Recitation 3: Randomized Algorithms

1 Quicksort

Quicksort is a divide and conquer algorithm for sorting an array $A$.

1. Divide: Choose a pivot element $x$ from $A$. Partition $A$ into subarrays $L$, $E$, and $G$ as shown below.

<table>
<thead>
<tr>
<th></th>
<th>&lt; $x$</th>
<th>= $x$</th>
<th>&gt; $x$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$L$</td>
<td>$E$</td>
<td>$G$</td>
</tr>
</tbody>
</table>

2. Conquer: Recursively sort $L$ and $G$.


1.1 Basic Quicksort

Choose $A[1]$ as pivot. But then in the worst case, $L$ or $G$ has $n-1$ elements and the other is empty. The divide step takes time $\Theta(n)$, since we have to compare each element to $x$. Then we have to recurse on a problem of size $n-1$. So we have

$$T(n) = T(n-1) + \Theta(n) = \Theta(n^2)$$

1.2 “Paranoid” Quicksort

In Randomized Quicksort, we instead choose $x$ randomly from $A$. Then we can achieve an expected runtime of $\Theta(n \log n)$. To ease the analysis, we will analyze a slightly different version of Quicksort, which we will call Paranoid Quicksort.

In Paranoid Quicksort, we begin by choosing an element from $A$ at random, and then we partition the elements as we normally would. However, if we find that either $L$ or $G$ contains more than $3/4$ of the elements of $A$, then we randomly choose a new pivot and repeat. Once we find a pivot which satisfies that condition, we proceed with Quicksort as usual.

We begin by analyzing the time it will take us to perform the divide step. Each time we choose a pivot, making the appropriate comparisons will take time $\Theta(n)$. It remains to determine the number of new pivots we expect to need to choose.
If \( x \) is one of the smallest \( 1/4 \) of the elements, then \( G \) will have size greater than \( 3/4 \) of \( A \) and we will have to repeat. Similarly, we will have to repeat if \( x \) is in the largest \( 1/4 \) of \( A \). Thus, exactly half the elements of \( A \) are “good” pivots. So, when we choose a pivot, there is probability \( p = 1/2 \) that the pivot is good.

Let \( X \) be a random variable representing the number of pivots we choose before we find a good one. Then \( X \) is a geometric random variable with parameter \( p = 1/2 \), so we know

\[
E[X] = \frac{1}{1/2} = 2.
\]

Since the expected number of pivots we have to choose is a constant, the over expected runtime of the divide step is \( \Theta(n) \).

In the worst case, \( x \) will divide the elements into arrays of size \( n/4 \) and \( 3n/4 \). So we have the recursion

\[
T(n) = T(n/4) + T(3n/4) + \Theta(n).
\]

To solve this recurrence we use a recursion tree.

At each level of the tree, at most \( cn \) work is done. Furthermore, the maximum height of the tree is \( \log_{4/3} n \). Thus, the total amount of work is \( O(n \log n) \).

2 Polynomial Identity Testing

Given two polynomials \( p(x) \) and \( q(x) \), we want to determine whether \( p(x) \equiv q(x) \). This will be the case if \( p \) and \( q \) evaluate to the same value for all \( x \). Formally, we have

\[
p(x) \equiv q(x) \iff \forall x, p(x) = q(x).
\]

To test this, it is certainly not feasible to try all possible values of \( x \).
To simplify, we’ll consider an equivalent problem. Now, given some polynomial \( r(x) \), we want to determine whether \( r(x) \equiv 0 \). We can see that if we let \( r(x) = p(x) - q(x) \) then the problems are equivalent.

Recall that an \( n \)-degree polynomial can be uniquely determined by \( n + 1 \) points. So, if we can find \( n + 1 \) values of \( x \) such that \( r(x) = 0 \), then it must be the case that \( r(x) \equiv 0 \). Now to determine if \( r(x) \equiv 0 \) we can use the following algorithm:

1. Choose \( n + 1 \) distinct values \( x_0, \ldots, x_{n+1} \).
2. Evaluate \( r(x_i) \) for all \( i \).
3. If there is any value of \( i \) such that \( r(x_i) \neq 0 \), output \( r(x) \neq 0 \). Otherwise output \( r(x) \equiv 0 \).

