HKOI 2017/18 Training
Randomized Algorithms

Alex Tung
alex20030190@yahoo.com.hk

7 April 2018
Table of Contents

1 Introduction

2 Probability Crash Course

3 The Classics
   • Quicksort
   • std::nth_element
   • Hashing
   • Primality Test

4 How to know these algos are good?

5 RNG

6 Can you do it when it matters?
What is a “randomized algorithm”? 

- **Usual algorithm:**
  - Input
  - Process
  - Output

Besides the input data, a randomized algorithm also uses data from a source of randomness.
What is a “randomized algorithm”? 

- Usual algorithm:
  - Input → Process → Output

- Randomized algorithm:
  - Input → Process → Output
  - Besides the input data, a randomized algorithm also uses data from a source of randomness.
Why use a randomized algorithm?

Reason 1: To “get rid of” worst cases.
- An “adversary” (software tester, contest setter, CF hacker, etc.) may give you input in the worst possible order.
- A random shuffle (or two) will help you dodge these cases almost surely.
Why use a randomized algorithm?

Reason 1: To “get rid of” worst cases.
- An “adversary” (software tester, contest setter, CF hacker, etc.) may give you input in the worst possible order.
- A random shuffle (or two) will help you dodge these cases almost surely.

Reason 2: To “create” good conditions.
- You need to optimize an objective function over a HUGE state space.
- Some portion of the state space has a good property.
- Random sample, then do the rest. Repeat $N$ times.
Why use a randomized algorithm?

Reason 3a: To approximate.
- Again you need to work with a HUGE state space.
- **Random** sample for an approximate behaviour (e.g. average value).
- Monte-Carlo Algorithm
Why use a randomized algorithm?

Reason 3a: To approximate.
- Again you need to work with a HUGE state space.
- **Random** sample for an approximate behaviour (e.g. average value).
- Monte-Carlo Algorithm

Reason 3b: To approximate.
- Say the problem you are working with is NP-hard.
- One way is to use a *randomized* algorithm for a fast and reasonably good (likely not optimal) solution.
What we will do today

- Review basic probability
What we will do today

- Review basic probability
- Look at several classical randomized algorithms
What we will do today

- Review basic probability
- Look at several classical randomized algorithms
- Learn how to analyse the performance of a randomized algorithm
What we will do today

- Review basic probability
- Look at several classical randomized algorithms
- Learn how to analyse the performance of a randomized algorithm
- Learn how to use generate random numbers in C++
What we will do today

- Review basic probability
- Look at several classical randomized algorithms
- Learn how to analyse the performance of a randomized algorithm
- Learn how to use generate random numbers in C++
- Solve some contest problems
Digression: where are these digits from?
A (finite) probability space consists of a set $\Omega$ and a function $P : 2^\Omega \to [0, 1]$.

Say $\Omega$ consists of objects $a_1, a_2, \ldots, a_n$.

We usually assign nonnegative real numbers $p_1, p_2, \ldots, p_n$ to them, which sum to 1.

$P(S)$ will then equal $\sum \{p_i | a_i \in S\}$.

Usually (don’t take for granted!) we take $p_1 = p_2 = \cdots = p_n = \frac{1}{n}$.

Examples: fair dice, coins, permutations, ...
Given probability space $\Omega$.

One can construct a space of length-$N$ sequences $\Omega^N := \Omega_1 \times \Omega_2 \times \cdots \times \Omega_N$ (here $\Omega_1 = \Omega_2 = \cdots = \Omega_N = \Omega$).

An element in $\Omega^N$ looks like $(a_{i_1}, a_{i_2}, \ldots, a_{i_N})$, where $i_1, \ldots, i_N$ are indices in $[1, n]$. (There are $n^N$ elements.)

Such element has probability $p_{i_1} \times p_{i_2} \times \cdots \times p_{i_N}$.

Examples: sequence of coin tosses, sequence of dice rolls, ...
Example 1

- Suppose that $\Omega$ consists of $n$ elements $a_1, a_2, \ldots, a_n$, such that $p_i = \frac{1}{n}$ for all $i$.
- The first $k$ elements are “good”.

Questions

1. What is the probability that a chosen element is good?
2. If we sample $m$ elements with replacement, what is the probability that at least one chosen element is good?

Moral of the story
If the proportion of good elements is “not too small”, sampling “many” times should give a good element.

Alex Tung
Randomized Algorithms
7 Apr 18
Example 1

- Suppose that \( \Omega \) consists of \( n \) elements \( a_1, a_2, \ldots, a_n \), such that \( p_i = \frac{1}{n} \) for all \( i \).
- The first \( k \) elements are “good”.

Questions

1. What is the probability that a chosen element is good?
2. If we sample \( m \) elements with replacement, what is the probability that at least one chosen element is good?

- Try \( \frac{k}{n} = 0.01 \) and \( m = 1000 \).
Example 1

- Suppose that $\Omega$ consists of $n$ elements $a_1, a_2, \ldots, a_n$, such that $p_i = \frac{1}{n}$ for all $i$.
- The first $k$ elements are “good”.

