Review session

Lecture 99
March 4, 2021

Some of the slides are courtesy Prof. Chekuri
What we saw so far...

Fast Fourier Transform (FFT).

Dynamic Programming

- String algorithms.
- Graph algorithms: shortest path, independent set, dominating set, etc.

Randomized Algorithms

- Quick sort,
- High probability analysis: Markov, Chebyshev, and Chernoff inequalities
What we saw so far...

Fast Fourier Transform (FFT).

Dynamic Programming
- String algorithms.
- Graph algorithms: shortest path, independent set, dominating set, etc.

Randomized Algorithms
- Quick sort,
- High probability analysis: Markov, Chebyshev, and Chernoff inequalities
- Hashing, Fingerprinting
Part I

FFT
What is Fast Fourier Transform

**Definition**

Given a polynomial \( a = (a_0, a_1, \ldots, a_{n-1}) \) in coefficient representation the *Discrete Fourier Transform* (DFT) of \( a \) is the vector \( a' = (a'_0, a'_1, \ldots, a'_{n-1}) \) where \( a'_j = a(\omega^n_j) \) for \( 0 \leq j < n \).

\( a' \) is a sample representation of polynomial with coefficient representation \( a \) at \( n' \)th roots of unity.

We have shown that \( a' \) can be computed from \( a \) in \( O(n \log n) \) time. This divide and conquer *algorithm* is called the *Fast Fourier Transform* (FFT).
Convolution

Convolution of vectors \( a = (a_0, a_1, \ldots a_{n-1}) \) and \( b = (b_0, b_1, \ldots b_{n-1}) \) is a vector \( c = (c_0, c_1, \ldots, c_{2n-2}) \), where

\[
c_k = \sum_{i,j: \ i+j=k} a_i \cdot b_j
\]
Why FFT? Convolution and Polynomial Multiplication

**Convolution**

Convolution of vectors \( a = (a_0, a_1, \ldots a_{n-1}) \) and \( b = (b_0, b_1, \ldots b_{n-1}) \) is a vector \( c = (c_0, c_1, \ldots, c_{2n-2}) \), where

\[
c_k = \sum_{i,j: i+j=k} a_i \cdot b_j
\]

**Polynomial Multiplication**

If vectors \( a \) and \( b \) are coefficients of two \( n - 1 \) degree polynomials, (abusing notation) \( a(x) = \sum_{i=0}^{n-1} a_i x^i \), \( b(x) = \sum_{i=0}^{n-1} b_i x^i \) then \( c \) is the coefficient vector of the product polynomial \( a(x) \ast b(x) \).
Why FFT? Convolution and Polynomial Multiplication

Convolution

Given vectors \( a = (a_0, a_1, \ldots, a_{n-1}) \) and \( b = (b_0, b_1, \ldots, b_{n-1}) \) find its convolution vector \( c = (c_0, c_1, \ldots, c_{2n-2}) \).

1. Evaluate polynomials \( a \) and \( b \) at the \( 2n \)th roots of unity, to get their sample representation \( a' \) and \( b' \).

2. Compute sample representation \( c' = (a'_0 b'_0, \ldots, a'_{2n-2} b'_{2n-2}) \) of product \( c = a \cdot b \)

3. Compute \( c \) from \( c' \) using inverse Fourier transform.

- Step 1 takes \( O(n \log n) \) using two FFTs
- Step 2 takes \( O(n) \) time
- Step 3 takes \( O(n \log n) \) using one FFT
Let $\bar{a} = a_0, a_1, \ldots, a_{n-1}$ be a sequence of $n$ numbers representing value of a function at different points, we would like to “smooth” it using vector $\bar{b} = (b_0, b_1, \ldots, b_{k-1})$ for $k \leq n$ as follows:

$\bar{a}' = a'_0, a'_1, \ldots, a'_{n-1}$ where $a'_i = a_i b_0 + (a_{i+1} b_1 + \ldots + a_{i+k-1} b_{k-1}) + (a_{i-1} b_1 + a_{i-2} b_2 + \ldots + a_{i-k+1} b_{k-1})$. If an index goes out of bounds we assume that the corresponding value is 0.

Given $\bar{a}$ and $\bar{b}$ describe how $\bar{a}'$ can be computed in $O(n^2)$ time.
Application of FFT

Let $\bar{a} = a_0, a_1, \ldots, a_{n-1}$ be a sequence of $n$ numbers representing value of a function at different points, we would like to “smooth” it using vector $\bar{b} = (b_0, b_1, \ldots, b_{k-1})$ for $k \leq n$ as follows:

$\bar{a}' = a'_0, a'_1, \ldots, a'_{n-1}$ where $a'_i = a_ib_0 + (a_{i+1}b_1 + \ldots + a_{i+k-1}b_{k-1}) + (a_{i-1}b_1 + a_{i-2}b_2 + \ldots + a_{i-k+1}b_{k-1})$. If an index goes out of bounds we assume that the corresponding value is 0.

Given $\bar{a}$ and $\bar{b}$ describe how $\bar{a}'$ can be computed in $O(n \log n)$ time.
Part II

Dynamic Programming
Recursion

Reduction:
Reduce one problem to another

Recursion
A special case of reduction
1. reduce problem to a smaller instance of itself
2. self-reduction

1. Problem instance of size $n$ is reduced to one or more instances of size $n - 1$ or less.
2. For termination, problem instances of small size are solved by some other method as base cases.
What is Dynamic Programming?

