Problem Solving
2-9 Sorting and Selection

Ma Jun

Institute of Computer Software

April 19, 2022
### Contents

1. **Sorting**
   - Quicksort
   - Randomized Quicksort
   - Comparison-based Sort
   - Sorting in Linear Time

2. **Selection**
   - Minimum and Maximum
   - Selection in Expected Linear Time
   - Selection in Worst-case Linear Time
Q: What is the **KEY** idea of Quicksort?
Quicksort

Q: What is the **KEY** idea of Quicksort?

![Diagram showing the key idea of Quicksort with a pivot and two segments of elements, one green and one red, indicating elements less than and greater than the pivot.](image-url)
Q : What is the **KEY** idea of Quicksort?

for any element in this segment, the key is *not greater than pivot.*
Q: What is the **KEY** idea of Quicksort?

For any element in this segment, the key is *not greater than pivot*.

For any element in this segment, the key is *greater than pivot*.
Q : What is the **KEY** idea of Quicksort?

For any element in this segment, the key is *not greater than* pivot.

For any element in this segment, the key is *greater than* pivot.

**To Be Sorted Recursively**
Quicksort

Q: What are the **SIMILARITIES** and **DIFFERENCES** between Quicksort and Mergesort?

**QUICKSORT**\((A, p, r)\)

1. **if** \(p < r\)
2. \(q = \text{PARTITION}(A, p, r)\)
3. **QUICKSORT**\((A, p, q - 1)\)
4. **QUICKSORT**\((A, q + 1, r)\)

**MERGE-SORT**\((A, p, r)\)

1. **if** \(p < r\)
2. \(q = \lfloor(p + r)/2\rfloor\)
3. **MERGE-SORT**\((A, p, q)\)
4. **MERGE-SORT**\((A, q + 1, r)\)
5. **MERGE**\((A, p, q, r)\)
**Quicksort**

**Q**: What are the **SIMILARITIES** and **DIFFERENCES** between Quicksort and Mergesort?

**QUICKSORT** \((A, p, r)\)

1. \textbf{if} \(p < r\)
2. \(q = \text{PARTITION}(A, p, r)\)
3. \textbf{QUICKSORT} \((A, p, q - 1)\)
4. \textbf{QUICKSORT} \((A, q + 1, r)\)

**MERGE-SORT** \((A, p, r)\)

1. \textbf{if} \(p < r\)
2. \(q = \lceil(p + r)/2\rceil\)
3. \textbf{MERGE-SORT} \((A, p, q)\)
4. \textbf{MERGE-SORT} \((A, q + 1, r)\)
5. \textbf{MERGE} \((A, p, q, r)\)

**Similarity**: both are **divide-and-conquer** strategies.
Q: What are the **SIMILARITIES** and **DIFFERENCES** between Quicksort and Mergesort?

**QUICKSORT**

1. if \( p < r \)
2. \( q = \text{PARTITION}(A, p, r) \)
3. \( \text{QUICKSORT}(A, p, q - 1) \)
4. \( \text{QUICKSORT}(A, q + 1, r) \)

**MERGE-SORT**

1. if \( p < r \)
2. \( q = \lfloor (p + r)/2 \rfloor \)
3. \( \text{MERGE-SORT}(A, p, q) \)
4. \( \text{MERGE-SORT}(A, q + 1, r) \)
5. \( \text{MERGE}(A, p, q, r) \)

**Similarity:** both are **divide-and-conquer** strategies.

**Difference:** the process

<table>
<thead>
<tr>
<th></th>
<th>QuickSort</th>
<th>MergeSort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partition</td>
<td>hard</td>
<td>easy</td>
</tr>
<tr>
<td>Combination</td>
<td>easy</td>
<td>hard</td>
</tr>
</tbody>
</table>
QuickSort: **PARTITION**

**Q**: How to prove the correctness of **PARTITION**?

**PARTITION**\((A, p, r)\)

1. \(x = A[r]\)
2. \(i = p - 1\)
3. **for** \(j = p\) **to** \(r - 1\)
4. **if** \(A[j] \leq x\)
5. \(i = i + 1\)
6. **exchange** \(A[i]\) with \(A[j]\)
7. **exchange** \(A[i + 1]\) with \(A[r]\)
8. **return** \(i + 1\)
QuickSort: **PARTITION**

