Average-Case Analysis and Randomization

CSci 3110

Textbook readings:

Chapter 2
Interspersed throughout the book
Hamiltonian cycle
Overview

**Design principle:**
- Make random choices and hope they are good

**Analysis techniques:**
- Average-case analysis
- Probability theory

**Problems:**
- Sorting (Quicksort & Bucket Sort)
- Selection
- Space partitions
Selection

partition on the mean

\[ L \leq x \leq R \]

search where the rank should be!

\[ > \frac{3}{10} - 6 \]

\[ \sim \frac{2}{10} \]
Quicksort Revisited

The problem with deterministic Quicksort:

The running time is $\mathcal{O}(n \lg n)$, but the algorithm for finding a pivot is non-trivial.
Quicksort Revisited

The problem with deterministic Quicksort:

The running time is $O(n \lg n)$, but the algorithm for finding a pivot is non-trivial.

Remedy:

Blindly use the first input element as the pivot.
The problem with deterministic Quicksort:

The running time is $O(n \log n)$, but the algorithm for finding a pivot is non-trivial.

Remedy:

Blindly use the first input element as the pivot.

```
SIMPLE-QUICKSORT(A)
1  if |A| ≤ 1
2     then return A
3  else p ← A[1]
4  Partition A into three pieces:
5     L = {x ∈ A | x < p}
6     {p}
7     R = {x ∈ A \ {p} | x ≥ p}
5  L' ← QUICKSORT(L)
6  R' ← QUICKSORT(R)
7  return L' o {p} o R'
```
Quicksort Revisited

The problem with deterministic Quicksort:

The running time is $O(n \lg n)$, but the algorithm for finding a pivot is non-trivial.

Remedy:

Blindly use the first input element as the pivot.

```
SIMPLE-QUICKSORT(A)
1  if |A| ≤ 1
2    then return A
3  else $p \leftarrow A[1]$
4  Partition $A$ into three pieces:
5      $L = \{ x \in A \mid x < p \}$
6      $\{p\}$
7      $R = \{ x \in A \setminus \{p\} \mid x \geq p \}$
8  $L' \leftarrow$ QUICKSORT($L$)
9  $R' \leftarrow$ QUICKSORT($R$)
10 $\text{return } L' \circ \{p\} \circ R'$
```

Running time:
Quicksort Revisited

The problem with deterministic Quicksort:

The running time is $O(n \lg n)$, but the algorithm for finding a pivot is non-trivial.

Remedy:

Blindly use the first input element as the pivot.

```
SIMPLE-QUICKSORT(A)
1   if |A| ≤ 1
2       then return A
3   else $p \leftarrow A[1]$
4   Partition A into three pieces:
5       $L = \{x \in A \mid x < p\}$
6       $\{p\}$
7       $R = \{x \in A \setminus \{p\} \mid x \geq p\}$
8   $L' \leftarrow \text{QUICKSORT}(L)$
9   $R' \leftarrow \text{QUICKSORT}(R)$
10  return $L' \circ \{p\} \circ R'$
```

Running time:

- Worst case: $O(n^2)$
Quicksort Revisited

The problem with deterministic Quicksort:

The running time is $O(n \log n)$, but the algorithm for finding a pivot is non-trivial.

Remedy:

Blindly use the first input element as the pivot.

```
SIMPLE-QUICKSORT(A)
1  if |A| ≤ 1
2    then return A
3  else $p \leftarrow A[1]$
4  Partition A into three pieces:
5    $L = \{x \in A \mid x < p\}$
6    $\{p\}$
7    $R = \{x \in A \setminus \{p\} \mid x \geq p\}$
8  $L' \leftarrow$ QUICKSORT($L$)
9  $R' \leftarrow$ QUICKSORT($R$)
10 return $L' \circ \{p\} \circ R'$
```

Running time:

- Worst case: $O(n^2)$
- Average case: $O(n \log n)$
Average-Case Analysis of Simple-Quicksort

**Simple-Quicksort** $(A)$

1. **if** $|A| \leq 1$
2. **then return** $A$
3. **else** $p \leftarrow A[1]$
4. Partition $A$ into three pieces:
   - $L = \{x \in A \mid x < p\}$
   - $\{p\}$
   - $R = \{x \in A \setminus \{p\} \mid x \geq p\}$
5. $L' \leftarrow \text{QUICKSORT}(L)$
6. $R' \leftarrow \text{QUICKSORT}(R)$
7. **return** $L' \circ \{p\} \circ R'$

**Lemma:** The expected running time of algorithm Simple-Quicksort is $O(n \lg n)$. 

CSci 3110 • Average-Case Analysis and Randomization • 4/18
Observation: The running time of Simple-Quicksort is $O(n + X)$, where $X$ is the number of comparisons it performs.