Questions

1. What is the probability that a chosen element is good?
2. If we sample $m$ elements with replacement, what is the probability that at least one chosen element is good?

- Try $\frac{k}{n} = 0.01$ and $m = 1000$.

Moral of the story

If the proportion of good elements is “not too small”, sampling “many” times should give a good element.
Example 2

- Suppose that $\Omega$ consists of $n$ elements $a_1, a_2, \ldots, a_n$, such that $p_i = \frac{1}{n}$ for all $i$.
- Now you want to pick elements with replacement.

Questions

1. If you pick two elements, what is the probability that they are equal?
2. If you pick $m$ elements, what is the probability that some of them are equal?

Birthday paradox: if you randomly pick 23 people, there is a 50-50 chance that two of them have the same birthday.

Moral of the story
To avoid hash collision with probability $\sim 1$, one needs a “huge” modulo.
Example 2

Suppose that $\Omega$ consists of $n$ elements $a_1, a_2, \ldots, a_n$, such that $p_i = \frac{1}{n}$ for all $i$.

Now you want to pick elements with replacement.

Questions

1. If you pick two elements, what is the probability that they are equal?
2. If you pick $m$ elements, what is the probability that some of them are equal?

Birthday paradox: if you randomly pick 23 people, there is a 50-50 chance that two of them have the same birthday.
Example 2

- Suppose that \( \Omega \) consists of \( n \) elements \( a_1, a_2, \ldots, a_n \), such that \( p_i = \frac{1}{n} \) for all \( i \).
- Now you want to pick elements with replacement.

Questions

1. If you pick two elements, what is the probability that they are equal?
2. If you pick \( m \) elements, what is the probability that some of them are equal?

Birthday paradox: if you randomly pick 23 people, there is a 50-50 chance that two of them have the same birthday.

Moral of the story

To avoid hash collision with probability \( \sim 1 \), one needs a “huge” modulo.
Classical Randomized Algorithms

We will look into:

- Quicksort
- Finding $k$-th smallest element
- Hashing
- Primality test

You should check these out later:

- Bogosort (just for fun :P)
- Closest pair of points
- Minimum enclosing circle
- Treap (Data structures IV)
Classical Randomized Algorithms

We will look into:

- Quicksort
- Finding \( k \)-th smallest element
- Hashing
- Primality test

You should check these out later:

- Bogosort (just for fun :P)
- Closest pair of points
- Minimum enclosing circle
- Treap (Data structures IV)
Recall the deterministic quicksort:

**Algorithm (Quicksort, deterministic)**

```java
void quicksort(int[] A, int l, int r) {
    if l == r
        return
    choose pivot p = r
    find correct position x of A[p]
    quicksort(A, l, x-1)
    quicksort(A, x+1, r)
}
```

Of course one can choose $p = l$ or $p = (l + r) / 2$. The point is that pivot-choosing is deterministic. Average runtime = $O(n \log n)$; Worst case runtime = $O(n^2)$. Plus, it's easy to construct worst-case input.
Recall the deterministic quicksort:

**Algorithm (Quicksort, deterministic)**

```c
void quicksort(int[] A, int l, int r) {
    if l == r
        return
    choose pivot p = r
    find correct position x of A[p]
    quicksort(A, l, x-1)
    quicksort(A, x+1, r)
}
```

- Of course one can choose $p = l$ or $p = (l + r)/2$.
- The point is that pivot-choosing is **deterministic**.
- Average runtime $= O(n \log n)$; Worst case runtime $= O(n^2)$.
- Plus, it’s *easy* to construct worst-case input :(
Here is one way to implement randomized quicksort:

**Algorithm (Quicksort, randomized)**

```java
void quicksort(int[] A, int l, int r) {
    if l == r
        return
    choose pivot p uniformly randomly in [l, r]
    find correct position x of A[p]
    quicksort(A, l, x-1)
    quicksort(A, x+1, r)
}
```

Expected runtime = $O(n \log n)$; Worst case runtime = $O(n^2)$. What has changed? There is no "worst-case input"!
Here is one way to implement randomized quicksort:

**Algorithm (Quicksort, randomized)**

```c
void quicksort(int[] A, int l, int r) {
    if l == r
        return
    choose pivot p uniformly randomly in [l, r]
    find correct position x of A[p]
    quicksort(A, l, x-1)
    quicksort(A, x+1, r)
}
```

- **Expected** runtime $= O(n \log n)$; Worst case runtime $= O(n^2)$.
- What has changed?
- There is no “worst-case input”!
Finding $k$-th smallest element

- How to do this?
  - In worst case $O(n \log n)$?
Finding $k$-th smallest element

- How to do this?
  - In worst case $O(n \log n)$? (Sorting)
Finding $k$-th smallest element

- How to do this?
  - In worst case $O(n \log n)$? (Sorting)
  - In worst case $O(n)$?