Every recursion can be memoized. Automatic memoization does not help us understand whether the resulting algorithm is efficient or not.
What is Dynamic Programming?

Every recursion can be memoized. Automatic memoization does not help us understand whether the resulting algorithm is efficient or not.

Dynamic Programming:
A recursion that when memoized leads to an *efficient* algorithm.
Edit Distance

Definition

Edit distance between two words \( X \) and \( Y \) is the number of letter insertions, letter deletions and letter substitutions required to obtain \( Y \) from \( X \).

Example

The edit distance between FOOD and MONEY is at most 4:

\[
\text{FOOD} \rightarrow \text{MOOD} \rightarrow \text{MONOD} \rightarrow \text{MONED} \rightarrow \text{MONEY}
\]
Alignment

Place words one on top of the other, with gaps in the first word indicating insertions, and gaps in the second word indicating deletions.

FOOD
MONEY
Alignment

Place words one on top of the other, with gaps in the first word indicating insertions, and gaps in the second word indicating deletions.

\[
\begin{array}{cccccc}
F & O & O & D \\
M & O & N & E & Y
\end{array}
\]

Formally, an **alignment** is a set \( M \) of pairs \((i, j)\) such that each index appears exactly once, and there is no “crossing”: if \((i, j), \ldots, (i', j')\) then \(i < i'\) and \(j < j'\). One of \(i\) or \(j\) could be empty, in which case no comparison. In the above example, this is \(M = \{(1, 1), (2, 2), (3, 3), (4, 4), (4, 5)\}\).
Alignment

Place words one on top of the other, with gaps in the first word indicating insertions, and gaps in the second word indicating deletions.

<table>
<thead>
<tr>
<th>F</th>
<th>O</th>
<th>O</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>O</td>
<td>N</td>
<td>E</td>
</tr>
</tbody>
</table>

Formally, an alignment is a set $M$ of pairs $(i, j)$ such that each index appears exactly once, and there is no “crossing”: if $(i, j), \ldots, (i', j')$ then $i < i'$ and $j < j'$. One of $i$ or $j$ could be empty, in which case no comparison. In the above example, this is $M = \{(1, 1), (2, 2), (3, 3), (4, 4), (4, 5)\}$.

Cost of an alignment: the number of mismatched columns.
Edit Distance Problem

Problem
Given two words, find the edit distance between them, i.e., an alignment of smallest cost.
Basic observation

Let \( A = \alpha x \) and \( B = \beta y \)

\( \alpha, \beta \): strings. \( x \) and \( y \) single characters.

Possible alignments between \( A \) and \( B \):

\[
\begin{array}{|c|c|}
\hline
\alpha & x \\
\hline
\beta & y \\
\hline
\end{array}
\quad \text{or} \quad
\begin{array}{|c|c|}
\hline
\alpha & x \\
\hline
\beta y & \\
\hline
\end{array}
\quad \text{or} \quad
\begin{array}{|c|c|}
\hline
\alpha x & \\
\hline
\beta & y \\
\hline
\end{array}
\]

Observation

*Prefixes must have optimal alignment!*
Edit Distance

Basic observation

Let \( A = \alpha x \) and \( B = \beta y \)

\( \alpha, \beta \): strings. \( x \) and \( y \) single characters.

Possible alignments between \( A \) and \( B \)

\[
\begin{array}{c|c}
\alpha & x \\
\hline
\beta & y \\
\end{array}
\quad \text{or} \quad
\begin{array}{c|c}
\alpha & x \\
\hline
\beta y & \\
\end{array}
\quad \text{or} \quad
\begin{array}{c|c}
\alpha x & \\
\hline
\beta & y \\
\end{array}
\]

Observation

Prefixes must have optimal alignment!

\[
EDIST(A, B) = \min \left\{ EDIST(\alpha, \beta) + [x \neq y], 1 + EDIST(\alpha, B), 1 + EDIST(A, \beta) \right\}
\]
Recursive Algorithm

Assume strings are given as arrays $A[1..m]$ and $B[1..n]$

$$EDIST(A[1..i], B[1..j])$$
- If $(i = 0)$ return $j$
- If $(j = 0)$ return $i$
- $m_1 = 1 + EDIST(A[1..(i - 1)], B[1..j])$
- $m_2 = 1 + EDIST(A[1..i], B[1..(j - 1)])$
- If $(A[m] = B[n])$ then
  - $m_3 = EDIST(A[1..(i - 1)], B[1..(j - 1)])$
- Else
  - $m_3 = 1 + EDIST(A[1..(i - 1)], B[1..(j - 1)])$
- return $\min(m_1, m_2, m_3)$

Call $EDIST(A[1..m], B[1..n])$
Memoizing the Recursive Algorithm

\[
\text{int } M[0..m][0..n]
\]

Initialize all entries of \( M[i][j] \) to \( \infty \)

return \( EDIST(A[1..m], B[1..n]) \)

\[
EDIST(A[1..i], B[1..j])
\]

If \( (M[i][j] < \infty) \) return \( M[i][j] \) (* return stored value *)

If \( (i = 0) \)
\[
M[i][j] = j
\]

ElseIf \( (j = 0) \)
\[
M[i][j] = i
\]