**Q:** How to prove the correctness of **PARTITION**?

**PARTITION**($A$, $p$, $r$)

1. $x = A[r]$
2. $i = p - 1$
3. **for** $j = p$ **to** $r - 1$
4. **if** $A[j] \leq x$
5. \hspace{1em} $i = i + 1$
7. exchange $A[i + 1]$ with $A[r]$
8. **return** $i + 1$
QuickSort: **Partition**

**Q**: How to prove the correctness of **Partition**?

**PARTITION**\((A, p, r)\)

1. \(x = A[r]\)
2. \(i = p - 1\)
3. \(\textbf{for } j = p \textbf{ to } r - 1\)
4. \(\textbf{if } A[j] \leq x\)
5. \(i = i + 1\)
6. exchange \(A[i]\) with \(A[j]\)
7. exchange \(A[i + 1]\) with \(A[r]\)
8. return \(i + 1\)
QuickSort: **PARTITION**

Q: How to prove the correctness of **PARTITION**?

At the beginning of each iteration of the loop of lines 3-6, for any array index $k$, we have:

1. If $p \leq k \leq i$, then $A[k] \leq x$.
2. If $i + 1 \leq k \leq j - 1$, then $A[k] > x$.
3. If $k = r$, then $A[k] = x$.

**PARTITION**($A, p, r$)

1. $x = A[r]$
2. $i = p - 1$
3. **for** $j = p$ **to** $r - 1$
4. **if** $A[j] \leq x$
5. $i = i + 1$
7. **exchange** $A[i + 1]$ with $A[r]$
8. **return** $i + 1$
QuickSort: Time Complexity

Q: What is the time complexity of QuickSort?

QuickSort(A, p, r)
1 if p < r
2 q = Partition(A, p, r)
3 QuickSort(A, p, q - 1)
4 QuickSort(A, q + 1, r)
QuickSort: Time Complexity

Q: What is the time complexity of QUICKSORT?

```
QUICKSORT(A, p, r)
1   if p < r
2       q = PARTITION(A, p, r)
3   QUICKSORT(A, p, q - 1)
4   QUICKSORT(A, q + 1, r)
```

The recurrence: \( T(n) = T(n_1) + T(n_2) + cn \)

where:
Quicksort: Time Complexity

**Q**: What is the time complexity of **QUICKSORT**?

```
QUICKSORT(A, p, r)
1  if p < r
2    q = PARTITION(A, p, r)
3  QUICKSORT(A, p, q - 1)
4  QUICKSORT(A, q + 1, r)
```

The recurrence: \( T(n) = T(n_1) + T(n_2) + cn \)

where:

\[
\begin{align*}
    n_1 &= q - 1 - p + 1 = q - p \\
    n_2 &= r - (q + 1) + 1 = r - q \\
    n_1 + n_2 &= r - p \\
\end{align*}
\]

*initially*, \( p = 1, r = n \)
QuickSort: Time Complexity

Q: What is the time complexity of QuickSort?

```plaintext
QuickSort(A, p, r)
1  if p < r
2     q = Partition(A, p, r)
3     QuickSort(A, p, q - 1)
4     QuickSort(A, q + 1, r)
```

The recurrence: $T(n) = T(n_1) + T(n_2) + cn$

where:

$n_1 = q - 1 - p + 1 = q - p$
$n_2 = r - (q + 1) + 1 = r - q$
$n_1 + n_2 = r - p$

initially, $p = 1, r = n$

$n_1, n_2$ vary and depend on $q = Partition(A, p, r)$
Quicksort: Time Complexity

Q: Which factor would affect the efficiency of Quicksort?

always produces a 9-to-1 split

\[
\begin{align*}
\log_{10} n & \\
\log_{10/9} n & \\
1 & \\
\text{always produces a 9-to-1 split} & \\
O(n \log n) & \\
\end{align*}
\]
Q: Which factor would affect the efficiency of Quicksort?

always produces a 9-to-1 split

the choice of Pivot would affect the tree height.