$\sum_{i=0}^{n} \text{ constants for each pivot}$

$n \sim n^2$
**Observation:** The running time of Simple-Quicksort is $O(n + X)$, where $X$ is the number of comparisons it performs.

\[ \therefore \text{It suffices to show that } E(X) = O(n \log n). \]
**Observation:** The running time of Simple-Quicksort is $O(n + X)$, where $X$ is the number of comparisons it performs.

∴ It suffices to show that $E(X) = O(n \lg n)$.

**Observation:** Any two elements are compared at most once.
Let $a_1 < a_2 < \cdots < a_n$ be the elements in $A$.

Let $E_{ij} = \text{“}a_i \text{ and } a_j \text{ are compared”}$

Let $X_{ij} = I(E_{ij})$
Let \( a_1 < a_2 < \cdots < a_n \) be the elements in \( A \).

Let \( E_{ij} = \) “\( a_i \) and \( a_j \) are compared”

Let \( X_{ij} = I(E_{ij}) \)

\[
E(X) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E(X_{ij})
\]
Let \( a_1 < a_2 < \cdots < a_n \) be the elements in \( A \).

Let \( E_{ij} = \text{“} a_i \text{ and } a_j \text{ are compared”} \)

Let \( X_{ij} = I(E_{ij}) \)

\[
E(X) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E(X_{ij})
\]

**Lemma:** \( E(X_{ij}) = \Pr(E_{ij}) = \frac{2}{j-i+1} \).
\[
\frac{2}{2} \cdot (n-1) + \frac{2}{3} \cdot (n-2) + \frac{2}{4} \cdot (n-3) + \frac{2}{5} \cdot (n-4) + \frac{2}{6} \cdot (n-5)
\]
\[
+ \frac{2}{7} \cdot n - 6 + \frac{2}{8} \cdot n - 7
\]

\[
\langle 2n \rangle \quad \langle \frac{1}{2} \cdot 4 \cdot n \rangle \leq 2n \quad \geq 2n
\]

\[
\text{first 2} \quad \text{next 4} \quad \text{next 8}
\]
$$\sum_{i=0}^{\log n} 2^n = \Theta(n \log n)$$
Let \( a_1 < a_2 < \cdots < a_n \) be the elements in \( A \).

Let \( E_{ij} = "a_i \text{ and } a_j \text{ are compared}" \)

Let \( X_{ij} = I(E_{ij}) \)

\[
E(X) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E(X_{ij})
\]

**Lemma:** \( E(X_{ij}) = Pr(E_{ij}) = \frac{2}{j-i+1} \).

**Corollary:** \( E(X) = \mathcal{O}(n \lg n) \).
Can we sort faster than $\mathcal{O}(n \lg n)$ time?
Can we sort faster than $O(n \lg n)$ time?

Assumption: Elements are numbers drawn uniformly at random from $(0, 1]$. 


Sorting in Linear Time

*Can we sort faster than $O(n \lg n)$ time?*

**Assumption:** Elements are numbers drawn uniformly at random from $(0, 1]$.

---

**Bucket-Sort($A$, $n$)**

1. Allocate an array $B$ of size $n$; each entry $B[i]$ stores a pointer to an initially empty linked list.
2. For $i \leftarrow 1$ to $n$
3. Do Insert $A[i]$ into list $B[{\lceil n \cdot A[i] \rceil}]$
4. For $i \leftarrow 1$ to $n$
5. Do Sort $B[i]$ using Insertion Sort
6. Append the sorted list to the output

---

$n = 5$

\[
\begin{array}{cccccc}
0.2 & 0.5 & 0.55 & 0.3 & 0.7 \\
\end{array}
\]

---

\[
\begin{array}{cccccc}
0.2 & 0.3 & 0.5 & 0.55 & 0.7 \\
\end{array}
\]
Can we sort faster than $O(n \lg n)$ time?

Assumption: Elements are numbers drawn uniformly at random from $(0, 1]$.

**Bucket-Sort** $(A, n)$

1. Allocate an array $B$ of size $n$; each entry $B[i]$ stores a pointer to an initially empty linked list.
2. for $i \leftarrow 1$ to $n$
3.   do Insert $A[i]$ into list $B[\lceil n \cdot A[i] \rceil]$
4. for $i \leftarrow 1$ to $n$
5.   do Sort $B[i]$ using Insertion Sort
6.   Append the sorted list to the output

**Observation:** The worst-case running time of Bucket Sort is $O(n^2)$. 

CSci 3110 • Average-Case Analysis and Randomization • 7/18
Sorting in Linear Time

Can we sort faster than $\mathcal{O}(n \lg n)$ time?