Else
\[
\begin{align*}
    m_1 &= 1 + EDIST(A[1..(i - 1)], B[1..j]) \\
    m_2 &= 1 + EDIST(A[1..i], B[1..(j - 1)])
\end{align*}
\]

If \( (A[i] = B[j]) \)
\[
m_3 = EDIST(A[1..(i - 1)], B[1..(j - 1)])
\]

Else
\[
m_3 = 1 + EDIST(A[1..(i - 1)], B[1..(j - 1)])
\]

\[
M[i][j] = \min(m_1, m_2, m_3)
\]

return \( M[i][j] \)
Matrix and DAG of Computation
Removing Recursion to obtain Iterative Algorithm

\[
EDIST(A[1..m], B[1..n])
\]

\[
\begin{align*}
\text{int} & \quad M[0..m][0..n] \\
\text{for } i = 0 & \text{ to } m \text{ do } M[i, 0] = i \\
\text{for } j = 0 & \text{ to } n \text{ do } M[0, j] = j \\
\text{for } i = 1 & \text{ to } m \text{ do } \\
& \text{for } j = 1 \text{ to } n \text{ do } \\
& \quad M[i][j] = \min \left\{ \begin{array}{l}
\left[ x_i \neq y_j \right] + M[i - 1][j - 1], \\
1 + M[i - 1][j], \\
1 + M[i][j - 1]
\end{array} \right\}
\end{align*}
\]
Removing Recursion to obtain Iterative Algorithm

\[ \text{EDIST}(A[1..m], B[1..n]) \]

\[
\begin{align*}
\text{int } & \ M[0..m][0..n] \\
\text{for } i = 0 \text{ to } m \text{ do } & \ M[i, 0] = i \\
\text{for } j = 0 \text{ to } n \text{ do } & \ M[0, j] = j \\
\text{for } i = 1 \text{ to } m \text{ do } & \\
& \text{for } j = 1 \text{ to } n \text{ do } \\
& \quad M[i][j] = \min \left\{ \begin{array}{ll}
[x_i \neq y_j] + M[i - 1][j - 1], \\
1 + M[i - 1][j], \\
1 + M[i][j - 1]
\end{array} \right. 
\end{align*}
\]

Analysis

Running time is \( O(mn) \).
**Removing Recursion to obtain Iterative Algorithm**

\[ EDIST(A[1..m], B[1..n]) \]

\[
\text{int } M[0..m][0..n] \\
\text{for } i = 0 \text{ to } m \text{ do } M[i, 0] = i \\
\text{for } j = 0 \text{ to } n \text{ do } M[0, j] = j \\
\text{for } i = 1 \text{ to } m \text{ do } \\
\quad \text{for } j = 1 \text{ to } n \text{ do } \\
\quad \quad M[i][j] = \min \left\{ \begin{array}{ll}
[x_i \neq y_j] + M[i - 1][j - 1], \\
1 + M[i - 1][j], \\
1 + M[i][j - 1]
\end{array} \right. 
\]

**Analysis**

1. Running time is \( O(mn) \).
2. Space used is \( O(mn) \).
Matrix and DAG of Computation

Figure: Iterative algorithm in previous slide computes values in row order.
Problem

Given a graph $G = (V, E)$ a matching is a set of edges $M \subseteq E$ such that no two edges in $M$ share an end point. Describe an efficient algorithm that given a tree $T = (V, E)$ and non-negative weights $w : E \rightarrow R^+$ finds a maximum weight matching in $T$. 
Dijkstra’s Algorithm

Initialize for each node \( v \), \( \text{dist}(s, v) = \infty \)
Initialize \( S = \emptyset \), \( \text{dist}(s, s) = 0 \)

for \( i = 1 \) to \( |V| \) do

Let \( v \) be such that \( \text{dist}(s, v) = \min_{u \in V - S} \text{dist}(s, u) \)

\( S = S \cup \{v\} \)

for each \( u \) in \( \text{Adj}(v) \setminus S \) do

\( \text{dist}(s, u) = \min \left( \text{dist}(s, u), \text{dist}(s, v) + \ell(v, u) \right) \)

1. Using Fibonacci heaps. Running time: \( O(m + n \log n) \).
2. Can compute shortest path tree.
Single-Source Shortest Path Problems

**Input:** A directed graph \( G = (V, E) \) with arbitrary (including negative) edge lengths. For edge \( e = (u, v) \), \( \ell(e) = \ell(u, v) \) is its length.

- Given nodes \( s, t \) find shortest path from \( s \) to \( t \).
- Given node \( s \) find shortest path from \( s \) to all other nodes.
Negative Length Cycles

Definition

A cycle $C$ is a negative length cycle if the sum of the edge lengths of $C$ is negative.

Dijkstra’s algorithm does not work with negative edges.
1. Compute the shortest path distance from \( s \) to \( t \) recursively?
2. What are the smaller sub-problems?

Lemma

Let \( G \) be a directed graph with arbitrary edge lengths. If \( s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k \) is a shortest path from \( s \) to \( v_k \) then for \( 1 \leq i < k \):

\[ s = v_0 \rightarrow v_1 \rightarrow v_2 \ldots \rightarrow v_i \]

is a shortest path from \( s \) to \( v_i \).

Sub-problem idea: paths of fewer hops/edges.
1. Compute the shortest path distance from $s$ to $t$ recursively?
2. What are the smaller sub-problems?