Examples:

- **std::nth_element(A + 1, A + k, A + n + 1);** (See documentation)
- Randomized quickselect

Alex Tung
Randomized Algorithms
7 Apr 18
Finding $k$-th smallest element

- How to do this?
  - In worst case $O(n \log n)$? (Sorting)
  - In worst case $O(n)$? (“Median of medians”)

```cpp
std::nth_element(A + 1, A + k, A + n + 1);
(See documentation)
```
Finding $k$-th smallest element

- How to do this?
  - In worst case $O(n \log n)$? (Sorting)
  - In worst case $O(n)$? (“Median of medians”)
- Any easy algorithm with expected runtime $O(n)$?

\[
\text{std::nth_element}(A + 1, A + k, A + n + 1);
\]
(See documentation)

Randomized quickselect :)

Alex Tung
Randomized Algorithms
7 Apr 18
Finding $k$-th smallest element

- How to do this?
  - In worst case $O(n \log n)$? (Sorting)
  - In worst case $O(n)$? (“Median of medians”)
- Any easy algorithm with expected runtime $O(n)$?
- `std::nth_element(A + 1, A + k, A + n + 1);`
  (See documentation)
- *Randomized quickselect :)*
Quickselect is heavily based on quicksort.

**Algorithm (Quickselect)**

```c
void quickselect(int[] A, int l, int r, int k){
  // Make sure A[l + k - 1] is the k-th smallest element
  if l == r then return;
  choose pivot p = r
  find correct position x of A[p]
  if l + k - 1 < x
    quickselect(A, l, x - 1, k);
  else if l + k - 1 > x
    quickselect(A, x + 1, r, k - (x - l + 1));
}
```

Alex Tung
Randomized Algorithms
7 Apr 18
16 / 57
void quickselect(int[] A, int l, int r, int k) {
    // Make sure A[l + k - 1] is the k-th smallest element
    if l == r then return;
    choose pivot p uniformly randomly in [l, r]
    find correct position x of A[p]
    if l + k - 1 < x
        quickselect(A, l, x - 1, k);
    else if l + k - 1 > x
        quickselect(A, x + 1, r, k - (x - l + 1));
}
Randomized quickselect

Algorithm (Quickselect, randomized)

```c
void quickselect(int[] A, int l, int r, int k) {  
    // Make sure A[l + k - 1] is the k-th smallest element
    if l == r then return;
    choose pivot p uniformly randomly in [l, r]
    find correct position x of A[p]
    if l + k - 1 < x
        quickselect(A, l, x - 1, k);
    else if l + k - 1 > x
        quickselect(A, x + 1, r, k - (x - l + 1));
}
```

- **Expected** runtime = $O(n)$; Worst case runtime = $O(n^2)$.
- Library implementations mix quickselect with other algorithms.
Quickselect

Visualization:

(Source: Wikipedia)
Quickselect

Visualization:

(Source: Wikipedia)
Quickselect

Visualization:

(Source: Wikipedia)
Quickselect

Visualization:

(Source: Wikipedia)
Quickselect

Visualization:

(Source: Wikipedia)
Quickselect

Visualization:

(Source: Wikipedia)
Quickselect

Visualization:

(Source: Wikipedia)
Quickselect

Visualization:

(Source: Wikipedia)
Quickselect

Visualization:

(Source: Wikipedia)
Quickselect

Visualization:

(Source: Wikipedia)
Quickselect

Visualization:

(Source: Wikipedia)
Quickselect

Visualization:

(Source: Wikipedia)
Quickselect

Visualization:

(Source: Wikipedia)
Quickselect

Visualization:

(Source: Wikipedia)
Quickselect

Visualization:

(Source: Wikipedia)
Quickselect

Visualization:

(Source: Wikipedia)
Quickselect

Visualization:

(Source: Wikipedia)
Quickselect

Visualization:

(Source: Wikipedia)
Quickselect

Visualization:

(Source: Wikipedia)
Quickselect

Visualization:

(Source: Wikipedia)
Quickselect Visualization:

(Source: Wikipedia)
Quickselect

Visualization:

(Source: Wikipedia)
Quickselect

Visualization:

(Source: Wikipedia)
Quickselect

Visualization:

(Source: Wikipedia)
Quickselect Visualization:

(Source: Wikipedia)
Quickselect

Visualization:

(Source: Wikipedia)
Of course, hashing itself is deterministic: the same object must be mapped to the same hash value!

Consider polynomial hash: given \( B \) (base) and \( M \) (modulo), an integer sequence \( x_0, x_1, \ldots, x_n \) will be mapped to 
\[
\left( x_0 + x_1 \times B + x_2 \times B^2 + \cdots + x_n \times B^n \right) \mod M.
\]

For example, \( B = 26, M = 100 \):
\[
\text{abaca} \rightarrow 1, 2, 1, 3, 1 \rightarrow \\
(1 + 2 \times 26 + 26^2 + 3 \times 26^3 + 26^4) \mod 100 \rightarrow 33.
\]
Of course, hashing itself is deterministic: the same object must be mapped to the same hash value!

