15-451: Algorithms October 24, 2019

Lecture Notes: Quicksort and Random Incremental Convex Hull

Lecturer: Gary Miller *Scribe:*

[1]

1 Backwards analysis

In this section, we present the method of **backwards analysis** and show how it can simply the running time analyses of randomized algorithms.

1.1 Randomized quicksort

Recall the classic randomized algorithm for quicksort to sort an array M of numbers. (For simplicity, assume that all numbers are distinct.)

Algorithm `RandomizedQuicksort`

1. Pick a uniformly random element of the array a, called the *pivot*.
2. Let L and R be the elements in the array that are smaller and larger than a, respectively. Note that L and R can be constructed using $|M| - 1$ comparisons by comparing every other element to a.
3. Recursively apply quicksort to L and R.
4. Concatenate $L + a + R$ in that order, and output the result. Again, this can be done in linear time.

To analyze the running time of `RandomizedQuicksort`, we can focus on the expected number of comparisons:

Question 1.1. What is the expected number of comparisons of `RandomizedQuicksort`?

To answer this question, we can attempt a probabilistic argument as follows. Let $T(n)$ be the expected number of comparisons on an array of length n. (Make sure you understand why only the length of the array matters when determining the expected number of comparisons—at least, when all numbers are distinct.) With probability $1/n$, quicksort chooses the i'th smallest element of the list, resulting in L and R of sizes $i-1$ and $n-i$. Therefore, we have the following recurrence:

$$T(0) = 0,$$

$$T(n) = \sum_{i=1}^{n} \Pr[i\text{'th smallest element chosen}] \cdot (n-1+T(i-1)+T(n-i)) = \sum_{i=1}^{n} \frac{1}{n}(n-1+T(i-1)+T(n-i)).$$

There are multiple ways to solve this recurrence. In our case, we consider a different procedure with the exact same recurrence relation and which is easier to analyze.

[1] Originally 15-750 notes by Jason Li

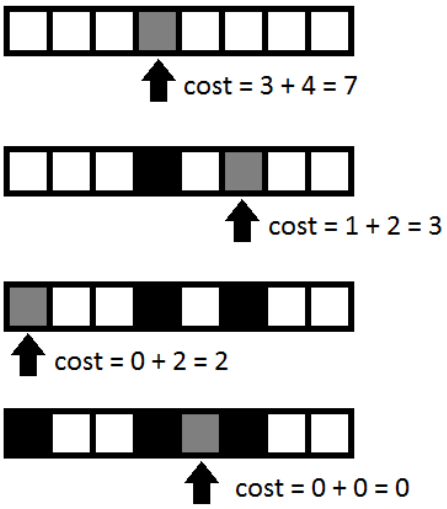

Figure 1: A partially completed dart game with $n = 8$.

1.2 A dart game: the same as quicksort?

Consider the following randomized *dart game* and its associated cost function:

Procedure `DartGame`

1. Initially. there is a dart board of n consecutive, empty squares, arranged in a row.
2. For n iterations, throw a dart at a uniformly random empty square, and pay cost equal to the number of *consecutive* empty squares to the left and right of the dart (see figure 1).

Observe that, once we throw our first dart, the empty segment to the left of the dart and the empty segment to the right can be treated as two separate, *independent* dart games: there may no longer be a dart hitting the left segment every round (since the dart can still hit the right segment), but *conditioned* on a dart hitting the left segment, the square it hits is still uniformly random. Therefore, if $C(n)$ be the expected cost of `DartGame` with n initial squares, then:

$$C(0) = 0,$$

$$C(n) = \sum_{i=1}^{n} \Pr[\text{dart hits } i\text{'th square}] \cdot (n-1+C(i-1)+C(n-i)) = \sum_{i=1}^{n} \frac{1}{n}(n-1+C(i-1)+C(n-i)).$$

Since this recurrence is identical to that of `RandomizedQuicksort`, we conclude the following:

Claim 1.2. *The expected number of comparisons of* `RandomizedQuicksort` *on n elements is the same as the expected cost of* `DartGame` *on n initial squares.*

1.3 Dart game using backwards analysis

Consider *reversing* the dart game as follows: start with a full board of marked squares, and unmark a random square each iteration, again paying cost equal to the number of consecutive empty squares to the left and right of the unmarked square.

