CS 787: Advanced Algorithms

Randomness

Instructor: Dieter van Melkebeek

Randomized algorithms have an additional primitive operation that deterministic algorithms do not have. We can select a number from a range $[1 \ldots x]$ uniformly at random, at a cost assumed to be linearly dependent on the size of x in binary representation. The algorithm then makes a decision based on the outcome of this random selection.

We first look at some defining characteristics of randomized algorithms that differ from deterministic algorithms. We then look at several problems that can be solved efficiently with randomized algorithms (arithmetic formula identity testing, arithmetic circuit identity testing, primality testing, minimum cut, selection, sorting).

1 Characteristics of randomized algorithms

There are a few characteristics of randomized algorithms that we wish to emphasize.

1.1 Error

The output of a randomized algorithm on a given input is a random variable. Thus, there may be a positive probability that the outcome is incorrect. As long as the probability of error is small for every possible input to the algorithm, this is not a problem. By rerunning the algorithm a number of times we can reduce the probability of error.

Consider any decision problem Π and any input x for Π of size n. Let A be a randomized algorithm for Π and $Error(A)$ be the event that A returns an incorrect answer. In order for us to be able to amplify A's probability of success, it is sufficient that $\Pr(Error(A)) \leq \frac{1}{2} - \delta$ for some $\delta \geq \frac{1}{n^c}, c > 0$. Running the algorithm k times, where k is polynomial in n, we obtain an error probability which is exponentially small:

$$\begin{aligned}
\Pr(\text{majority of } k \text{ runs on input } x \text{ is wrong}) &= \sum_{i=\frac{k}{2}}^{k} \binom{k}{i} \left(\frac{1}{2}-\delta\right)^{i} \left(\frac{1}{2}+\delta\right)^{k-i} \\
&\leq \left(\frac{1}{2}-\delta\right)^{\frac{k}{2}} \left(\frac{1}{2}+\delta\right)^{\frac{k}{2}} \sum_{i=\frac{k}{2}}^{k} \binom{k}{i} \\
&\leq \left(\frac{1}{2}-\delta\right)^{\frac{k}{2}} \left(\frac{1}{2}+\delta\right)^{\frac{k}{2}} 2^k \\
&= \left(\frac{1}{4}-\delta^2\right)^{\frac{k}{2}} 2^k \\
&= \left(1-4\delta^2\right)^{\frac{k}{2}} \\
&< e^{-2k\delta^2}
\end{aligned}$$

So we can choose k such that the probability of error is exponentially small, for example by choosing $k = \frac{1}{\delta^2} = n^{2c}$. A similar bound can be obtained by applying the Chernoff bound.

1.2 Running time

Using random numbers in our algorithms means that we are essentially allowing the algorithm to make a choice between two or more options, but we have no way of knowing what decision the algorithm will make. Apart from introducing the possibility of error in our output, this may also influence the running time of our algorithm.

Randomized algorithms can be classified into the following classes:

- *Monte Carlo* algorithms have a small probability of producing erroneous output, but they have a running time polynomial in the input size.
- *Las Vegas* algorithms have an *expected* running time which is polynomial in the size of the input, but in some instances they could run forever. However, if they do terminate they never produce erroneous output.

Note that there are two other options which we have not considered:

- "Atlantic City" algorithms are randomized algorithms with expected running time polynomial but may run forever (like a Las Vegas algorithm), and may produce erroneous output with small probability (like Monte Carlo algorithms).
- Randomized algorithms which run in polynomial time and never err. This class is by definition the class of algorithms deciding languages in P. We still have randomness, but it can be substituted by essentially picking "heads" for every coin flip giving us a deterministic algorithm.

2 Arithmetic Formula Identity Testing

Given an arithmetic formula that involves only addition, subtraction and multiplication over the integers, we wish to learn whether the formula evaluates to zero. An example of such an arithmetic formula can be seen below:

$$(x_1 + x_2) \cdot (x_1 - x_2) - x_1 \cdot x_1 + x_2 \cdot x_2 \tag{1}$$

Put more succinctly, given an arithmetic formula $\mathcal{Y}(x_1, x_2, \ldots x_n)$ we wish to determine whether $\mathcal{Y} \equiv 0$.

We devise the following randomized algorithm: We pick values $\{s_1, s_2, \ldots, s_n\}$ for the variables at random uniformly from some range S. If $\mathcal{Y}(s_1, s_2, \ldots s_n) = 0$, return "true", otherwise return "false."