**Lemma**

Let $G$ be a directed graph with arbitrary edge lengths. If $s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k$ is a shortest path from $s$ to $v_k$ then for $1 \leq i < k$:

1. $s = v_0 \rightarrow v_1 \rightarrow v_2 \ldots \rightarrow v_i$ is a shortest path from $s$ to $v_i$
1. Compute the shortest path distance from $s$ to $t$ recursively?
2. What are the smaller sub-problems?

**Lemma**

Let $G$ be a directed graph with arbitrary edge lengths. If $s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k$ is a shortest path from $s$ to $v_k$ then for $1 \leq i < k$:

1. $s = v_0 \rightarrow v_1 \rightarrow v_2 \ldots \rightarrow v_i$ is a shortest path from $s$ to $v_i$

Sub-problem idea: paths of fewer hops/edges
Hop-based Recursion: Bellman-Ford Algorithm

Single-source problem: fix source $s$. Assume that all nodes can be reached by $s$ in $G$. (Remove nodes unreachable from $s$).

$d(v, k)$: shortest walk length from $s$ to $v$ using at most $k$ edges.
Hop-based Recursion: Bellman-Ford Algorithm

Single-source problem: fix source \( s \).
Assume that all nodes can be reached by \( s \) in \( G \). (Remove nodes unreachable from \( s \)).

\( d(v, k) \): shortest walk length from \( s \) to \( v \) using at most \( k \) edges.

Recursion for \( d(v, k) \):
Hop-based Recursion: Bellman-Ford Algorithm

Single-source problem: fix source $s$.
Assume that all nodes can be reached by $s$ in $G$. (Remove nodes unreachable from $s$).

$d(v, k)$: shortest walk length from $s$ to $v$ using at most $k$ edges.

Recursion for $d(v, k)$:

\[
d(v, k) = \min \begin{cases} 
\min_{u \in V} (d(u, k - 1) + \ell(u, v)), \\
 d(v, k - 1) 
\end{cases}
\]

Base case: $d(s, 0) = 0$ and $d(v, 0) = \infty$ for all $v \neq s$. 
Lemma

Assume $s$ can reach all nodes in $G = (V, E)$. Then,

1. There is a negative length cycle in $G$ iff $d(v, n) < d(v, n - 1)$ for some node $v \in V$.

2. If there is no negative length cycle in $G$ then $\text{dist}(s, v) = d(v, n - 1)$ for all $v \in V$. 
Bellman-Ford Algorithm

**for each** \( u \in V \) **do**

\[
d(u, 0) \leftarrow \infty
\]

\[
d(s, 0) \leftarrow 0
\]
Bellman-Ford Algorithm

for each $u \in V$ do
    $d(u, 0) \leftarrow \infty$
    $d(s, 0) \leftarrow 0$

for $k = 1$ to $n$ do
    for each $v \in V$ do
        $d(v, k) \leftarrow d(v, k - 1)$
    for each edge $(u, v) \in \text{In}(v)$ do
        $d(v, k) = \min\{d(v, k), d(u, k - 1) + \ell(u, v)\}$
Bellman-Ford Algorithm

for each $u \in V$ do
  $d(u, 0) \leftarrow \infty$
  $d(s, 0) \leftarrow 0$

for $k = 1$ to $n$ do
  for each $v \in V$ do
    for each edge $(u, v) \in \text{In}(v)$ do
      $d(v, k) = \min\{d(v, k), d(u, k - 1) + \ell(u, v)\}$

for each $v \in V$ do
  $\text{dist}(s, v) \leftarrow d(v, n - 1)$
  If $d(v, n) < d(v, n - 1)$
  Return ‘‘Negative Cycle in $G$’’

Running time: $O(mn)$

Space: $O(n^2)$

Space can be reduced to $O(m + n)$. 
Bellman-Ford Algorithm

for each $u \in V$ do
    $d(u, 0) \leftarrow \infty$
    $d(s, 0) \leftarrow 0$

for $k = 1$ to $n$ do
    for each $v \in V$ do
        $d(v, k) \leftarrow d(v, k - 1)$
        for each edge $(u, v) \in \text{In}(v)$ do
            $d(v, k) = \min\{d(v, k), d(u, k - 1) + \ell(u, v)\}$

for each $v \in V$ do
    $\text{dist}(s, v) \leftarrow d(v, n - 1)$
    If $d(v, n) < d(v, n - 1)$
    Return ‘‘Negative Cycle in $G$’’

Running time: $O(mn)$

Space: $O(n^2)$

Space can be reduced to $O(m + n)$. 
Bellman-Ford Algorithm

for each $u \in V$ do
    \[d(u, 0) \leftarrow \infty\]
    \[d(s, 0) \leftarrow 0\]

for $k = 1$ to $n$ do
    for each $v \in V$ do
        \[d(v, k) \leftarrow d(v, k - 1)\]
        for each edge $(u, v) \in \text{In}(v)$ do
            \[d(v, k) = \min\{d(v, k), d(u, k - 1) + \ell(u, v)\}\]

for each $v \in V$ do
    \[\text{dist}(s, v) \leftarrow d(v, n - 1)\]
    If $d(v, n) < d(v, n - 1)$
        Return ‘‘Negative Cycle in $G$’’