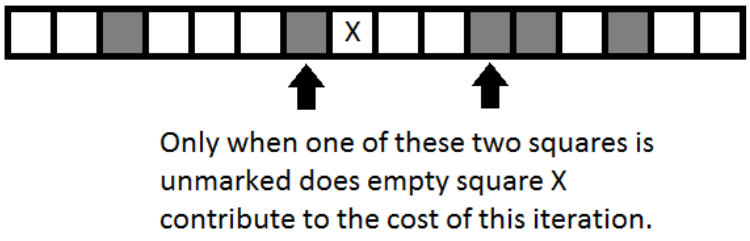

Figure 2: A reversed dart game with 5 darts remaining.

Claim 1.3. *The expected cost is the same for the original dart game and the reversed dart game.*

To see this, note that this procedure exactly captures the dart game backwards: for any specific sequence of chosen squares in the original game, reversing that sequence in the reversed dart game arrives at the same cost per round (and therefore, total cost). Both sequences (or, more precisely, permutations) occur with probability $1/n!$ in their respective games, so they contribute the same amount to the expected costs of each game.

Lastly, we analyze the expected running time of the reversed game. Consider an iteration when there are i darts left on the board. For each empty square, the probability that it contributes to the cost on this round is at most $2/i$ (see figure 2), so by linearity of expectation, the expected cost of this iteration is at most $(n-i) \cdot 2/i$. Summing over all i gives an expected running time of

$$\sum_{i=1}^{n}(n-i)\frac{2}{i} \leq 2n\sum_{i=1}^{n}\frac{1}{i} = 2nH_n = O(n\log n),$$

where $H_n \approx \ln n$ is the n'th harmonic number. Moving back to our original quicksort problem, we conclude that:

Claim 1.4. *The expected number of comparisons of* `RandomizedQuicksort` *is at most* $2nH_n = O(n\log n)$.

2 Convex hull

In this section, we provide an expected $O(n\log n)$ time randomized algorithm to compute the convex hull of n points in 2D, and then show a lower bound of $\Omega(n\log n)$ by reducing sorting to convex hull.

2.1 Definitions

Here, we review definitions related to the following discussions.

Definition 2.1. A set $A \subseteq \mathbb{R}^d$ is convex if $x, y \in A \rightarrow$ segment between x, y is completely contained by A. Alternatively, A is convex if A is closed under convex combination.

Definition 2.2. Convex closure of A, denoted as $CC(A)$, is the smallest convex set containing A.

Definition 2.3. Convex hull of A, denoted as $CH(A)$, is the boundary of $CC(A)$.

For $A \in \mathbb{R}^2$, $CH(A)$ is bounded by a simple closed path counterclockwise enclosing all elements in A. Segment $[a, b]$ is a side of $CH(A)$ if and only if two conditions are satisfied: (1) $a \neq b \in A$; (2) for all $a' \in A$, either a' is left of $[a, b]$ or $a' \in [a, b]$.