The result returned by the algorithm may be incorrect. If we picked a root of $\mathcal{Y}$, then our algorithm would have returned "true" even though the correct answer is "false." Nonetheless, the probability of this occurring is small. We make use of the following lemma:

Lemma 1. *Schwartz-Zippel Lemma If $\mathcal{Y}$ is a polynomial of degree at most d over n variables over a finite field S, then the probability over $x \in S^n$ that $\mathcal{Y}(x) = 0$, given that $\mathcal{Y}$ is not identically 0, is at most $d/|S|$.*

The proof of this lemma is by induction and is not stated here.

With this lemma, we realize that if $\mathcal{Y} \not\equiv 0$, then

$$\Pr(\mathcal{Y}(s_1, \ldots, s_n) = 0) \leq \frac{d}{|S|} = \epsilon \tag{2}$$

Even if ϵ is a poor bound, we can boost the confidence of the result of the algorithm by running it multiple times thereby increasing the probability that some run of the algorithm will return the correct result.

$$\Pr(\text{all k runs yield false positive}) \leq \epsilon^k \tag{3}$$

We thus run the algorithm k times and reject the circuit (i.e. determine it's not identical to 0) if *any* run returns "false".

The complexity of this algorithm depends on the size of the values in the computation. The input values $\{x_1, x_2, \ldots, x_n\}$ are bounded by $|S|$. However, multiplications increase the degree of $\mathcal{Y}$ and therefore increase the size of the partial results. Nonetheless, we realize that

$$Degree(\mathcal{Y}) \leq \# \text{ operators} \leq |\mathcal{Y}|,$$

where $|\mathcal{Y}|$ denotes the size of the polynomial measured in the number of bits needed to encode it. Thus, the complexity is polynomial to the size of the algorithm's input.

3 Arithmetic Circuit Identity Testing

Arithmetic circuit identity testing (a.k.a. polynomial identity testing) is similar to arithmetic formula identity testing, but we now consider polynomials represented by arithmetic circuits. Like arithmetic formula identity testing this can be performed simply and efficiently using a randomized algorithm. All known deterministic algorithms for this problem take exponential time. Furthermore, many problems reduce to this problem (e.g. primality testing).

We would like to apply the randomized algorithm from the arithmetic formula identity testing problem to this problem, but we notice an issue: the evaluation cannot be done efficiently because the degree of the polynomial may grow exponentially with the circuit size, and the intermediate values involved in the Schwartz-Zippel test may be doubly exponential. We would like a polynomial time solution.

First, let us denote $f = \mathcal{Y}(s_1, \ldots, s_n)$. We devise the following strategy for evaluating f: pick some value m and do all the computation modulo m. This bounds the size of all intermediate results. If $f = 0$, then it is also $0(\text{mod } m)$ and the algorithm will return the correct result. However, taking the modulo of each computation poses the new risk that the algorithm will return a false positive result (i.e. it will return "true" when the correct answer is "false"). This occurs when $f \equiv 0(\text{mod } m)$, but $f \neq 0$.

Our task is now to bound the probability that the algorithm returns a false positive result.

We pick m uniformly at random within some range $M = \{1, 2, \ldots, |M|\}$, and consider the conditional probability that $f \bmod m \neq 0$ given that $f \neq 0$. This is the probability that we correctly reject a circuit that is not equivalent to zero:

$$\begin{aligned}\Pr\left(f \not\equiv 0 \,(\text{mod } m) \mid f \neq 0\right) = & \Pr(m \text{ is prime}) \cdot \Pr(\text{Prime } m \text{ does not divide } f) + \\ & \Pr(m \text{ is } \textit{not} \text{ prime}) \cdot \Pr(\text{Composite } m \text{ does not divide } f)\end{aligned}$$