Assumption: Elements are numbers drawn uniformly at random from $(0, 1]$.

**Bucket-Sort($A$, $n$)**

1. Allocate an array $B$ of size $n$; each entry $B[i]$ stores a pointer to an initially empty linked list.
2. for $i \leftarrow 1$ to $n$
3. do Insert $A[i]$ into list $B[\lceil n \cdot A[i] \rceil]$
4. for $i \leftarrow 1$ to $n$
5. do Sort $B[i]$ using Insertion Sort
6. Append the sorted list to the output

**Observation:** The worst-case running time of Bucket Sort is $\mathcal{O}(n^2)$.

**Lemma:** If the elements of $A$ are drawn uniformly at random from $(0, 1]$, the expected running time of Bucket Sort is $\mathcal{O}(n)$. 
Since the algorithms are deterministic, they have worst-case inputs.

Our analysis **assumes** a uniform distribution over all possible inputs. We don’t know the real distribution.
Randomization

*The remedy:*

*Impose* the random distribution.

(Let the algorithm make random choices.)
Randomization

The remedy:

*Impose* the random distribution.
(Let the algorithm make random choices.)

Benefits:

- No more assumptions about given input distribution.
- Resulting algorithms are still extremely simple.
Randomization

The remedy:

*Impose* the random distribution.
(Let the algorithm make random choices.)

Benefits:

- No more assumptions about given input distribution.
- Resulting algorithms are still extremely simple.

Issues:

- Random permutations can usually be generated without altering the meaning of the input.
- The distribution of the values of the input elements *cannot* be controlled. (Bucket sort)
Randomized Quicksort

\textbf{RANDOMIZED-QUICKSORT}(A)
\begin{enumerate}
\item \textbf{if} $|A| \leq 1$
\item \textbf{then return} $A$
\item \textbf{else} $p \leftarrow A[\text{RANDOM}(1, |A|)]$
\item \text{Partition} $A$ \text{into three pieces:}
\begin{itemize}
\item $L = \{x \in A \mid x < p\}$
\item $\{p\}$
\item $R = \{x \in A \setminus \{p\} \mid x \geq p\}$
\end{itemize}
\item $L' \leftarrow \text{QUICKSORT}(L)$
\item $R' \leftarrow \text{QUICKSORT}(R)$
\item \textbf{return} $L' \circ \{p\} \circ R'$
\end{enumerate}

\textbf{Lemma:} The expected running time of algorithm Randomized-Quicksort is $O(n \lg n)$. 
\( \frac{2}{4} \)
**Randomized Selection**

**Randomized-Select** \((A, k)\)

1. if \(|A| = 1\) then return \(A[1]\)
2. \(p \leftarrow A[\text{Random}(1, |A|)]\)
3. Partition \(A\) into three pieces:
   - \(L = \{x \in A \mid x < p\}\)
   - \(\{p\}\)
   - \(R = \{x \in A \setminus \{p\} \mid x \geq p\}\)
4. if \(k = |L| + 1\) then return \(p\)
5. else if \(k < |L| + 1\) then return **Randomized-Select**\((L, k)\)
6. else return **Randomized-Select**\((R, k - |L| - 1)\)

**Lemma:** The expected running time of algorithm Randomized-Select is \(\mathcal{O}(n)\).
1 3 6 7 9 8 10 5 2 4

Select the median

Something is a good pivot if its rank is between $1/4n$ and $3/4n$.

$1/4n$  $3/4n$
\[ \frac{n}{2} \text{ "good" pivots} \]

50% of picking a good pivot

Look at 2 recursive calls \( E(x) \) where \( x \) is the number of good pivots chosen,

\[ \frac{x}{2} \cdot \frac{1}{2} = 1 \]

So expect 1 good pivot every 2 iterations expected \( O(n) \) running time
Rendering Scenes


Rendering a 3D scene on screen:

- Some objects hide other objects from view.
- Simplest algorithm: “Painter’s algorithm” = Render objects back to front and paint over
Rendering Scenes


Rendering a 3D scene on screen:

- Some objects hide other objects from view.
- Simplest algorithm: “Painter’s algorithm” = Render objects back to front and paint over

Problem: How to decide quickly in which order to render objects. What’s “back to front” for a given viewpoint?
Rendering Scenes


Rendering a 3D scene on screen:

- Some objects hide other objects from view.
- Simplest algorithm: “Painter’s algorithm” = Render objects back to front and paint over

Problem: How to decide quickly in which order to render objects. What’s “back to front” for a given viewpoint?
Recursively partition the plane using lines.

