_0, b_1) or (a_m, b_0, b_1).

In general, say $\angle a_i b_i b_{i+1}$ is a left turn, we can then remove b_i from the convex hull, and put in the edge (a_i, b_{i+1}). Consider the triangle $\Delta a_{i+1} b_{i+1} b_{i+2}$. Because the angle $\angle a_{i+1} b_{i+1} b_{i+2}$ was a left turn, all of the points on the interior of this triangle will be included in the interior of the convex hull. Therefore, this operation increases the set of points on the interior of the polygon, while simultaneously decreasing the number of vertices on the boundary. At any given point, as in the above paragraph, the only angles which can be left turns are the angles made by the segments which were just introduced.

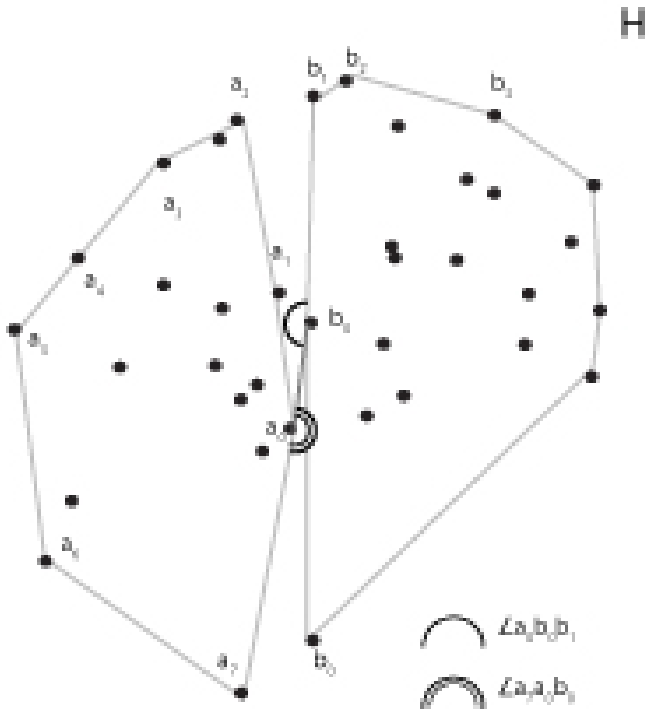

Figure 3: We've connected the convex hulls, but there's a problem.

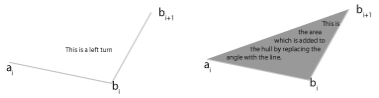

Figure 4: We can improve our output by replacing the angle with a segment.

Therefore, we can continue to work our way along the convex hull removing the left turn which was just introduced, until there are no more left turns, at which point our polygon will be convex. This will be our convex hull. We know that this will eventually terminate and we will obtain a convex set, as the number of vertices is constantly decreasing, and any 3 point set will be convex, so a convex set will be obtained in the first $n - 3$ iterations.

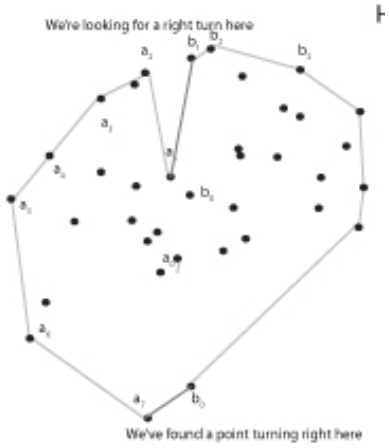

Figure 5: We can improve our output by replacing the angle with a segment.

Thus, we have the following algorithm:

```
merge(A, B):
    Let A[i] be the point in A closest to the dividing line
    Let B[j] be the point in B closest to the dividing line

    Let a_l = i, a_h = i, b_l = j, b_h = j

    While still_updating:
      If A[a_l-1], A[a_l], B[b_l] is a left turn:
        remove A[a_l] from A
      If A[a_l], B[b_l], B[b_l+1]  is a left turn:
        remove B[b_l] from B
      If A[a_h-1], A[a_h], B[b_h] is a left turn:
        remove A[a_h] from A
      If A[a_h], B[b_h], B[b_h+1]  is a left turn:
        remove B[b_h] from B
      return A + B
```

1.4 Runtime

We'll apply the master theorem.