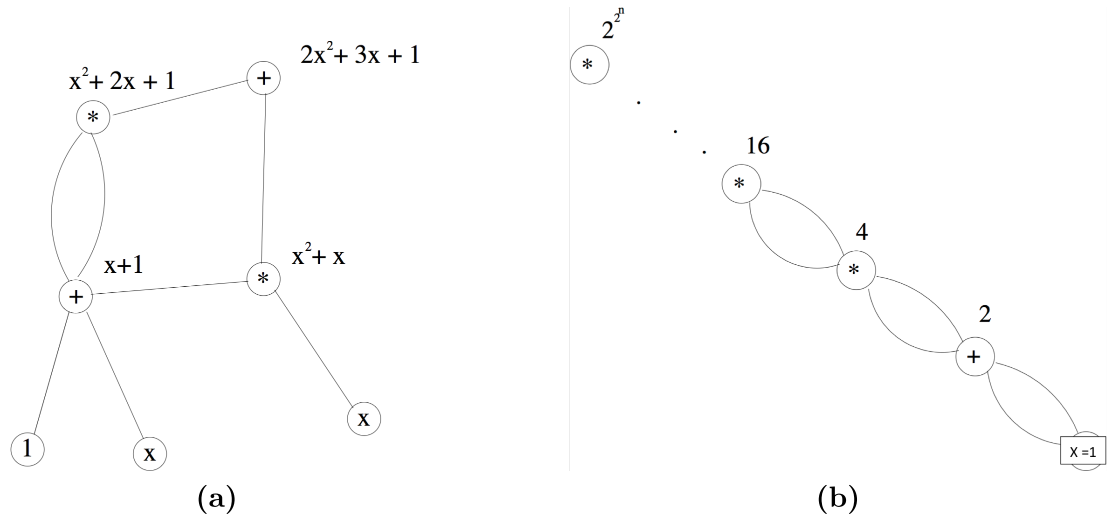

Figure 1: (a) An example of an arithmetic circuit problem. (b) Another example of a circuit for which the value is double exponential in its size

$$\geq \Pr\left(m \text{ is prime}\right) \cdot \left(1 - \frac{\#\text{ prime divisors of } f}{\#\text{ primes in } M}\right)$$

This states that the probability of correctly rejecting a non-zero circuit is greater than the probability of choosing a prime m that does not divide f. Again we would like to bound this probability.

Bounding # prime divisors of f. We realize that $|f| \leq |S|^{2^{|\mathcal{Y}|^c}}$ where c is some constant. Why? We see that the largest possible value for a substitution of a variable is $|S|$. Furthermore, we see that the largest result we can obtain for a fixed sized circuit occurs when we chain multiplication gates (as seen in Figure 1). This results in a value proportional to $|S|^{2^{|\mathcal{Y}|}}$ where $|\mathcal{Y}|$ is proportional to the size of the circuit. The constant c ensures the inequality holds.

We realize that the maximum number of prime divisors of $|f|$ must be less than $\log_2|f|$. This is because if all of the prime divisors of f were the smallest possible prime number, 2 , at most $2^{|f|}$ of them would need to be multiplied in order to equal f. This is stated more succinctly as follows:

$$|f| \geq \prod_{m \in P} m \geq 2^{|P|},$$

where P is the set of all prime divisors of f. This in turn implies the bound:

$$|P| \leq \log_2|f| \leq \log_2|S|^{2^{|\mathcal{Y}|^c}} = (\log_2|S|)\, 2^{|\mathcal{Y}|^c}$$

Bounding # primes in M.

Theorem 1 (Prime Number Theorem)**.** *Let $\pi(x)$ denote the number of primes less than some number x. Then*

$$\pi(x) \sim \frac{x}{\ln x}.$$

We set M as follows:

$$|M| = 2^{|\mathcal{Y}|^d}$$

and see that

$$\begin{aligned}\#\text{ primes in } M &= \frac{2^{|\mathcal{Y}|^d}}{\ln\left(2^{|\mathcal{Y}|^d}\right)} \\ &= \Theta\left(\frac{2^{|\mathcal{Y}|^d}}{|\mathcal{Y}|^d}\right) \\ &> 2^{|\mathcal{Y}|^c} \text{ (i.e. } |f|) \text{ if } d > c\end{aligned}$$

Bounding $\Pr(m$ **is prime).** We now use the prime number theorem to bound the probability of picking a prime from M:

$$\begin{aligned}\Pr(m \text{ is prime}) &= \frac{\#\text{ primes in } M}{\text{size of } M} \\ &= \frac{|M|}{\ln|M|}\cdot\frac{1}{|M|} \\ &= \Theta\left(\frac{1}{|\mathcal{Y}|^d}\right)\end{aligned}$$