Consider polynomial hash: given $B$ (base) and $M$ (modulo), an integer sequence $x_0, x_1, \ldots, x_n$ will be mapped to $(x_0 + x_1 \times B + x_2 \times B^2 + \cdots + x_n \times B^n) \% M$.

For example, $B = 26, M = 100$:

$abaca \rightarrow 1, 2, 1, 3, 1 \rightarrow$

$(1 + 2 \times 26 + 26^2 + 3 \times 26^3 + 26^4) \% 100 \rightarrow 33$.

Is it ok to change to $0, 1, 0, 2, 0$ instead of $1, 2, 1, 3, 1$?
Quiz

Suppose you are to hash strings which consist of uppercase letters only. Which of the following pairs of $B$ and $M$ are “good”? Why/why not?

1. $B = 10$, $M = 1,000,000,007$
2. $B = 32$, $M = 1,048,576$
3. $B = 499501$, $M = 1,000,003$

All are bad! Why?
What values of $B$ and $M$ are good?

Quiz

Suppose you are to hash strings which consist of uppercase letters only. Which of the following pairs of $B$ and $M$ are “good”? Why/why not?

1. $B = 10, M = 1,000,000,007$
2. $B = 32, M = 1,048,576$
3. $B = 499501, M = 1,000,003$

All are bad! Why?
Necessary conditions for good \((B, M)\)

Suppose \(x_i\) will always be in range \([1, X]\). We need:

- \(B \geq X\)
- \(\gcd(B, M) = 1\)
- \(\text{ord}_M B\) is not too small (ideally equal to \(\varphi(M)\)). Here \(\text{ord}_M B\) is the smallest positive integer \(d\) such that \(B^d \equiv 1 \pmod{M}\).
Necessary conditions for good \((B, M)\)

Suppose \(x_i\) will always be in range \([1, X]\). We need:

- \(B \geq X\)
- \(\gcd(B, M) = 1\)
- \(\text{ord}_MB\) is not too small (ideally equal to \(\varphi(M)\)). Here \(\text{ord}_MB\) is the smallest positive integer \(d\) such that \(B^d \equiv 1 \pmod{M}\).

Usual practice:

- \(B = X\) or \(B = X + 1\)
- \(M\) a large prime (\(10^9 + 7\) and \(10^9 + 9\) are good choices!)
- If necessary, choose 2-4 good \((B, M)\) pairs
OK, but where is randomization?

- Codeforces hackers are OP! See Anti-hash test.
- Randomly generate good \((B, M)\) pairs to avoid being hacked :)
- For a more involved discussion on hashing, see rng_58’s blogpost.
- Basically: if you compare two objects with polynomial hashes of degree \(\leq d\), \(M\) is prime, and \(B\) is randomly chosen, then probability of collision \(\leq \frac{d}{M}\).
Primality Test

- Scenario: Given $N \leq 10^{18}$, you are to determine if $N$ is a prime.
- Your favourite $O(\sqrt{N})$ algorithm is unfortunately too slow.
- Simple solution: Miller-Rabin, Solovay-Strassen, ...
- Primes is in P (Agrawal, Kayal, Saxena 2002) - AKS Primality Test
As we shall see, many probabilistic primality tests are based on what **must** be true for primes.

<table>
<thead>
<tr>
<th>Return value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>FALSE</td>
<td>Composite</td>
</tr>
<tr>
<td>TRUE</td>
<td>Prime or Composite</td>
</tr>
</tbody>
</table>

Repeat many times with different parameters.

Passing more tests → more likely a prime.
Prime Fact

Given odd prime $n$, write $n - 1 = 2^s \times d$, where $d$ is odd.

Given $a$ not a multiple of $n$, either

$$a^d \equiv 1 \pmod{n}$$

or

$$a^{2^r \times d} \equiv -1 \pmod{n}$$

for some $r \in [0, s - 1]$. 
Take contrapositive, and we get a primality test.

Prime Test

Given $n$ odd, write $n - 1 = 2^s \times d$, where $d$ is odd. If there exists an $a$, not a multiple of $n$, such that

$$a^d \not\equiv 1 \pmod{n}$$

and

$$a^{2^r \times d} \not\equiv -1 \pmod{n}$$

for all $r \in [0, s - 1]$, then $n$ is composite.
Take contrapositive, and we get a primality test.

**Prime Test**

Given $n$ odd, write $n - 1 = 2^s \times d$, where $d$ is odd.

If there exists an $a$, not a multiple of $n$, such that

$$a^d \not\equiv 1 \pmod{n}$$

and

$$a^{2^r \times d} \not\equiv -1 \pmod{n} \text{ for all } r \in \{0, s - 1\},$$

then $n$ is composite.

- Given odd composite $n$, it has many “witnesses” $a$.
- Randomly pick some $a \in [1, n - 1]$ and run the test.
- [Black box] Failure probability $= 4^{-\#\text{tests}}$. 
In fact, there is a deterministic version.
It works if the upper bound of $n$ is given but not too large ($10^{18}$ is ok).
Only test with a “strong probable prime base”.