Running time: $O(mn)$
Bellman-Ford Algorithm

for each $u \in V$ do
\[ d(u, 0) \leftarrow \infty \]
\[ d(s, 0) \leftarrow 0 \]

for $k = 1$ to $n$ do
\[ \text{for each } v \in V \text{ do} \]
\[ d(v, k) \leftarrow d(v, k - 1) \]
\[ \text{for each edge } (u, v) \in In(v) \text{ do} \]
\[ d(v, k) = \min\{d(v, k), d(u, k - 1) + \ell(u, v)\} \]

for each $v \in V$ do
\[ \text{dist}(s, v) \leftarrow d(v, n - 1) \]
If $d(v, n) < d(v, n - 1)$
\[ \text{Return ‘‘Negative Cycle in } G’’ \]

Running time: $O(mn)$ Space:
Bellman-Ford Algorithm

\[
\text{for each } u \in V \text{ do} \\
\quad d(u, 0) \leftarrow \infty \\
\quad d(s, 0) \leftarrow 0 \\
\text{for } k = 1 \text{ to } n \text{ do} \\
\quad \text{for each } v \in V \text{ do} \\
\quad \quad d(v, k) \leftarrow d(v, k - 1) \\
\quad \quad \text{for each edge } (u, v) \in \text{ln}(v) \text{ do} \\
\quad \quad \quad d(v, k) = \min\{d(v, k), d(u, k - 1) + \ell(u, v)\} \\
\text{for each } v \in V \text{ do} \\
\quad \text{dist}(s, v) \leftarrow d(v, n - 1) \\
\quad \text{If } d(v, n) < d(v, n - 1) \text{ Return ‘‘Negative Cycle in } G’’
\]

Running time: \(O(mn)\) Space: \(O(n^2)\)
Bellman-Ford Algorithm

for each \( u \in V \) do
\[
\begin{align*}
  d(u, 0) &\leftarrow \infty \\
  d(s, 0) &\leftarrow 0
\end{align*}
\]

for \( k = 1 \) to \( n \) do
  for each \( v \in V \) do
    \[
    d(v, k) \leftarrow d(v, k - 1)
    \]
    for each edge \((u, v) \in \text{In}(v)\) do
      \[
      d(v, k) = \min\{d(v, k), d(u, k - 1) + \ell(u, v)\}
      \]

for each \( v \in V \) do
  \[
  \text{dist}(s, v) \leftarrow d(v, n - 1)
  \]
  If \( d(v, n) < d(v, n - 1) \)
  Return ‘‘Negative Cycle in \( G \)’’

Running time: \( O(mn) \) Space: \( O(n^2) \)
Space can be reduced to \( O(m + n) \).
Bellman-Ford with Space Saving

\begin{algorithm}
\For {each $u \in V$}{
    $d(u) \leftarrow \infty$
\}
\For {$k = 1$ to $n - 1$}{
    \For {each $v \in V$}{
        \For {each edge $(u, v) \in \text{In}(v)$}{
            $d(v) = \min\{d(v), d(u) + \ell(u, v)\}$
        }
    }
\For {each $v \in V$}{
    \For {each edge $(u, v) \in \text{In}(v)$}{
        \If {$d(v) > d(u) + \ell(u, v)$}{
            Output ‘‘Negative Cycle’’
        }
    }
\}
\For {each $v \in V$}{
    $\text{dist}(s, v) \leftarrow d(v)$
\}
\end{algorithm}
Problem

Given a directed graph $G = (V, E)$ with non-negative edge lengths $\ell : E \rightarrow R^+$, describe an algorithm that finds the shortest cycle in $G$ that contains a specific node $s$. 
Problem

Given a directed graph $G = (V, E)$ with non-negative edge lengths $\ell : E \rightarrow R^+$. Describe an algorithm to find the shortest cycle containing $s$ with at most $k$ edges.
Randomized Algorithms

Input $x$ → Deterministic Algorithm → Output $y$

Randomized Algorithm

random bits $r$
Randomized Algorithms

Deterministic Algorithm

Input $x$ → Output $y$

Randomized Algorithm

Input $x$ → Output $y_r$

Input $x$ → random bits $r$ → Output $y_r$
Types of Randomized Algorithms

Typically one encounters the following types:

1. **Las Vegas randomized algorithms:** for a given input $x$, the output of the algorithm is always correct but the running time is a random variable. Analyze expected running time.
2. **Monte Carlo randomized algorithms:** for a given input $x$, the running time is deterministic but the output is random; correct with some probability. Analyze the probability of the correct output (and also the running time).
3. Algorithms whose running time and output may both be random.
Types of Randomized Algorithms

Typically one encounters the following types:

1. **Las Vegas randomized algorithms**: for a given input $x$ output of algorithm is always correct but the running time is a random variable. Analyze expected running time.

2. **Monte Carlo randomized algorithms**: for a given input $x$ the running time is deterministic but the output is random; correct with some probability. Analyze the probability of the correct output (and also the running time).