Bounding the original $\Pr\left(f \not\equiv 0\,(\textbf{mod}\ m) \mid f \neq 0\right)$**.** With these bounds worked out, we calculate a total bound on the probability of interest – that is, we can bound the total probability that the algorithm returns a false positive:

$$\begin{aligned}\Pr\left(f \not\equiv 0\,(\text{mod}\ m) \mid f \neq 0\right) &\geq \left(\frac{1}{|\mathcal{Y}|^d}\right)\cdot\left(1 - \frac{(\log_2|S|)\,2^{|\mathcal{Y}|^c}}{\frac{2^{|\mathcal{Y}|^d}}{|\mathcal{Y}|^d}}\right) \\ &= \left(\frac{1}{|\mathcal{Y}|^d}\right)\cdot\left(1 - \frac{2^{|\mathcal{Y}|^c}}{\frac{2^{|\mathcal{Y}|^d}}{\log_2(|S|)\cdot|\mathcal{Y}|^d}}\right) \\ &= \left(\frac{1}{|\mathcal{Y}|^d}\right)\cdot\left(1 - \frac{2^{|\mathcal{Y}|^c - |\mathcal{Y}|^d}}{\log_2(|S|)\cdot|\mathcal{Y}|^d}\right) \\ &= \Omega\left(\frac{1}{|\mathcal{Y}|^d}\right) \text{ since } |\mathcal{Y}|^c \ll |\mathcal{Y}|^d\end{aligned}$$

Improving Confidence. As with all Monte Carlo algorithms with sufficiently small probability for error, we can boost the confidence of the result by running the algorithm multiple times. Here we show how many times to run the algorithm to ensure the an output with at least a target probability of error ϵ. We can improve the confidence of the algorithm by running the algorithm k times. If any runs of the algorithm return "false", we know the circuit is identical to zero. If all runs return "true", then the probability of error is

$$\epsilon = \left(1 - \Omega\left(\frac{1}{|\mathcal{Y}|^d}\right)\right)^k$$

We now work to show how to determine k to achieve a fixed ϵ. We first realize the following:

$$(1-a) \leq e^{-a} \implies (1-a)^k \leq e^{-ak}$$

If we set $a \doteq \frac{1}{|\mathcal{Y}|^d}$ and set $\epsilon \doteq e^{-ak}$ then we know the following:

$$\epsilon = e^{-ak} \geq (1-a)^k$$

and can now solve easily for k in terms of ϵ:

$$\begin{aligned} e^{-ak} &= \epsilon \\ -ka &= \ln \epsilon \\ k &= -a \ln \epsilon \\ k &= a \ln \frac{1}{\epsilon} \end{aligned}$$

4 Primality Testing

Given an integer n, determine if n is prime. An efficient solution will solve this problem in polynomial time in the size of the bit representation of n, i.e., in time polynomial in $\log n$. For a many years primality testing was the "prime" example of a problem for which a randomized polynomial-time algorithm was known, but no deterministic polynomial-time algorithm. About a decade ago a deterministic polynomial-time algorithm was discovered by reducing the problem to arithmetic circuit identity testing, and "derandomizing" the randomized algorithm for the latter for those special cases. Obtaining a deterministic polynomial time algorithm for the general case of arithmetic circuit (or formula) identity testing remains open.

5 Min Cut on an Undirected Graph

5.1 Problem description and randomized algorithm

Consider the following problem:

Given: Undirected Graph $G = (V, E)$.
Goal: Partition V into (S, T) such that the number of edges between S and T is minimized.

This problem is similar to the network flow mincut problem, but note some differences

- The graph is not directed.
- All the edges are of capacity 1.
- There is no source or sink.

If we enforced the constraint that a designated pair of vertices needed to be separated by the cut, then we could easily reduce this problem to the max-flow/min-cut problem. That is, we need only replace each edge by two directional edges in opposite directions and run a standard max-flow algorithm.

However, in this problem, there is no such constraint and thus, any separation of vertices can occur. If we truly want to reduce this problem to the max-flow/min-cut it is still possible (albeit inefficient). To do so, we would consider every pair of vertices as the source and sink, run the max-flow algorithm on each pair and return the minimum cut from the set of results.