- if $n < 2,047$, it is enough to test $a = 2$;
- if $n < 1,373,653$, it is enough to test $a = 2$ and $3$;
- if $n < 9,080,191$, it is enough to test $a = 31$ and $73$;
- if $n < 25,326,001$, it is enough to test $a = 2$, $3$, and $5$;
- if $n < 3,215,031,751$, it is enough to test $a = 2$, $3$, $5$, and $7$;
- if $n < 4,759,123,141$, it is enough to test $a = 2$, $7$, and $61$;
- if $n < 1,122,004,669,633$, it is enough to test $a = 2$, $13$, $23$, and $1662803$;
- if $n < 2,152,302,898,747$, it is enough to test $a = 2$, $3$, $5$, $7$, and $11$;
- if $n < 3,474,749,660,383$, it is enough to test $a = 2$, $3$, $5$, $7$, $11$, and $13$;
- if $n < 341,550,071,728,321$, it is enough to test $a = 2$, $3$, $5$, $7$, $11$, $13$, and $17$.

Using the work of Feitsma and Galway enumerating all base 2 pseudoprimes in 2010, this was extended (see A014233), with the first result later shown using different methods in Jiang and Deng: \[12\]

- if $n < 3,825,123,056,546,413,051$, it is enough to test $a = 2$, $3$, $5$, $7$, $11$, $13$, $17$, $19$, and $23$.
- if $n < 18,446,744,073,709,551,616 = 2^{64}$, it is enough to test $a = 2$, $3$, $5$, $7$, $11$, $13$, $17$, $19$, $23$, $29$, $31$, and $37$.

(Source: Wikipedia)

It is conjectured that testing all $a < 2(\ln n)^2$ is sufficient.
Again, it starts with a prime fact.

Prime Fact

Given odd prime $n$. Let $\left( \frac{a}{n} \right)$ be the *Jacobi symbol*. Then for any integer $a$,

$$a^{\frac{(n-1)}{2}} \equiv \left( \frac{a}{n} \right) \pmod{n}$$

- Jacobi symbol is related to Legendre’s symbol.
- Check out *Wiki* or *M1706 Editorial*.
- At this moment, suffices to know it can be computed quickly.
Solovay-Strassen

Prime Test

Given odd $n$. Let $\left(\frac{a}{n}\right)$ be the Jacobi symbol.
If there exists an integer $a$ coprime with $n$, such that

$$a^{\frac{n-1}{2}} \not\equiv \left(\frac{a}{n}\right) \pmod{n},$$

then $n$ is not a prime.
Given odd $n$. Let $\left( \frac{a}{n} \right)$ be the Jacobi symbol.
If there exists an integer $a$ coprime with $n$, such that

$$a^{\frac{n-1}{2}} \not\equiv \left( \frac{a}{n} \right) \pmod{n},$$

then $n$ is not a prime.

- [Black box] Among those $a$ coprime with $n$, at least half are witnesses.
- Randomly pick some $a \in \mathbb{Z} \cap [1, n - 1]$ and run the test.
- Failure probability $= 2^{-\#\text{tests}}$. 
break;
Analysis of algorithms

Now, we analyse the performance of several randomized algorithms.
Say we have a probability space with $\Omega = \{a_1, a_2, \ldots, a_n\}$ and associated probabilities $p_1, p_2, \ldots, p_n$.

Given a random variable $X : \Omega \rightarrow \mathbb{R}$.

Its **expected value** is given by $\mathbb{E}X := \sum_{i=1}^{n} f(a_i)p_i$.

Its **variance** is given by $\text{Var} X := \mathbb{E}((X - \mathbb{E}X)^2)$.

Its **standard deviation** is given by $\sqrt{\text{Var} X}$.

Many people use $\mu$ for expected value, $\sigma^2$ for variance, and $\sigma$ for standard deviation.

Examples: fair dice, coins
Theorem 1 (Linearity of Expectation)

Given random variables $X_1, X_2, \ldots, X_k$ on the same probability space, we have

$$\mathbb{E}(X_1 + X_2 + \cdots + X_k) = \mathbb{E}X_1 + \mathbb{E}X_2 + \cdots + \mathbb{E}X_k.$$
A powerful theorem

**Theorem 1 (Linearity of Expectation)**

Given random variables $X_1, X_2, \ldots, X_k$ on the same probability space, we have

$$\mathbb{E}(X_1 + X_2 + \cdots + X_k) = \mathbb{E}X_1 + \mathbb{E}X_2 + \cdots + \mathbb{E}X_k.$$  

**Consequence**

One can break a complicated r.v. into smaller, simpler parts.
Theorem 2 (Chebyshev’s Theorem)

Given a random variable $X$ with expected value $\mu$ and variance $\sigma^2$. Then, for $\varepsilon > 0$ we have

$$P(|X - \mu| \geq \varepsilon) \leq \frac{\sigma^2}{\varepsilon^2}$$
Theorem 2 (Chebyshev’s Theorem)

Given a random variable $X$ with expected value $\mu$ and variance $\sigma^2$. Then, for $\varepsilon > 0$ we have

$$P(|X - \mu| \geq \varepsilon) \leq \frac{\sigma^2}{\varepsilon^2}$$

Consequence

If $X$ has “small” variance, then its values are concentrated near $\mathbb{E}X$. 