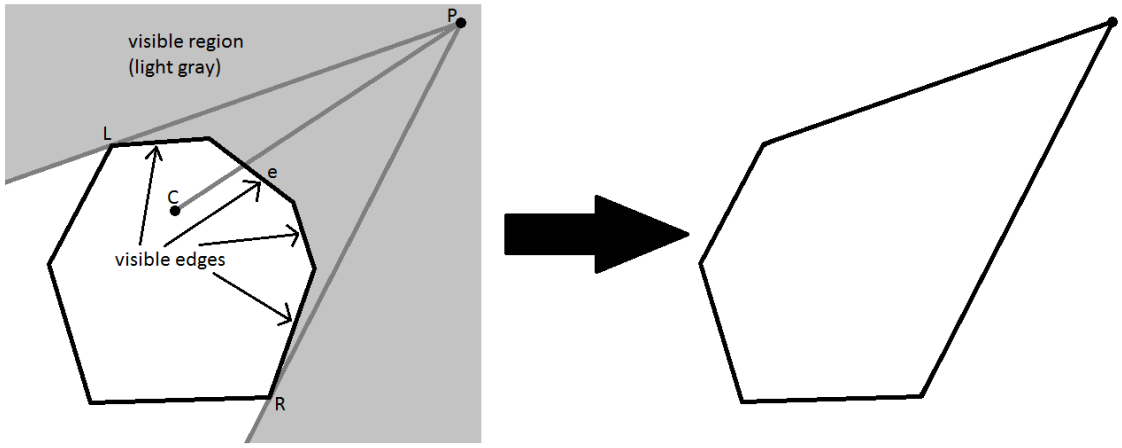

Figure 3: Subroutine BuildTent (CH, P, e), with e as the edge intersected by ray CP.

2.2 Random incremental convex hull

Here, we give a randomized convex hull algorithm and analyze its running time using backwards analysis. (For simplicity, assume that no three points in the input are collinear.)

The algorithm is incremental: start with the convex hull of points P_1, P_2, P_3, and iteratively insert the remaining points $P_4, P_5, \ldots, P_n$ in some order. For each iteration i, maintain the convex hull of the first i inserted points in, say, clockwise order in a doubly-linked list. So, on iteration i, we have the convex hull of the first $i-1$ points and need to figure out how to modify this hull to include the i'th point. If the new point is inside the current convex hull, then clearly, nothing needs to be done. For the case when the new point is outside, we need the notion of "visibility": if we view the convex hull CH as an (opaque) building, the *visible region* from a point P outside the building is what you can see if you stand at point P (see figure 3, light gray). Formally, it is the set of points Q in the plane such that the segment $\overline{PQ}$ does not intersect the hull. The following claim, while nontrivial to prove, is geometrically intuitive:

Claim 2.4 (Proof omitted)**.** *Suppose P is outside of CH, and that the edges of CH visible from P are $e_1, \ldots, e_k$ in clockwise order. Let L be the left endpoint of e_1 and R be the right endpoint of e_k (see figure 3). Then, removing edges $e_1, \ldots, e_k$ and inserting the segments $\overline{LP}$ and $\overline{PR}$ in their place produces the convex hull with point P included.*

Note that at most 2 edges are inserted per iteration, and every edge is deleted at most once, So, by amortized analysis, the number of added or removed edges, and thus the number of doubly-linked list operations, is $O(1)$ per iteration.

It remains to find, for each new point, whether it lies within the current convex hull, and if not, the set of visible edges. First, note that, in the latter case, it suffices to find *one* visible edge: once such an edge e is known for a new point P, we can search the doubly-linked list both clockwise and counterclockwise starting from e to determine the complete set of visible edges; this searching cost is also amortized to $O(1)$. We can thus declare a subroutine `BuildTent`(CH, P, e) that takes in a point P outside of CH and an edge e of CH visible from P, and outputs the new convex hull with P included. Note that the total running time of all calls to `BuildTent`(CH, P, e) is linear.

Subroutine `BuildTent`(CH, P, e)

1. Starting from e, go counterclockwise along CH until the next edge to be visited is no longer visible from P. Let L be the left endpoint of the last visible edge.
2. Do the same in the clockwise direction starting from e. Let R be the right endpoint of the last visible edge.
3. Remove all visited edges in both directions, and add the edges $\overline{LP}$ and $\overline{PR}$ in their place in the doubly-linked list.