Nonetheless, such a solution is inefficient and thus we illustrate the following randomized algorithm:

```
Algorithm 1: Randomized Algorithm for the Min-Cut Problem
Input: A graph $G = (V, E)$, $|V| = n$ and $|E| = m$
Output: A Cut $C$ containing the edges of the minimum cut
RAND_MIN_CUT($G$)
(1)     Repeat $n - 2$ times
(2)         Pick edge $e$ uniformly at random
(3)         Contract $e$ (Refer Figure 2.)
(4)     return the edges between the two remaining vertices.
```

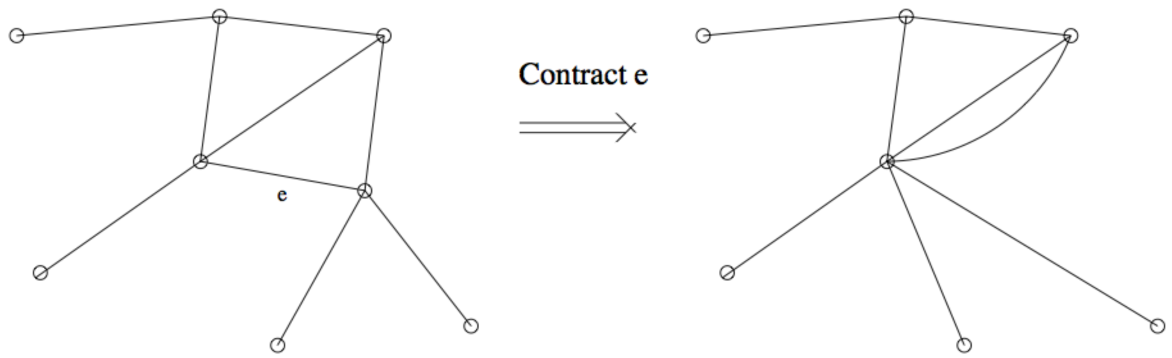

Figure 2: As seen in the figure above, the edge e is contracted. This operation merges the two vertices that e connects such that all edges adjacent to these two vertices are now adjacent to this new merged vertex.

Each contract operation reduces the number of vertices in the graph by 1. So at the end of $n-2$ steps, only 2 vertices remain. The edges between these two super-vertices constitute a cut. Note that multiple edges between a pair of vertices produced by the contract operation are not removed.

5.2 Bounding the error

Theorem 2. *For any minimum cut C:*

$$\Pr\left(\text{Algorithm outputs } C\right) \geq \frac{1}{\binom{n}{2}} \tag{4}$$

Proof. The algorithm outputs C if and only if it never chooses to contract an edge in C. Let A_k be the event that our algorithm does not choose an edge from C to contract in the k^{th} step. Using the identities described in the probability primer, we realize that:

$$\begin{aligned}\Pr(\text{Algorithm outputs } C) &= \Pr\left(\bigcap_{k=1}^{n-2} A_k\right) \\ &= \Pr(A_{n-2} \cap \cdots \cap A_2 \cap A_1) \\ &= \Pr(A_{n-2} \mid A_{n-3} \cap \cdots \cap A_2 \cap A_1) \ldots \Pr(A_2 \mid A_1) \cdot \Pr(A_1)\end{aligned} \tag{5}$$

Now we try to bound the values of the conditional probability terms in the last line of Equation 5. We note that the l^{th} conditional probability can be expressed as follows:

$$\Pr(A_l \mid A_{l-1} \cap \cdots \cap A_2 \cap A_1) = 1 - \frac{|C|}{\#\text{ Edges that remain after the } (l-1)^{th} \text{ step}} \tag{6}$$

Given that we have not chosen an edge from C during any previous $l-1$ steps (due to the definition of the event A_l), then there still remains $|C|$ edges in C that we have not contracted. Thus, the above probability is the probability that we do *not* choose an edge from C during the l^{th} step.

First, we note that for any undirected graph, if we sum over all of the degrees of the vertices, the result will be twice the number of edges in the graph. This is because we will be counting each edge twice (once at each vertex the edge connects).

Second, we know that for the original graph G the minimum degree of any vertex is at least $|C|$. How do we know this? If we assume otherwise, then we run into a contradiction. That is, if there existed a vertex, v, with degree less than $|C|$, then the cut $(\{v\}, V - \{v\})$ would be of size less than C. This contradicts the fact that C is defined to be the minimum cut.