Here’s another one...
Randomized quicksort is $O(n \log n)$ in expectation

Reference: Chapter 5 of *Introduction to Algorithms* (Cormen, Leiserson, Reviest, Stein)

- Key observation: runtime is proportional to number of pairs of elements compared.
- Let $X := \sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij}$.
- $X_{ij}$ are random variables; equal to 1 if $A[i]$ is compared to $A[j]$ and 0 otherwise.
- **Here $A[1..n]$ is the sorted version of the array.**
- Then, expected runtime can be measured by $\mathbb{E}X$. 
Randomized quicksort is $O(n \log n)$ in expectation

Reference: Chapter 5 of *Introduction to Algorithms* (Cormen, Leiserson, Reviest, Stein)

- Key observation: runtime is proportional to number of pairs of elements compared.
- Let $X := \sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij}$.
- $X_{ij}$ are random variables; equal to 1 if $A[i]$ is compared to $A[j]$ and 0 otherwise.
- **Here $A[1..n]$ is the sorted version of the array.**
- Then, expected runtime can be measured by $\mathbb{E}X$.

- By **linearity of expectation**, $\mathbb{E}X := \sum_{i=1}^{n} \sum_{j=i+1}^{n} \mathbb{E}X_{ij}$.
- What is $\mathbb{E}X_{ij}$?
Randomized quicksort is $O(n \log n)$ in expectation

- What is $\mathbb{E}X_{ij}$?
- Consider the subarray $A[i..j]$.
- $A[i]$ and $A[j]$ will be compared if and only if $i$ or $j$ is chosen as pivot before any of $i + 1, \ldots, j - 1$.
- As pivot selection is uniform, $X_{ij}$ equals 1 with probability $\frac{2}{j-i+1}$.
- In other words, $\mathbb{E}X_{ij} = \frac{2}{j-i+1}$.
Randomized quicksort is $O(n \log n)$ in expectation

- $\mathbb{E}X_{ij} = \frac{2}{j-i+1}$
- Therefore, we have

\[
\mathbb{E}X = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{d=2}^{n} \sum_{i=1}^{n-d+1} \frac{2}{d} = 2[(n + 1)(\frac{1}{2} + \cdots + \frac{1}{n}) - (n - 1)] = O(n \log n).
\]
How meaningful is expected runtime?

- Not really, if performance can deviate wildly from the norm...
- If we know runtime **variance** is small, then great!
- For randomized quicksort, variance is $O(n^2)$. (See, for example, this paper).
- Chebyshev gives a trivial bound:

\[
P(|X - \mathbb{E}X| \leq cn \log n) = O\left(\frac{n^2}{(cn \log n)^2}\right) = O\left(\frac{1}{(c \log n)^2}\right)
\]
How meaningful is expected runtime?

- Not really, if performance can deviate wildly from the norm...
- If we know runtime variance is small, then great!
- For randomized quicksort, variance is $O(n^2)$. (See, for example, this paper).
- Chebyshev gives a trivial bound:

$$P(|X - \mathbb{E}X| \leq cn \log n) = O\left(\frac{n^2}{(cn \log n)^2}\right) = O\left(\frac{1}{(c \log n)^2}\right)$$

- In practice, quicksort is really quick.
Randomized quickselect is $O(n)$ in expectation

- Similarly, let $X := \sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij}$.
- $X_{ij}$ are random variables; equal to 1 if $A[i]$ is compared to $A[j]$ and 0 otherwise.
- **Here $A[1..n]$ is the sorted version of the array.**
- What is $\mathbb{E}X_{ij}$ this time?
Randomized quickselect is $O(n)$ in expectation

- What is $E X_{ij}$?
- Consider subarray $A[i..j]$.
- $A[i]$ and $A[j]$ will be compared if and only if $i$ or $j$ is the first chosen pivot among $min(k, i) ... max(k, j)$.
- Therefore, $E X_{ij} = \frac{2}{max(k, j) - min(k, i) + 1}$.
- We can show that $\sum_{i=1}^{n} \sum_{j=i+1}^{n} E X_{ij} = O(n)$, but the calculation is somewhat more involved.
Theorem 3 (Weak law of large numbers)

Suppose $X$ is a random variable with mean $\mu$ and variance $\sigma^2$. Let $Y_n := X_1 + X_2 + \cdots + X_n$, where $X_i = X$ for all $i$. Then, for any given $\varepsilon > 0$,

$$P(|\frac{Y_n}{n} - \mu| \geq \varepsilon) \to 0 \text{ as } n \to \infty.$$ 

The result follows from this bound:

$$P(|\frac{Y_n}{n} - \mu| \geq \varepsilon) \leq \frac{\sigma^2}{\varepsilon^2 n}.$$ 

This theorem is the basis for random sampling algorithms.
How to get random numbers in C++?