The key idea in computing visible edges efficiently is to *precompute* them: for each point P not yet inserted, keep track of one edge of the current CH that is visible from P. One method is to fix a point C inside CH, and associate a future point P with the edge e of CH intersected by ray $\overrightarrow{CP}$ (see figure 3); e is necessarily visible from P. Note that a point C inside the initial convex hull of P_1, P_2, P_3 will work and never needs to be changed. Also, we can test whether or not P already lies in CH for free: simply compute the intersection point of $\overrightarrow{CP}$ and e and see if it lies beyond P in ray $\overrightarrow{CP}$.

The only trouble with this approach is that, on each iteration, an edge e with an associated future point P might be deleted, in which case we need to re-associate P with one of the two newly inserted edges. It turns out that, if we randomly permute the points $P_4, P_5, \ldots, P_n$ before inserting them, the additional cost in maintaining visible edges is small in expectation.

Thus, the complete algorithm is as follows:

Algorithm `RandomIncrementalCH`

1. Construct the convex hull CH of P_1, P_2, P_3 in clockwise order, stored in a doubly-linked list.
2. Compute a point C inside the convex hull (e.g., the centroid $(P_1 + P_2 + P_3)/3$).
3. Randomly permute the remaining points, and call the new order $P_4, P_5, \ldots, P_n$.
4. For each $P_4, \ldots, P_n$, compute the edge of CH intersected by ray $\overrightarrow{CP_i}$, and associate this edge with P_i.
5. For $i = 4, \ldots, n$:
   (a) Retrieve the associated edge e of P_i, which is visible from P_i.
   (b) Compute the intersection of $\overrightarrow{CP}$ and e; call this point Q. If length CQ is greater than length CP, then P is inside CH, so do nothing and `continue` onto the next iteration.
   (c) Run `BuildTent`(CH, P, e).
   (d) For each deleted edge, reassign the future points associated with that edge to whichever of $\overline{LP}$ and $\overline{PR}$ that intersects ray $\overrightarrow{CP_i}$.

Finally, we analyze the expected running time of assigning and reassigning future points to edges. Whenever we add a new edge, we pay a cost equal to the number of future points assigned to that edge. To apply backwards analysis, we consider the incremental algorithm backwards: on

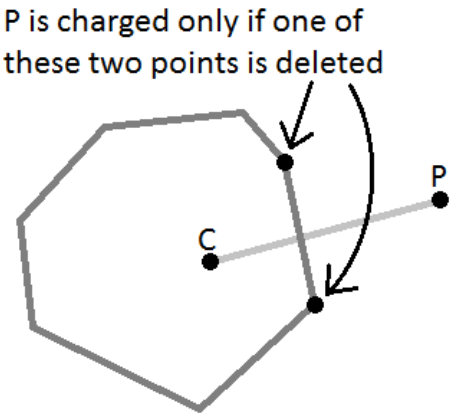

Figure 4: In the backwards analysis: cases when "future" point P is charged.

each iteration, remove a random point $P \notin \{P_1, P_2, P_3\}$, until only P_1, P_2, P_3 remain. If P is inside the current convex hull CH, pay nothing. Otherwise, remove the two edges of CH that contain P, and pay cost equal to the number of "future" points whose rays intersect these two edges (where "future" is technically the past now). If there are i points remaining besides P_1, P_2, P_3, then each "future" point has probability at most $2/i$ of contributing to the cost of this iteration (see figure 4). By linearity of expectation, the expected cost of this iteration is $(n-3-i) \cdot 2/i$. Summing over all i gives an expected total cost of

$$\sum_{i=1}^{n-3}(n-3-i)\frac{2}{i} \leq (n-3)\sum_{i=1}^{n-3}\frac{2}{i} = 2(n-3)H_{n-3} = O(n \log n).$$

Thus, the expected total running time of assigning and re-assigning future points is $O(n \log n)$. Since everything else is linear time, the entire `RandomIncrementalCH` also has expected running time $O(n \log n)$.