With the aforementioned two realizations, we can bound our conditional probability as follows:

$$\begin{aligned}\#\text{ Edges that remain after the } (l-1)^{th} \text{ step} &= \frac{1}{2}\sum_{v \epsilon G'} Degree(v) \\ &\geq \frac{1}{2}|V'| \cdot |C| \\ &= \frac{1}{2}(n-l+1) \cdot |C|,\end{aligned} \tag{7}$$

where $G' = (V', E')$ is the graph at the end of the $(l-1)^{th}$ step. This bound leads to a bound on the probability of Equation 6:

$$\Pr(A_l \mid A_{l-1} \cap \cdots \cap A_2 \cap A_1) \geq 1 - \frac{2}{n-l+1} = \frac{n-l-1}{n-l+1} \tag{8}$$

Finally, we can bound Equation 5 by substituting the result from above:

$$\begin{aligned}\Pr(A_1) \cdot \Pr(A_2 \mid A_1) \cdot \ldots \cdot \Pr(A_{n-2} \mid A_{n-3} \cap \cdots \cap A_2 \cap A_1) &\geq \prod_{l=1}^{n-2} \frac{n-l-1}{n-l+1} \\ &= \frac{2}{n(n-1)} \\ &= \frac{1}{\binom{n}{2}}\end{aligned} \tag{9}$$

The product in Equation 9 is a telescoping product. All but the last two terms ($l = n-2, n-3$) are cancelled from the top and all but the first two terms ($l = 1, 2$) are cancelled from the bottom. Thus we proved the bound on the probability of the algorithm returning minimum cut C. $\square$

Note that there may be more than one min cut so we might have a better chance of finding some minimum cut. However, in the worst case, there may be only one minimum cut, and this bound is tight.

Corollary 1.

$$\Pr(\textit{Algorithm Correct}) \geq \frac{1}{\binom{n}{2}} \tag{10}$$

5.3 Improving Confidence

The bound we proved is not very good. We would like to be able to solve this problem with an arbitrarily low error. To accomplish this, we simply need to run the algorithm k times and return the minimum cut over all of the runs. The probability of error (that we return a non-minimal cut over all runs) decreases the more times we run the algorithm:

$$\Pr(\text{Error}) \leq \left(1 - \frac{1}{\binom{n}{2}}\right)^k \leq e^{-k/\binom{n}{2}} \tag{11}$$

If we choose $k = c\binom{n}{2}$ for some constant c, we can decrease the error exponentially by linearly increasing c. Therefore, we can achieve exponentially small error by repeating the algorithm $O(n^2)$ times. This is not faster than the deterministic algorithm, but it is considerably simpler to implement than a network-flow/min-cut algorithm.

6 Randomized Selection

Recall the specification of the selection problem:

Given: An array A of n numbers, and an integer k with $1 \leq k \leq n$.
Goal: Find the kth element of A when sorted in non-decreasing order.

The deterministic, divide-and-conquer, linear-time algorithm for this problem uses an approximate median of A as a pivot p to break up the array A into the subarray L_p of all entries with value less than p, the entries equal to p, and the subarray R_p of all entries larger than p. Given p, we can construct L_p and R_p in linear time, and based on their sizes we know whether the kth smallest element of A lies in L_p, equals p, or lies in R_p. In the middle case, we're done; in the other two cases we recurse on L_p (with the same value of k) or on R_p (with the value of k reduced by $n - |R_p|$), respectively

The procedure for finding an approximate median was somewhat complicated and involved another recursive call. Instead of doing that, we now simply pick the pivot p uniformly at random among the elements of A. Intuitively, such a random pivot has a good probability of being an approximate median in the sense of reducing the size of the remaining array by a constant factor less than one. As such a reduction in size can only happen $O(\log n)$ times, chances are the recursion bottoms out within $O(\log n)$ levels. Our intuition is that the amount of local work at each level would roughly behave like a geometric sequence with ratio less than one, resulting in a linear overall expected running time. We now formalize this intuition.

For the purposes of the analysis, we refer to the elements of the array A by their index in the sorted order, and we think of picking the pivot p as follows: take a real r in the interval $(0, n)$ uniformly at random, and round up to the next integer. Note that the distribution of p is indeed uniform over $\{1, \ldots, n\}$, over the entries of A.

For any integer $i \geq 0$, consider the following random variable:

$$X_i = \begin{cases} \text{size of the subarray at the } i\text{th level of recursion} & \text{if the } i\text{th level exists} \\ 0 & \text{otherwise.} \end{cases}$$

We consider the original call to the procedure to be the 0th level of recursion, so $X_0 = n$. Since the amount of work we spend at the ith level of recusion is at most $c \cdot X_i$ for some constant c, the total amount of work is at most $c \cdot \sum_i X_i$.