- Basic: \texttt{rand()} and \texttt{srand(seed)} in <cstdlib>.
- The <\texttt{random}> library has powerful stuff, but we don't talk about it.
- True randomness is impossible with "normal" computers.
- Pseudo-randomness is the best one can hope for.
int rand (void);

**Generate random number**
Returns a pseudo-random integral number in the range between 0 and `RAND_MAX`.

This number is generated by an algorithm that returns a sequence of apparently non-related numbers each time it is called. This algorithm uses a seed to generate the series, which should be initialized to some distinctive value using function `srand`.

`RAND_MAX` is a constant defined in `<cstdlib>`.

(Source: cplusplus.com)

- `RAND_MAX` usually equals 32767; value depends on compiler
- To get random integer in range $[0, R)$, use `rand()%R`.  

int rand (void);

**Generate random number**
Returns a pseudo-random integral number in the range between 0 and RAND_MAX.

This number is generated by an algorithm that returns a sequence of apparently non-related numbers each time it is called. This algorithm uses a seed to generate the series, which should be initialized to some distinctive value using function srand.

RAND_MAX is a constant defined in `<cstdlib>`.

- **RAND_MAX** usually equals 32767; value depends on compiler
- To get random integer in range \([0, R]\), use \(\text{rand}() \% R\).
- Question 1: assuming \(\text{rand}()\) is uniformly random. Is \(\text{rand}() \% R\) uniformly random?
int rand (void);

**Generate random number**

Returns a pseudo-random integral number in the range between 0 and `RAND_MAX`.

This number is generated by an algorithm that returns a sequence of apparently non-related numbers each time it is called. This algorithm uses a seed to generate the series, which should be initialized to some distinctive value using function `srand`.

`RAND_MAX` is a constant defined in `<cstdlib>`.

- `RAND_MAX` usually equals 32767; value depends on compiler
- To get random integer in range \([0, R]\), use `rand()\%R`.
- Question 1: assuming `rand()` is uniformly random. Is `rand()\%R` uniformly random?
- Question 2: what if \(R > RAND_MAX + 1\)?
\texttt{srand}(\textit{seed})

- \texttt{rand}() depends on a \textbf{random seed}.
- Same random seed + same input + same program → same output!
- You may start your program with \texttt{srand}(\textit{seed}).
- Varying the seed will change output.
$srand(seed)$

- $rand()$ depends on a random seed.
- Same random seed + same input + same program $\rightarrow$ same output!
- You may start your program with $srand(seed)$.
- Varying the seed will change output.

```c
int main(){
    srand(689);
    for(int i = 1; i <= 10; i++)
        printf("%d\n", rand() % 15);
    return 0;
}
```

```c
int main(){
    srand(777);
    for(int i = 1; i <= 10; i++)
        printf("%d\n", rand() % 15);
    return 0;
}
```
But it is still “deterministic”!

- Solution: Use \textit{time} as seed!
- The \textit{time()} function in \texttt{<ctime>} library can help.

\begin{verbatim}
#include <ctime>

int main() {
    srand(time(0));
    // Your code here...
    return 0;
}
\end{verbatim}

\textbf{Get current time}

Get the current calendar time as a value of type \texttt{time_t}.

The function returns this value, and if the argument is not a \textit{null pointer}, it also sets this value to the object pointed by \textit{timer}.

The value returned generally represents the number of seconds since 00:00 hours, Jan 1, 1970 UTC (i.e., the current time).

\textit{srand(time(0))} is commonly used.

(Source: cplusplus.com)
Usually \texttt{srand(time(0))} is good enough, but...
Usually *srand(time(0))* is good enough, but...