3. Algorithms whose running time and output may both be random.
Consider a deterministic algorithm \( A \) that is trying to find an element in an array \( X \) of size \( n \). At every step it is allowed to ask the value of one cell in the array, and the adversary is allowed after each such ping, to shuffle elements around in the array in any way it seems fit. For the best possible deterministic algorithm the number of rounds it has to play this game till it finds the required element is

(A) \( O(1) \)
(B) \( O(n) \)
(C) \( O(n \log n) \)
(D) \( O(n^2) \)
(E) \( \infty \)
Consider an algorithm \textbf{randFind} that is trying to find an element in an array \( X \) of size \( n \). At every step it asks the value of one random cell in the array, and the adversary is allowed after each such ping, to shuffle elements around in the array in any way it seems fit. This algorithm would stop in expectation after

(A) \( O(1) \)
(B) \( O(\log n) \)
(C) \( O(n) \)
(D) \( O(n^2) \)
(E) \( \infty \).

steps.
Consider the problem of finding an “approximate median” of an unsorted array $A[1..n]$: an element of $A$ with rank between $n/4$ and $3n/4$.

- Finding an approximate median is not any easier than a proper median.
- $n/2$ elements of $A$ qualify as approximate medians and hence a random element is good with probability $1/2!$
Part IV

Basics of Randomization
Definition

A discrete probability space is a pair \((\Omega, \Pr)\) consists of finite set \(\Omega\) of elementary events and function \(p : \Omega \rightarrow [0, 1]\) which assigns a probability \(\Pr[\omega]\) for each \(\omega \in \Omega\) such that \(\sum_{\omega \in \Omega} \Pr[\omega] = 1\).

Example

An unbiased coin. \(\Omega = \{H, T\}\) and \(\Pr[H] = \Pr[T] = 1/2\).
Events

Definition
Event is a collection of elementary events. The probability of an event $A \subset \Omega$, denoted by $\Pr[A]$, is $\sum_{\omega \in A} \Pr[\omega]$. 
Events

**Definition**
Event is a collection of elementary events. The probability of an event $A \subset \Omega$, denoted by $\Pr[A]$, is $\sum_{\omega \in A} \Pr[\omega]$.

**Union Bound**
For any two events $\mathcal{E}$ and $\mathcal{F}$, we have that
$$\Pr[\mathcal{E} \cup \mathcal{F}] \leq \Pr[\mathcal{E}] + \Pr[\mathcal{F}]$$.
Definition

Event is a collection of elementary events. The probability of an event $A \subset \Omega$, denoted by $\Pr[A]$, is $\sum_{\omega \in A} \Pr[\omega]$.

Union Bound

For any two events $\mathcal{E}$ and $\mathcal{F}$, we have that
$$\Pr[\mathcal{E} \cup \mathcal{F}] \leq \Pr[\mathcal{E}] + \Pr[\mathcal{F}]$$

Independence

Events $A$ and $B$ are called independent if
$$\Pr[A \cap B] = \Pr[A] \Pr[B]$$
Random Variables

**Definition**

Given a probability space \((\Omega, \Pr)\) a (real-valued) random variable \(X\) over \(\Omega\) is a function \(X : \Omega \rightarrow \mathbb{R}\).
Random Variables

**Definition**
Given a probability space \((\Omega, \Pr)\) a (real-valued) random variable \(X\) over \(\Omega\) is a function \(X : \Omega \to \mathbb{R}\).

**Definition (Expectation: Average of X as per Pr)**
Expectation of \(X\), \(E[X]\), is defined as \(\sum_{\omega \in \Omega} \Pr[\omega] X(\omega)\). If \(S\) is the set of all values that \(X\) takes, then expectation can also be written as \(\sum_{x \in S} x \Pr[X = x]\).
Random Variables

**Definition**

Given a probability space \((\Omega, \Pr)\) a (real-valued) random variable \(X\) over \(\Omega\) is a function \(X : \Omega \rightarrow \mathbb{R}\).

**Definition (Expectation: Average of \(X\) as per \(\Pr\))**

Expectation of \(X\), \(\mathbb{E}[X]\), is defined as \(\sum_{\omega \in \Omega} \Pr[\omega] X(\omega)\).

If \(S\) is the set of all values that \(X\) takes, then expectation can also be written as \(\sum_{x \in S} x \Pr[X = x]\).

**Linearity of Expectation**

Given two random variables \(X_1\) and \(X_2\),

\[\mathbb{E}[X_1 + X_2] = \mathbb{E}[X_1] + \mathbb{E}[X_2].\]
Independence of Random Variables

Random variables $X$ and $Y$ are said to be independent if

$$\forall x, y, \quad \Pr[X = x \wedge Y = y] = \Pr[X = x] \cdot \Pr[Y = y]$$

**Multiplication**

If $X$ and $Y$ are independent then $E[XY] = E[X]E[Y]$.
Part V

Randomized Quick Sort
Randomized **QuickSort**

1. Pick a pivot element *uniformly at random* from the array.
2. Split array into 3 subarrays: those smaller than pivot, those larger than pivot, and the pivot itself.
3. Recursively sort the subarrays, and concatenate them.
Given array $A$ of size $n$, let $Q(A)$ be number of comparisons of randomized QuickSort on $A$.

Note that $Q(A)$ is a random variable.