Claim 1. *For every integer $i \geq 0$,*

$$\mathrm{E}[X_{i+1}] \leq \frac{3}{4} \cdot \mathrm{E}[X_i]. \tag{12}$$

Proof. We first show the following for every fixed integer ℓ:

$$\mathrm{E}[X_{i+1} \mid X_i = \ell] \leq \frac{3}{4}\ell. \tag{13}$$

Since $X_{i+1} = 0$ whenever $X_i = 0$, we only need to consider cases with $\ell > 0$. In order to pick the pivot p at level i, we take a real r in $(0, \ell)$ uniformly at random, and set $p \doteq \lceil r \rceil$. Note that $L_p = \{1, \ldots, p-1\}$ and $R_p = \{p+1, \ldots, \ell\}$. It follows that $|L_p| \leq r$ and $|R_p| \leq \ell - r$. Since $X_{i+1} \leq \max(|L_p|, |R_p|)$, the LHS of (13) is at most the expected value of $\max(r, \ell - r)$ when r is picked uniformly at random from $(0, \ell)$. As can be gleaned from the plot below, the latter expected value equals $\frac{3}{4}\ell$.

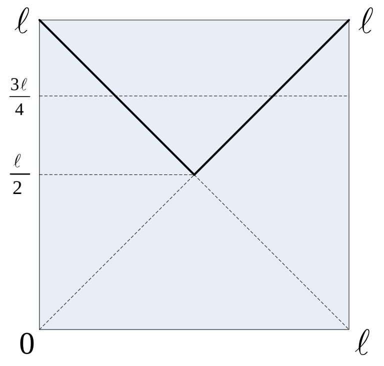

Figure 3: Plot of $\max(r, \ell - r)$ for $r \in (0, \ell)$.

This follows because on the first half of the interval, i.e., for $r \in (0, \frac{\ell}{2})$, $\max(r, \ell - r) \equiv \ell - r$, a linear function that evolves from ℓ to $\frac{\ell}{2}$ on the interval $(0, \frac{\ell}{2})$, and averages $\frac{1}{2} \cdot (\ell + \frac{\ell}{2}) = \frac{3}{4}\ell$ on that interval. Similarly, on the second part of the interval, i.e., for $r \in (\frac{\ell}{2}, \ell)$, $\max(r, \ell - r) \equiv r$, and the average of r for $r \in (\frac{\ell}{2}, \ell)$ also equals $\frac{3}{4}\ell$. Thus, the overall average equals $\frac{3}{4}\ell$ as well. This establishes (13).

We obtain equation (12) from equation (13) as follows:

$$\mathrm{E}[X_{i+1}] = \sum_{\ell} \Pr[X_i = \ell] \cdot \mathrm{E}[X_{i+1} \mid X_i = \ell] \leq \sum_{\ell} \Pr[X_i = \ell] \cdot \frac{3}{4} \cdot \ell = \frac{3}{4} \cdot \mathrm{E}[X_i].$$

□

Claim 1 has the following implications.

Lemma 2. *For every integer $i \geq 0$,*

$$\mathrm{E}[X_i] \leq \left(\frac{3}{4}\right)^i \cdot n. \tag{14}$$

Proof. Repeated application of (12) shows that $\mathrm{E}[X_i] \leq \left(\frac{3}{4}\right)^i \cdot \mathrm{E}[X_0]$. Since $X_0 = n$, the lemma follows. □

Theorem 3. *The expected running time of randomized selection is $O(n)$.*

Proof. The expected running time is upper bounded by $c \cdot \sum_i \mathrm{E}[X_i]$. By Lemma 2, the latter sum is no more than $\sum_i \left(\frac{3}{4}\right)^i \cdot n = 4n$ (due to the convergence of a geometric series with common ratio between 0 and 1). □

Lemma 3. *The probability that randomized selection has more than $s \doteq \lceil \log_{\frac{4}{3}}(n) \rceil + \Delta$ levels of recursion is at most $\left(\frac{3}{4}\right)^{\Delta}$.*

Proof. Randomized selection has more than s levels of recursion iff $X_s > 0$. As X_s is a non-negative random variable that only takes integer values, Markov's inequality and Lemma 2 show that

$$\Pr[X_s > 0] = \Pr[X_s \geq 1] \leq \mathrm{E}[X_s] \leq \left(\frac{3}{4}\right)^s \cdot n \leq \left(\frac{3}{4}\right)^{\Delta}.$$