We have that each iteration, we divide the work into two subcases, and perform the algorithm recursively on each one. Then, we merge them together by scanning through at most n vertices and removing at most n vertices, so that the total amount of work required on each iteration is $O(n)$. Therefore, we see that the total amount of time required by the algorithm after sorting satsfies

$$T(n) = 2T(\frac{n}{2}) + cn$$

The master theorem tells us that because $cn \leq (2/2)n$ for all n, the runtime also satisfies

$$T(n) \in \Theta(n)$$

Therefore, the runtime is actually dominated by the sorting at the beginning, which takes $\Theta(n \log(n))$

2 Streaming Algorithms

We've haven't really considered space complexity yet, but the idea shouldn't be too foreign to you. The idea is that the data set we are considering is of size n where n is so large that we can't store all of it. So, we're going to have process it part by part and remember only some auxillary information and not the data itself.

You have already seen a couple examples of streaming algorithms and you might not even know it. Common streaming algorithms are max, min, sum, product, average, standard deviation, etc. All of these have a principal characteristic that we are going to employ.

2.1 Formalism

Streaming Algorithm Property: Given a function $f : \Sigma^* \to \Gamma$, we can calculate $f(x_1 \dots x_n)$ as $g(x_n, n, f(x_1, \dots, x_{n-1}))$ for some function $g : \Sigma \times \mathbb{N} \times \Gamma \to \Gamma$. Note, this is not always the case, but a good starting point. Lets take average as an example. We write

$$f(x_1 \dots x_n) = \frac{x_1 + \dots + x_n}{n} \qquad g(x, n, s) = \frac{s(n-1) + x}{n} \tag{1}$$

Its easy to see that f, g satisfy the requirements above. This gives us a good generic algorithm framework:
Generalized Streaming Algorithm:

- Initialization: $m \leftarrow f(\emptyset)$
- Process x_j: $m \leftarrow g(x_i, j, m)$
- Output: m

2.2 Uniform Sampling

Problem. Given a stream (x_j) of unknown length over alphabet Σ, uniformly sample 1 element from the stream.

Solution.

$$f(x_1 \dots x_n) = \begin{cases} x_1 & \text{with prob.} 1/n \\ x_2 & \text{with prob.} 1/n \\ \vdots & \vdots \\ x_n & \text{with prob.} 1/n \end{cases} \qquad g(x, n, s) = \begin{cases} x & \text{with prob.} 1/n \\ s & \text{with prob.} (n-1)/n \end{cases} \tag{2}$$

Let's verify this follows the streaming algorithm property: With probability $1/n$, $g(x_n, n, f(x_1 \dots x_{n-1}))$ does equal x_n. And with probability $(n-1)/n$ it equals $f(x_1 \dots x_{n-1})$ which with probability $1/(n-1)$ equals each of $x_1, \dots, x_{n-1}$. $(n-1)/n \cdot 1/(n-1) = 1/n$ so $g(x_n, n, f(x_1 \dots x_{n-1}))$ equals each of $x_1, \dots, x_n$ with probability $1/n$. □

Note: Not all streaming functions $h : \Sigma^* \to X$ will satisfy the steaming property. Instead we might have to find a related function f that does satisfying the streaming property and then after completed processing the whole stream calculate h from f. You will see this on your problem set.

3 Fast Fourier Transform

There are A TON of ways of looking at the discrete Fourier transform: EE's might phrase it as changing from the time domain to the frequency domain; mathematicians might look at it as an irreducible representation of $\mathbb{Z}/n$ into $\mathbb{C}$. On the one hand, it's just a big matrix and a linear transformation, and on the other hand, it's a hugely powerful technique that is frequently used in a lot of different applications.