<table>
<thead>
<tr>
<th>B - Interactive LowerBound</th>
<th>View test</th>
<th>Unsuccessful hacking attempt</th>
<th>B - Interactive LowerBound</th>
<th>View test</th>
<th>Unsuccessful hacking attempt</th>
</tr>
</thead>
<tbody>
<tr>
<td>B - Interactive LowerBound</td>
<td>View test</td>
<td>Successful hacking attempt</td>
<td>B - Interactive LowerBound</td>
<td>View test</td>
<td>Successful hacking attempt</td>
</tr>
<tr>
<td>B - Interactive LowerBound</td>
<td>View test</td>
<td>Successful hacking attempt</td>
<td>B - Interactive LowerBound</td>
<td>View test</td>
<td>Successful hacking attempt</td>
</tr>
<tr>
<td>B - Interactive LowerBound</td>
<td>View test</td>
<td>Successful hacking attempt</td>
<td>B - Interactive LowerBound</td>
<td>View test</td>
<td>Successful hacking attempt</td>
</tr>
<tr>
<td>B - Interactive LowerBound</td>
<td>View test</td>
<td>Successful hacking attempt</td>
<td>B - Interactive LowerBound</td>
<td>View test</td>
<td>Successful hacking attempt</td>
</tr>
<tr>
<td>B - Interactive LowerBound</td>
<td>View test</td>
<td>Successful hacking attempt</td>
<td>B - Interactive LowerBound</td>
<td>View test</td>
<td>Successful hacking attempt</td>
</tr>
<tr>
<td>B - Interactive LowerBound</td>
<td>View test</td>
<td>Successful hacking attempt</td>
<td>B - Interactive LowerBound</td>
<td>View test</td>
<td>Successful hacking attempt</td>
</tr>
<tr>
<td>B - Interactive LowerBound</td>
<td>View test</td>
<td>Unsuccessful hacking attempt</td>
<td>B - Interactive LowerBound</td>
<td>View test</td>
<td>Unsuccessful hacking attempt</td>
</tr>
<tr>
<td>B - Interactive LowerBound</td>
<td>View test</td>
<td>Invalid input</td>
<td>B - Interactive LowerBound</td>
<td>View test</td>
<td>Successful hacking attempt</td>
</tr>
<tr>
<td>B - Interactive LowerBound</td>
<td>View test</td>
<td>Successful hacking attempt</td>
<td>B - Interactive LowerBound</td>
<td>View test</td>
<td>Successful hacking attempt</td>
</tr>
<tr>
<td>B - Interactive LowerBound</td>
<td>View test</td>
<td>Unsuccessful hacking attempt</td>
<td>B - Interactive LowerBound</td>
<td>View test</td>
<td>Invalid input</td>
</tr>
<tr>
<td>B - Interactive LowerBound</td>
<td>View test</td>
<td>Successful hacking attempt</td>
<td>B - Interactive LowerBound</td>
<td>View test</td>
<td>Successful hacking attempt</td>
</tr>
<tr>
<td>B - Interactive LowerBound</td>
<td>View test</td>
<td>Invalid input</td>
<td>B - Interactive LowerBound</td>
<td>View test</td>
<td>Successful hacking attempt</td>
</tr>
<tr>
<td>B - Interactive LowerBound</td>
<td>View test</td>
<td>Successful hacking attempt</td>
<td>B - Interactive LowerBound</td>
<td>View test</td>
<td>Successful hacking attempt</td>
</tr>
<tr>
<td>B - Interactive LowerBound</td>
<td>View test</td>
<td>Successful hacking attempt</td>
<td>B - Interactive LowerBound</td>
<td>View test</td>
<td>Successful hacking attempt</td>
</tr>
<tr>
<td>B - Interactive LowerBound</td>
<td>View test</td>
<td>Invalid input</td>
<td>B - Interactive LowerBound</td>
<td>View test</td>
<td>Successful hacking attempt</td>
</tr>
</tbody>
</table>
Usually \texttt{srand(time(0))} is good enough, but...

If still unsatisfied, consult \texttt{<chrono>} library.

I started using microseconds (or even nanoseconds, if available) for initializing the random seed, which is almost impossible to predict.

```cpp
srand(std::chrono::high_resolution_clock::now().time_since_epoch().count());
```

(Source: Codeforces)
Another useful function: $random\_shuffle(A, A + n)$

- In `<algorithm>` library.
- Also depends on random seed.
Another useful function: \textit{random\_shuffle}(A, A+n)

- In \texttt{<algorithm>} library.
- Also depends on random seed.

Think about...

How would you implement \textit{random\_shuffle}(A, A+n) using \texttt{rand()} only?
Now, try to solve contest problems using what you’ve learned.
Problem 1

Number of distinct $k$-substrings

Given a string $str[1..n]$ and a parameter $k$. Counter the number of distinct $k$-substrings of $str$.

Find a randomized $O(n \log n)$ solution.

Bonus: find a $o(n^2)$ (i.e. better than $\Theta(n^2)$) deterministic solution.

For example, $str = \text{"ababac"}$ and $k = 3 \rightarrow \text{answer} = 3.$
Rooted tree isomorphism (See also POJ 1343)

Given two rooted trees $T_1$ and $T_2$.
Determine if they are isomorphic, in $O(n \log n)$ time.

Bonus: look up for fast deterministic algorithms.
Problem 3

CF 843B - Interactive LowerBound

Interactive problem.
There is a hidden sorted linked list of length $n \leq 50000$.
It is built on an array.
Given $x$ and $start$, find the smallest integer in the list $\geq x$.
You can query 1999 times, specifying an index $i$ and get $value_i$ and $next_i$.

The list may look like this (pic from problem statement):

```
97  58  16  81  79
```
Problem 4

NEERC 15 Problem J - Jump

Interactive problem.
There is a hidden binary string $S$ of even length $n \leq 1000$.
You want to guess the string in $(n + 500)$ queries.
Each time, you can guess a length-$n$ binary string $T$.

Let $x$ be the number of matched positions between $S$ and $T$.
If $x = n$ or $x = \frac{n}{2}$ (!!), you will get $x$ as feedback.
Otherwise, you will get 0 as feedback.
Problems

- HKOJ M0932 - String Rotation
- HKOJ M16B4 - AVX Primality Test (well…)
- CF 364D - Ghd
- CF 763D - Timofey and a flat tree
- CF 840D - Destiny
The end