Let $A^i_{\text{left}}$ and $A^i_{\text{right}}$ be the left and right arrays obtained if:

Let $X_i$ be indicator random variable, which is set to 1 if the pivot is of rank $i$ in $A$, else zero.

$$Q(A) = n + \sum_{i=1}^{n} X_i \cdot \left( Q(A^i_{\text{left}}) + Q(A^i_{\text{right}}) \right).$$
Analysis via Recurrence

1. Given array $A$ of size $n$, let $Q(A)$ be number of comparisons of randomized QuickSort on $A$.

2. Note that $Q(A)$ is a random variable.

3. Let $A^i_{\text{left}}$ and $A^i_{\text{right}}$ be the left and right arrays obtained if:

   Let $X_i$ be indicator random variable, which is set to 1 if the pivot is of rank $i$ in $A$, else zero.

   $$Q(A) = n + \sum_{i=1}^{n} X_i \cdot \left( Q(A^i_{\text{left}}) + Q(A^i_{\text{right}}) \right).$$

Since each element of $A$ has probability exactly of $1/n$ of being chosen:

$$E[X_i] = \Pr[\text{pivot is the element with rank } i] = 1/n.$$
Independence of Random Variables

**Lemma**

Random variables $X_i$ is independent of random variables $Q(A_{\text{left}}^i)$ as well as $Q(A_{\text{right}}^i)$, i.e.

$$E\left[X_i \cdot Q(A_{\text{left}}^i)\right] = E[X_i] E\left[Q(A_{\text{left}}^i)\right]$$

$$E\left[X_i \cdot Q(A_{\text{right}}^i)\right] = E[X_i] E\left[Q(A_{\text{right}}^i)\right]$$

**Proof.**

This is because the algorithm, while recursing on $Q(A_{\text{left}}^i)$ and $Q(A_{\text{right}}^i)$ uses new random coin tosses that are independent of the coin tosses used to decide the first pivot. Only the latter decides value of $X_i$. 

□
Analysis via Recurrence

Let \( T(n) = \max_{A:|A|=n} \mathbb{E}[Q(A)] \) be the worst-case expected running time of randomized \texttt{QuickSort} on arrays of size \( n \).

We have, for any \( A \):

\[
Q(A) = n + \sum_{i=1}^{n} X_i \left( Q(A^i_{\text{left}}) + Q(A^i_{\text{right}}) \right)
\]
Analysis via Recurrence

Let \( T(n) = \max_{|A|=n} E[Q(A)] \) be the worst-case expected running time of randomized \texttt{QuickSort} on arrays of size \( n \).

We have, for any \( A \):

\[
Q(A) = n + \sum_{i=1}^{n} X_i \left( Q(A_{\text{left}}^i) + Q(A_{\text{right}}^i) \right)
\]

By linearity of expectation, and independence random variables:

\[
E[Q(A)] = n + \sum_{i=1}^{n} E[X_i] \left( E[Q(A_{\text{left}}^i)] + E[Q(A_{\text{right}}^i)] \right)
\]
Analysis via Recurrence

Let \( T(n) = \max_{|A| = n} E[Q(A)] \) be the worst-case expected running time of randomized QuickSort on arrays of size \( n \).

We have, for any \( A \):

\[
Q(A) = n + \sum_{i=1}^{n} X_i \left( Q(A_{\text{left}}^i) + Q(A_{\text{right}}^i) \right)
\]

By linearity of expectation, and independence random variables:

\[
E[Q(A)] = n + \sum_{i=1}^{n} E[X_i] \left( E[Q(A_{\text{left}}^i)] + E[Q(A_{\text{right}}^i)] \right)
\]

\[
\leq n + \sum_{i=1}^{n} \frac{1}{n} \left( T(i - 1) + T(n - i) \right).
\]
Analysis via Recurrence

Let \( T(n) = \max_{A:|A|=n} \mathbb{E}[Q(A)] \) be the worst-case expected running time of randomized \texttt{QuickSort} on arrays of size \( n \).

We derived:

\[
\mathbb{E}[Q(A)] \leq n + \sum_{i=1}^{n} \frac{1}{n} (T(i - 1) + T(n - i)).
\]

Note that above holds for any \( A \) of size \( n \). Therefore
Analysis via Recurrence

Let \( T(n) = \max_{A:|A|=n} \mathbb{E}[Q(A)] \) be the worst-case expected running time of randomized QuickSort on arrays of size \( n \). We derived:

\[
\mathbb{E}[Q(A)] \leq n + \sum_{i=1}^{n} \frac{1}{n} (T(i - 1) + T(n - i)).
\]

Note that above holds for any \( A \) of size \( n \). Therefore

\[
\max_{A:|A|=n} \mathbb{E}[Q(A)] = T(n) \leq n + \sum_{i=1}^{n} \frac{1}{n} (T(i - 1) + T(n - i)).
\]
Solving the Recurrence

\[ T(n) \leq n + \sum_{i=1}^{n} \frac{1}{n} (T(i - 1) + T(n - i)) \]

with base case \( T(1) = 0 \).
Solving the Recurrence

\[ T(n) \leq n + \sum_{i=1}^{n} \frac{1}{n} (T(i - 1) + T(n - i)) \]

with base case \( T(1) = 0 \).

Lemma

\( T(n) = O(n \log n) \).
Solving the Recurrence

\[ T(n) \leq n + \sum_{i=1}^{n} \frac{1}{n} (T(i - 1) + T(n - i)) \]

with base case \( T(1) = 0 \).

**Lemma**

\( T(n) = O(n \log n) \).