A simple way of looking at the DFT is as follows: we can look at a vector of complex numbers and go 'man this is literally just a list of numbers; how can I make sense of this?'. Well, mathematicians like to take vectors, which are in this sense just structureless lists, and turn them into polynomials. Associated to any vector v, where $v[k]$ is the kth element in v, we can write down a polynomial as follows:

$$p(x) = \sum_{k=0}^{n} v[k] x^k$$

Now, our list of numbers has some structure associated to it.

Now, we can think about a natural problem: how do we multiply these polynomials? Multiplying polynomials is an inherently important problem and one we will see that can have applications in positions that we hadn't previously expected.

Well, the naive algorithm of just doing out all of the additions and multiplications in the way that we learned in elementary school will take $\Omega(n^2)$ time, which isn't great. Can we do better?

Well, we can imagine what would happen if our polynomial were represented slightly differently, this might be easier. What if, instead of having the list of coefficients, we just had a list of values that the polynomial takes on? It turns out that if you specify what a degree n polynomial has to be in n positions, this uniquely determines what that polynomial is, so we can imagine flipping between these two representations interchangably.

But, now we have a good way of multiplying these polynomials: by just multiplying the values that it takes on at specified points, and then converting back to the natural way of doing this.

But how do we convert between these representations? Well, if we let ω be a 2^lth root of unity, then we can write down the values that it takes on at powers of ω as

$$p(\omega^m) = \sum_k v[k] \omega^{km}$$

If we write this down as a matrix, this is

$$\begin{pmatrix} p(1) \\ p(\omega) \\ \dots \\ p(\omega^{l-1}) \end{pmatrix} = \begin{pmatrix} 1 & 1 & \dots & 1 \\ \omega & \omega^2 & \dots & \omega^{l-1} \\ \omega^2 & \omega^4 & \dots & \omega^{2(l-1)} \\ & & \dots & \\ \omega^{l-1} & \omega^{2l-2} & \dots & \omega^{(l-1)^2} \end{pmatrix} \begin{pmatrix} v[1] \\ v[2] \\ \dots \\ v[l-1] \end{pmatrix}$$

Well, hey that's the Fourier Transform Matrix!

Now, we have a way of converting between the two representations of the polynomials. So now, we just have to have a way of computing it efficiently, and we can do polynomial multiplication efficiently.

3.1 FFT

Actually, there is a fairly natural way to compute this efficiently using the following matrix manipuation:

$$\begin{pmatrix} 1 & 1 & \dots & 1 \\ \omega & \omega^2 & \dots & \omega^{l-1} \\ \omega^2 & \omega^4 & \dots & \omega^{2(l-1)} \\ & & \dots & \\ \omega^{l-1} & \omega^{2l-2} & \dots & \omega^{(l-1)^2} \end{pmatrix}$$

$$= \left(\begin{array}{ccc|ccc} 1 & & & 1 & & \\ & 1 & & & \omega & \\ & & 1 & & & \omega^2 \\ & & \vdots & & \vdots & \\ & & 1 & & & \omega^{l/2-1} \\ \hline 1 & & & \omega^{l/2} & & \\ & 1 & & & \omega^{l/2+1} & \\ & & \vdots & & \vdots & \\ & & 1 & \dots & & \omega^{l} \end{array}\right) \begin{pmatrix} F_{l-1} & \\ & F_{l-1} \end{pmatrix} \left(\begin{array}{ccc|ccc} 1 & & & & & \\ & 1 & & & & \\ & & 1 & & & \\ & & & 1 & & \\ \hline & 1 & & & & \\ & & 1 & & & \\ & & & 1 & & \\ & & & & 1 & \end{array}\right)$$

I won't explain where this decomposition comes from, except to say that it basically is just splitting the Fourier Transform into the even index and odd index parts, but it is important to recognize that matrix decomposition implies already an algorithm for performing the FFT.