Represent partition using rooted tree.

Every internal node $v$ corresponds to a region $R(v)$.

Leaves below $v$ store (pieces of) line segments contained in $R(v)$. 
Binary Space Partitions
The lines used to partition the plane are spanned by the line segments.

Every node in the partition tree represents a (piece of a) line segment $s$, a region $R(s)$, and the line $\ell(s)$ spanned by $s$.

The descendants of a node $s$ store all (pieces of) line segments contained in $R(s)$.
The lines used to partition the plane are spanned by the line segments.

Every node in the partition tree represents a (piece of a) line segment $s$, a region $R(s)$, and the line $\ell(s)$ spanned by $s$.

The descendants of a node $s$ store all (pieces of) line segments contained in $R(s)$.

**Goal:** Compute a small autopartition for a set $S$ of line segments.
Computing a Small Autopartition

**Autopartition**(*S*)
1. Permute the elements in *S* uniformly at random.
2. Rec-Autopartition(*S*)

**Rec-Autopartition**(*S*)
1. Create a node *r* with *s*(r) = *S*[1]
2. ℓ ← the line spanned by *S*[1]
3. *S*<sub>l</sub> ← all segments in *S* to the left of ℓ
4. *S*<sub>r</sub> ← all segments in *S* to the right of ℓ
5. *S*<sub>i</sub> ← all segments in *S* that intersect ℓ
6. *S*<sub>l</sub> ← *S*<sub>l</sub> ∪ *S*<sub>i</sub>
7. *S*<sub>r</sub> ← *S*<sub>r</sub> ∪ *S*<sub>i</sub>
8. if |*S*<sub>l</sub>| > 0
   9. then left(*r*) ← Rec-Autopartition(*S*<sub>l</sub>)
10. if |*S*<sub>r</sub>| > 0
   11. then right(*r*) ← Rec-Autopartition(*S*<sub>r</sub>)
12. return *r*
**Size of the Computed Partition**

**Observation:** The size of the computed partition is $O(n + I)$, where $I$ is the number of intersections.
**Observation:** The size of the computed partition is $O(n + I)$, where $I$ is the number of intersections.

**Lemma:** The expected number of intersections is $O(n \lg n)$. 
**Observation:** The size of the computed partition is $O(n + I)$, where $I$ is the number of intersections.

**Lemma:** The expected number of intersections is $O(n \lg n)$.

**Corollary:** Algorithm Autopartition computes an autopartition of expected size $O(n \lg n)$.
Generating Uniform Random Permutations

**Random-Permutation** \((A, n)\)

1. for \(i \leftarrow 1\) to \(n - 1\)
2. do swap \(A[i] \leftrightarrow A[\text{random}(i, n)]\)

\[\begin{array}{c}
12345 \\
\hline
32145 \\
34125 \\
34521 \\
34512
\end{array}\]

\(n!\) permutations
Generating Uniform Random Permutations

**Algorithm: Random-Permutation**

```plaintext
RANDOM-PERMUTATION(A, n)
1 for i ← 1 to n − 1
2 do swap A[i] ↔ A[random(i, n)]
```

**Observation:** Algorithm Random-Permutation takes linear time.
Generating Uniform Random Permutations

**Random-Permutation** \((A, n)\)

1. for \(i \leftarrow 1\) to \(n - 1\)
2. do swap \(A[i] \leftrightarrow A[\text{random}(i, n)]\)

**Observation:** Algorithm Random-Permutation takes linear time.

**Lemma:** Algorithm Random-Permutation produces any permutation with probability \(1/n!\).
12345... n

n-1, n-2, n-3

n(n-1)(n-2)(n-3)... n!}

12345

21345
Summary

**Average-case analysis** analyzes the expected running time of deterministic algorithms, assuming a suitable random distribution of the inputs.

**Randomized algorithms** make random choices.

Their expected running time depends on the random choices, not on any input distribution.

**Benefits:**
- Randomized algorithms have no worst-case inputs. (An adversary is powerless.)
- Randomized algorithms are often simpler than equally efficient deterministic algorithms.
- Randomized algorithms are often faster than comparable deterministic algorithms.

**Drawback:**
- In the worst case, a randomized algorithm may be very slow.