**Proof.**

(Guess and) Verify by induction.
Part VI

Inequalities
Markov’s Inequality

**Markov’s inequality**

Let $X$ be a **non-negative** random variable over a probability space $(\Omega, \Pr)$. For any $a > 0$,

$$\Pr[X \geq a] \leq \frac{\mathbb{E}[X]}{a}$$
Variance

Variance of $X$ is the measure of how much does it deviate from its mean value. Formally,

$$\text{Var}(X) = \mathbb{E}[(X - \mathbb{E}[X])^2] = \mathbb{E}[X^2] - \mathbb{E}[X]^2$$

Chebyshev’s Inequality

Given $a \geq 0$, $\Pr[|X - \mathbb{E}[X]| \geq a] \leq \frac{\text{Var}(X)}{a^2}$
Chebyshev’s Inequality

Variance

Variance of $X$ is the measure of how much does it deviate from its mean value. Formally,

$$Var(X) = E[(X - E[X])^2] = E[X^2] - E[X]^2$$

Chebyshev’s Inequality

Given $a \geq 0$, $Pr[|X - E[X]| \geq a] \leq \frac{Var(X)}{a^2}$

If $X$ and $Y$ are independent then

$$Var(X + Y) = Var(X) + Var(Y).$$
Chebyshev’s Inequality: Under Mutual Independence

Let $X_1, \ldots, X_k$ be $k$ independent random variables such that, for each $i \in [1, k]$, $X_i$ equals 1 with probability $p_i$, and 0 with probability $(1 - p_i)$. Let $X = \sum_{i=1}^{k} X_i$ and $\mu = \mathbb{E}[X] = \sum_i p_i$. For any $0 < \delta < 1$, it holds that:

$$\text{Var}(X) \leq \mu \Rightarrow \Pr[|X - \mu| \geq a] \leq \frac{\text{Var}(X)}{a^2} < \frac{\mu}{a^2}$$
Chebyshev’s Inequality: Under Mutual Independence

Let $X_1, \ldots, X_k$ be $k$ independent random variables such that, for each $i \in [1, k]$, $X_i$ equals 1 with probability $p_i$, and 0 with probability $(1 - p_i)$. Let $X = \sum_{i=1}^{k} X_i$ and $\mu = \mathbb{E}[X] = \sum_i p_i$. For any $0 < \delta < 1$, it holds that:

$$\operatorname{Var}(X) \leq \mu \Rightarrow \Pr[|X - \mu| \geq a] \leq \frac{\operatorname{Var}(X)}{a^2} < \frac{\mu}{a^2}$$

For $\delta > 0$, $\Pr[X \geq (1 + \delta)\mu] \leq \frac{1}{\delta^2\mu}$

For $0 < \delta < 1$, $\Pr[X \leq (1 - \delta)\mu] \leq \frac{1}{\delta^2\mu}$
Chernoff Bound

Let $X_1, \ldots, X_k$ be $k$ independent random variables such that, for each $i \in [1, k]$, $X_i$ equals 1 with probability $p_i$, and 0 with probability $(1 - p_i)$. Let $X = \sum_{i=1}^{k} X_i$ and $\mu = \mathbb{E}[X] = \sum_i p_i$. For any $0 < \delta < 1$, it holds that:
Chernoff Bound

Let \( X_1, \ldots, X_k \) be \( k \) independent random variables such that, for each \( i \in [1, k] \), \( X_i \) equals 1 with probability \( p_i \), and 0 with probability \( (1 - p_i) \). Let \( X = \sum_{i=1}^{k} X_i \) and \( \mu = \mathbb{E}[X] = \sum_{i} p_i \). For any \( 0 < \delta < 1 \), it holds that:

\[
\Pr[X \geq (1 + \delta)\mu] \leq e^{-\frac{\delta^2}{3}\mu} \quad \text{and} \quad \Pr[X \leq (1 - \delta)\mu] \leq e^{-\frac{\delta^2}{2}\mu}
\]
Chernoff Bound

Let $X_1, \ldots, X_k$ be $k$ independent random variables such that, for each $i \in [1, k]$, $X_i$ equals 1 with probability $p_i$, and 0 with probability $(1 - p_i)$. Let $X = \sum_{i=1}^{k} X_i$ and $\mu = \mathbb{E}[X] = \sum_i p_i$. For any $0 < \delta < 1$, it holds that:

$$\Pr[X \geq (1 + \delta)\mu] \leq e^{-\frac{\delta^2 \mu}{3}} \quad \text{and} \quad \Pr[X \leq (1 - \delta)\mu] \leq e^{-\frac{\delta^2 \mu}{2}}$$

Tighter bound

For any $\delta > 0$, $\Pr[X \geq (1 + \delta)\mu] \leq \left(\frac{e^\delta}{(1 + \delta)^{(1 + \delta)}}\right)^\mu$

$\Pr[X \geq (1 - \delta)\mu] \leq \left(\frac{e^{-\delta}}{(1 - \delta)^{(1 - \delta)}}\right)^\mu$
Problem: Approximate Median

Suppose you are presented with a very large set $S$ of real numbers, and you would like to approximate the median of these numbers by sampling. Let $|S| = n$. We say $x$ is an $\epsilon$-approximate median of $S$ if at least $(1/2 - \epsilon)n$ are less than $x$ and at least $(1/2 - \epsilon)n$ are greater than $x$. Consider an algorithm that samples a number $c$ times u.a.r. from $S$, forms set $S'$ of sampled numbers, and outputs a median of $S'$. Show that for the algorithm to return $\epsilon$-approximate median w.p. at least $(1 - \delta)$, it suffices to have sample size $c$ that is an absolute constant, independent of $n$. 