We can rewrite it in a slightly more suggestive way in block form:

$$= \left(\begin{array}{c|c} I & D(1, \omega, \dots \omega^{l/2-1}) \\ \hline I & D(\omega^{l/2}, \omega^{l/2+1}, \dots \omega^{l-1}) \end{array}\right) \begin{pmatrix} F_{l-1} & \\ & F_{l-1} \end{pmatrix} \begin{pmatrix} E \\ \hline O \end{pmatrix}$$

If we want to apply this big matrix to a vector v, what do we do? The first matrix on the right says to first split v into the coordinates with even and odd indices: E says that we move the even indexed entries to the top of the vector and O tells you to move the odd indexed entries to the bottom. The next matrix tells you to do the Fourier transform on each of these parts, and the last matrix tells you to multiply each of the odd entried terms by powers of ω, and then add it to the even entried terms to get the full Fourier transform.

3.2 Convolution

Now, from the above discussion, we have the most important fact about the DFT:

$$DFT(p(x)) * DFT(q(x)) = DFT(p(x) \otimes q(x))$$

Where $*$ denotes elementwise multiplication (easy), and $\otimes$ denotes convolution (hard), which is another way of saying polynomial multiplication.

4 Example: Cartesian Sum of Sets

Given 2 sets A and B of natural numbers, we might be interested in the numbers which can be represented as the sum of these sets; as a silly example, we might have a pair of mislabeled dice which have strange numbers on the side, and we want to know what numbers we can get from rolling them. Hence, we define the *Cartesian Sum* of A and B:

$$A + B = \{a + b | a \in A, b \in B\}$$

A naive solution would go through each pair of numbers in A and B, add them together, and add them to some sort of list. This is clearly an $O(n^2)$ solution, and it seems hard to see how we can do better. But, it turns out that if we make the assumption that the elements in A or B are no bigger than $O(n)$, for some constant n, we can use FFT's to do better.

4.1 Algorithm Idea

We can define *indicator* vectors v_A and v_B, each of length sn for both A and B, where

$$v_A[k] = \begin{cases} 1 & \text{if } k \in A \\ 0 & \text{if } k \notin A \end{cases}$$

so basically, we are going to characterize each set as a vector which indicates for each k whether or not k is in A. Similarly, we will define

$$v_B[k] = \begin{cases} 1 & \text{if } k \in B \\ 0 & \text{if } k \notin B \end{cases}$$

Now, we have representations for our sets, but how can we find out whether an element is in its sum? Well, we can use the trick we described above to turn these vectors into polynomials:

$$p_A(x) = \sum_{k=0} v_A[k]x^k \qquad\qquad p_B(x) = \sum_{k} v_B[k]x^k$$

These polynomials will have a x^k term iff $k \in A$.

Now, we can think about multiplying these polynomials

$$p_A(x)p_B(x) = \sum_k (\sum_l v_A[k-l]v_B[l])x^k$$

Here, we can see that the coefficient of the x^k term will be nonzero iff $k - l \in A$ and $k \in B$, for some $l < k$, i.e. if $k \in A + B$. So, if we can just multiply these polynomials (equivalently convolve these vectors), then we can just read off directly from the result whether $k \in A + B$.

Now, we know what to do: just do the FFT on the two indicator vectors, pairwise multiply the results, transform back to get the convolution, and then just read off directly the elements from the results.

4.2 Algorithm

```
Cart_Sum(v_A, v_B):
    V_A = FFT(v_A)
    V_B = FFT(v_B)


    for each k:
        V = V_A[k] * V_B[k]

    return FFT^-1(V)
```

4.3 Runtime

Since the vectors in question are of length $O(n)$, we have that each FFT will run in $O(n\log(n))$ time on these vectors. The element-wise multiplication then takes $O(n)$ multiplications, which each will take $O(1)$ time, as we are multiplying fixed size numbers. And finally, inverting the FFT also takes $O(n\log(n))$ time, so this entire algorithm takes $O(n\log(n))$ time. Equivalently, we could have just said that convolution, as a single operation takes $O(n\log(n))$ time.