2.3 Lower bound for convex hull

Consider the following reduction from sorting to convex hull: given an array $x_1, \ldots, x_n$ of distinct numbers, perform the following:

Reduction `SortViaConvexHull`

1. Run a `ConvexHull` algorithm on the 2D points $(x_1, x_1^2), \ldots, (x_n, x_n^2)$.
2. Find the smallest element in the array; call it $x_{\min}$.
3. Starting from $x_{\min}$, output the elements of the convex hull in counterclockwise order.

Aside from algorithm `ConvexHull`, the reduction takes linear time. One can show that `SortViaConvexHull` correctly outputs the numbers in sorted order (see figure 5 for intuition). Thus, if $T_{\text{sort}}(n)$ and $T_{\text{CH}}(n)$ are the minimum times required for sorting and convex hull (possibly allowing randomized algorithms), then $T_{\text{sort}}(n) \leq T_{\text{CH}}(n) + O(n)$. Since it is known that $T_{\text{sort}}(n) = \Omega(n \log n)$, we get $T_{\text{CH}} \geq \Omega(n \log n) - O(n) = \Omega(n \log n)$.

Next, we present how to convert MergeSort and QuickSort into a convex hull algorithm. Input A is a set of points in $\mathbb{R}^2$, $A = \{P_1, \cdots, P_n\}$ with $P_i = (x_i, y_i)$.

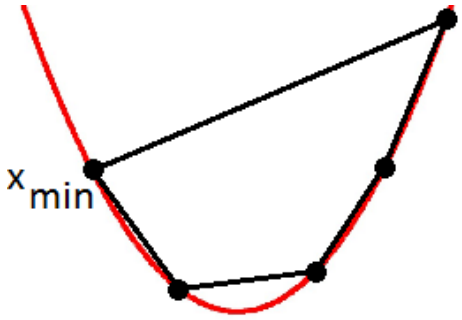

Figure 5: The convex hull of the points (x_i, x_i^2) must contain the numbers x_i in sorted order.

Preprocess `PrepAforSorting`

1. Sort A by x-coordinate.

Algorithm `MergeHull(A)`

1. If $|A| = 1$, return P_1;
2. Else, $CH_L = MergeHull(P_1, \ldots, P_{\frac{n}{2}}), CH_R = MergeHull(P_{\frac{n}{2}}, \ldots, P_n)$.
3. Stitch(L, R).

For convex hull CH_L, CH_R, define LowerBridge(L, R) as the lowest segment joining $(u, v) \in CH_L \times CH_R$ and upperbridge as the highest segment joining CH_L, CH_R. The subroutine Stitch(L, R) finds UpperBridge(L, R) B^u and LowerBridge(L, R) B^l, then find a convex hull containing both L, R with B^u, B^l as two sides.

Subroutine `Stitch(L,R)`

1. Let a denote the rightmost vertex in CH_L, b denote the leftmost vertex in CH_R.
2. Repeat (a), (b) until no more updates can be made:
   (a) If $\underline{a}$ is at the right side of segment $[a, b]$, set $a \leftarrow \underline{a}$.
   (b) If $\overline{b}$ is at the right side of segment $[a, b]$, set $b \leftarrow \overline{b}$.
3. Return $[a, b]$ as LowerBridge(L, R) B^l .
4. UpperBridge(L, R) B^u can be found in similar way, the only changes needed is to find $\underline{a}$ and $\overline{b}$ at the left side of $[a, b]$ in repeating updates.
5. Join CH_L and CH_R with B^l and B^u, the generated convex set contains CH_L and CH_R.

Stitch has time complexity $O(n)$ because we are only traversing through the points and the repeated updates will run at most $2n$ times. MergeHull has time complexity $O(n \log n)$, which can be computed from solving recursion $T(n) = 2T(\frac{n}{2}) + cn$.