Average-Case Analysis and Randomization

Textbook readings:

Chapter 2
Interspersed throughout the book
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
The problem with deterministic Quicksort:

The running time is $O(n \lg n)$, 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.
**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:
   ■ $L = \{x \in A \mid x < p\}$
   ■ $\{p\}$
   ■ $R = \{x \in A \setminus \{p\} \mid x \geq p\}$
5  $L' \leftarrow$ QUICKSORT($L$)
6  $R' \leftarrow$ QUICKSORT($R$)
7  return $L' \circ \{p\} \circ R'$
```
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**($A$)

```plaintext
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 $L' \leftarrow$ QUICKSORT($L$)
6 $R' \leftarrow$ QUICKSORT($R$)
7 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 $|A| \leq 1$
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 $L' \circ \{p\} \circ R'$

Running time:

- Worst case: $O(n^2)$
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 ← A[1]
4   Partition A into three pieces:
5      L = {x ∈ A | x < p}
6      {p}
7      R = {x ∈ A \ {p} | x ≥ p}
8    L′ ← QUICKSORT (L)
9    R′ ← QUICKSORT (R)
10   return L′ ⊕ {p} ⊕ R′
``` 

Running time:

- Worst case: $O(n^2)$
- Average case: $O(n \lg n)$
**Simple-Quicksort**

1. if \(|A| \leq 1\) then return \(A\)
2. else \(p \leftarrow A[1]\)
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. \(L' \leftarrow \text{QUICKSORT}(L)\)
5. \(R' \leftarrow \text{QUICKSORT}(R)\)
6. return \(L' \circ \{p\} \circ R'\)

**Lemma:** The expected running time of algorithm Simple-Quicksort is \(O(n \log n)\).
Observation: The running time of Simple-Quicksort is $O(n + X)$, where $X$ is the number of comparisons it performs.
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 $E(X) = O(n \log n)$.
**Observation:** The running time of Simple-Quicksort is $\mathcal{O}(n + X)$, where $X$ is the number of comparisons it performs.

∴ It suffices to show that $E(X) = \mathcal{O}(n \log n)$.

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

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

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

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

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

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

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

**Lemma:** \( E(X_{ij}) = \Pr(E_{ij}) = \frac{2}{j-i+1} \).
Let $a_1 < a_2 < \cdots < a_n$ be the elements in $A$.

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

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 $O(n \lg n)$ 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 \( \mathcal{O}(n \lg n) \) time?

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

\[
\text{BUCKET-SORT}(A, n)
\]

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. \hspace{0.5cm} do Insert \( A[i] \) into list \( B[\lceil n \cdot A[i] \rceil] \)
4. for \( i \leftarrow 1 \) to \( n \)
5. \hspace{0.5cm} do Sort \( B[i] \) using Insertion Sort
6. Append the sorted list to the output
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 $O(n^2)$. 
Sorting in Linear Time

Can we sort faster than $O(n \log n)$ 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 $O(n^2)$.

**Lemma:** If the elements of $A$ are drawn uniformly at random from $(0, 1]$, the expected running time of Bucket Sort is $O(n)$. 

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**

**RANDOMIZED-QUICKSORT**(A)

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

**Lemma:** The expected running time of algorithm Randomized-Quicksort is \(O(n \lg n)\).
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 \(\mathcal{O}(n)\).
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 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?
Binary Space Partitions

- 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. $\ell \leftarrow$ the line spanned by $S[1]$
3. $S_l \leftarrow$ all segments in $S$ to the left of $\ell$
4. $S_r \leftarrow$ all segments in $S$ to the right of $\ell$
5. $S_i \leftarrow$ all segments in $S$ that intersect $\ell$
6. $S_l \leftarrow S_l \cup S_i$
7. $S_r \leftarrow S_r \cup S_i$
8. **if** $|S_l| > 0$
9. **then** left($r$) $\leftarrow$ Rec-Autopartition($S_l$)
10. **if** $|S_r| > 0$
11. **then** right($r$) $\leftarrow$ Rec-Autopartition($S_r$)
12. **return** $r$
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

\textbf{RANDOM-PERMUTATION}(A, n)

1. for \texttt{i} ← 1 to \texttt{n} − 1
2. do swap \texttt{A[i]} ↔ \texttt{A[random(\texttt{i}, \texttt{n})]}
Generating Uniform Random Permutations

**Algorithm:** Random-Permutation

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.
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 \(1/n!\).
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.