This algorithm is deterministic: it will always output the correct answer no matter what values we choose for \( x_0, \ldots, x_n \). However, evaluating \( r \) at \( n + 1 \) distinct points will take \( O(n) \). Can we do better?

Next we consider a randomized algorithm. Let’s devise an algorithm which adheres to the following:

1. If \( r(x) \equiv 0 \), always output \( r(x) \equiv 0 \).
2. If \( r(x) \neq 0 \), output \( r(x) \neq 0 \) with \( p \geq 1/2 \).

Consider the following algorithm:

1. Choose \( x \) randomly from \([0 \ldots 2n]\).
2. Evaluate \( y = r(x) \).
3. If \( y = 0 \), output \( r(x) \equiv 0 \). Otherwise output \( r(x) \neq 0 \).

Clearly the runtime of this algorithm is \( O(1) \). But does it meet our conditions? Well, if it is the case that \( r(x) \equiv 0 \), then for every \( x \) we will have \( r(x) = 0 \). So condition (1) is met. Now consider the case where \( r(x) \neq 0 \). We know that a polynomial of degree \( n \) has \( n \) roots. So there are at most \( n \) values \( x \) in \([0 \ldots 2n]\) such that \( r(x) = 0 \). Thus we have

\[
P(r(x) = 0) = P(x \text{ is a root}) \leq \frac{n}{2n} = \frac{1}{2}.
\]

Thus condition (2) is met as well, and we have a constant time randomized algorithm for polynomial identity testing.
3 Man on the Moon problem or Testing Equality of Strings

Testing equality of strings has many applications in real life. For example, multiple copies of the same document may exist in different places and we would like to know if the two copies are equal. However, the communication cost may be expensive and we would like to test the equality without sending the whole document over the communication channel. This problem is commonly called “man on the moon problem”.

Say that Alice has an $n$-bit binary string $a$, and Bob has an $n$-bit binary string $b$. Alice is sitting on earth and Bob is sitting on the moon. They want to determine whether $a = b$, but since it’s clearly expensive to send messages to the moon and back they want to minimize the number of bits they have to transfer.

Trivially, Alice could send $a$ to Bob, and Bob could send Alice a bit back indicating if $a$ and $b$ are equal. But this would take $O(n)$ bits. Instead, lets represent $a$ and $b$ as polynomials. If $a = a_0a_1\ldots a_n$, then we define

$$a(x) = \sum_{i=0}^{n} a_i x^i,$$

and similarly define $b(x)$. Now clearly $a = b$ if and only if $a(x) \equiv b(x)$. But we already know how to solve this problem! Let Alice choose a random value $x$ from $[1 \ldots 2^n]$. Then Alice evaluates $a(x)$, and sends both $x$ and $a(x)$ to Bob. He can then evaluate $b(x)$, and then conclude that $a = b$ if $a(x) = b(x)$.

Since $x$ is chosen from $[1\ldots2^n]$, we know that sending $x$ takes $O(\log n)$ bits. However sending $a(x)$ could be expensive, as $a(x)$ could have even more than $n$ bits. To remedy this, we will work over a finite field instead of over $\mathbb{Z}$. Let’s operate over $\mathbb{Z}_p$, i.e. do all calculations mod $p$, where $p$ is some prime number. Now, $a(x)$ requires at most $\log p$ bits to represent. What are the constrains on the value of $p$?

1. Prime $p$ must be large enough so that $\mathbb{Z}_p$ has distinct values in $[1 \ldots 2n]$. Thus, $p > 2n$ should hold.

2. Prime $p$ should be small enough so that $\log p$ is $O(\log n)$.

We can see that by operating over $\mathbb{Z}_p$, we can reduce the number of bits sent to be $O(\log n)$. 