□

Theorem 4. *The expected number of levels of recursion of randomized selection is $O(\log n)$.*

Proof. The expected number of levels of recursion can be written as

$$\begin{aligned} & \sum_{s=1}^{\infty} \Pr[\text{ the number of levels is at least } s \text{ }] \\ = & \sum_{s=0}^{\infty} \Pr[\text{ the number of levels is more than } s \text{ }] \\ = & \sum_{s=0}^{\infty} \Pr[X_s > 0] \\ \leq & \lceil \log_{\frac{4}{3}}(n) \rceil + \sum_{\Delta=0}^{\infty} \left(\frac{3}{4}\right)^{\Delta} = \lceil \log_{\frac{4}{3}}(n) \rceil + 4 = O(\log n), \end{aligned}$$

where the inequality follows from breaking up the sum into:

- the terms with $s < \lceil \log_{\frac{4}{3}}(n) \rceil$, each of which is a probability and can therefore be upper bounded by 1, and
- the terms with s of the form $s = \lceil \log_{\frac{4}{3}}(n) \rceil + \Delta$ for non-negative integer Δ, each of which can be upper bounded by $\left(\frac{3}{4}\right)^{\Delta}$ by Lemma 3.

$\square$

7 Randomized Quicksort

Recall the specification of the sorting problem:

Given: An array A of n numbers.
Goal: Sort(A), i.e., the array A sorted in non-decreasing order.

We consider randomized quicksort:

1. Pick a pivot p uniformly at random among the elements of the array A.
2. Break up the array A into the subarray L_p of all entries with value less than p, the entries equal to p, and the subarray R_p of all entries larger than p.
3. Recursively sort L_p and R_p.
4. Return the concatenation of the sorted version of L_p, the entries equal to p, and the sorted version of R_p.

This algorithm always outputs the correctly sorted array A, but the running time is a random variable depending on the choices of the pivots. As in randomized selection, the local amount of work associated with a given node in the recursion tree is linear in the size of the corresponding subarray. Since the subarrays at a given level of recursion are disjoint, this implies that the amount of work per level of recursion can be upper bounded by $c \cdot n$ for some constant c. Thus, all that remains is to analyze the number of levels of recursion, which is a random variable depending on the choice of pivots.

In order to do this analysis, we observe that the number of levels of recursion for randomized quicksort on input A equals the maximum over all $k \in \{1, \ldots, n\}$ of the number of levels of recursion of randomized selection on input A and k. This observation allows us to use our analysis of randomized selection and derive the following.

Lemma 4. *The probability that randomized quicksort has more than $s \doteq 2\lceil \log_{\frac{4}{3}}(n) \rceil + \Delta'$ levels of recursion is at most $\left(\frac{3}{4}\right)^{\Delta'}$.*

Proof. By the above observation, randomized quicksort has more than s levels of recursion on input A iff for some $k \in \{1, \ldots, n\}$ randomized selection on input A and k has more than s levels of recursion. By Lemma 2, for any fixed $k \in \{1, \ldots, n\}$, the probability that randomized selection on input A and k has more than s levels of recursion is no more than $\left(\frac{3}{4}\right)^{\Delta}$, where $\Delta = \lceil \log_{\frac{4}{3}}(n) \rceil + \Delta'$. By a union bound over all n possible values of k, the probability that randomized quicksort on input A has more than s levels of recursion is no more than $n \cdot \left(\frac{3}{4}\right)^{\Delta} \leq \left(\frac{3}{4}\right)^{\Delta'}$. $\square$

In the same way that we derived Theorem 4 from Lemma 3, we obtain the following result from Lemma 4.

Theorem 5. *The expected number of levels of recursion of randomized quicksort is $O(\log n)$.*

As the amount of work per level of recursion can be bounded by $c \cdot n$, we conclude:

Theorem 6. *The expected running time of randomized quicksort is $O(n \log n)$.*

Finally, we point out that the randomized process induced by running randomized quiksort on the fixed array $A = (1, 2, \ldots, n)$ is the same as running a deterministic version of quicksort on a random permutation of A. In a deterministic version of quicksort the pivot is chosen in a deterministic way; several variants exist: the first element of the array, the last one, the middle one, etc. Theorem 4 therefore also shows that the average complexity of deterministic quicksort on a random input array is $O(n \log n)$.