$O(n \log n)$
**Quicksort: Time Complexity**

**Q**: Which factor would affect the efficiency of **QUICKSORT**?

- always produces a 9-to-1 split
- any split of **constant proportionality**
  - tree height: \( \Theta(\lg n) \)
  - cost of each level: \( cn \)
  - total running time: \( O(n \lg n) \)

\[
\begin{align*}
&n \\
&\frac{1}{10}n \\
&\frac{9}{100}n \\
&\frac{9}{10}n \\
&\frac{81}{1000}n \\
&\frac{729}{10000}n \\
&\frac{1}{1000}n \\
&\frac{1}{100}n \\
&\log_{10} n \\
&\log_{10/9} n
\end{align*}
\]
QuickSort: Time Complexity

Q: Which factor would affect the efficiency of QuickSort?

What is the WORST CASE?

any split of constant proportionality

What is the WORST CASE?

\[ T(n) = \max_{0 \leq q \leq n-1} (T(q) + T(n-q-1)) + \Theta(n) \]
Q: Which factor would affect the efficiency of QuickSort?

Any split of constant proportionality.

What is the WORST CASE?

\[ T(n) = \max_{0 \leq q \leq n-1} (T(q) + T(n - q - 1)) + \Theta(n) \]

\[ O(n \log n) \]
Quicksort: Time Complexity

Worst Case:

$$T(n) = \max_{0 \leq q \leq n-1} (T(q) + T(n - q - 1)) + \Theta(n)$$

Q : When would the worst case happen?
Quicksort: Time Complexity

Worst Case:

\[ T(n) = \max_{0 \leq q \leq n-1} (T(q) + T(n - q - 1)) + \Theta(n) \]

Q : When would the worst case happen?

The pivot is always the greatest or smallest element for each recursion.
Quicksort: Time Complexity

Worst Case:

\[ T(n) = \max_{0 \leq q \leq n-1} (T(q) + T(n - q - 1)) + \Theta(n) \]

Q : When would the worst case happen?

The pivot is always the greatest or smallest element for each recursion.

Unlucky: \( T(n) = O(n^2) \) for the worst case!
Quicksort: Time Complexity

Worst Case:

\[ T(n) = \max_{0 \leq q \leq n-1}(T(q) + T(n - q - 1)) + \Theta(n) \]

Q : When would the worst case happen?

The pivot is always the greatest or smallest element for each recursion.

Unlucky: \( T(n) = O(n^2) \) for the worst case!

Lucky: worst case seldom happens!
Quicksort: Time Complexity

Impression & Intuition:

Quick sort performs quite well in practice.
Quick sort performs quite well in practice.

We usually obtain an $O(n \lg n)$ execution in most cases, rather than the worst case.
Quick sort performs quite well in practice.

We usually obtain an $O(n \lg n)$ execution in most cases, rather than the worst case.

WHY?
**Impression & Intuition:**

Quick sort performs quite well in practice.

We usually obtain an $O(n \lg n)$ execution in most cases, rather than the worst case.

**WHY?**

Partition produces a mix of “good” and “bad” splits.
Impression & Intuition:

Quick sort performs quite well in practice.

We usually obtain an $O(n \lg n)$ execution in most cases, rather than the worst case.

**WHY?**

**Partition** produces a mix of “good” and “bad” splits.

$$T(n) = O(n \lg n)$$
QuickSort: Time Complexity

Critical operation?

- The key cost of QuickSort comes from **Partition**
- The key cost of **Partition** comes from line 4.

**QuickSort**($A, p, r$)

1. \textbf{if} $p < r$
2. \texttt{q = Partition(A, p, r)}
3. \texttt{QuickSort(A, p, q - 1)}
4. \texttt{QuickSort(A, q + 1, r)}

**Partition**($A, p, r$)

1. \texttt{x = A[r]}
2. \texttt{i = p - 1}
3. \texttt{for} $j = p \textbf{ to } r - 1$
4. \texttt{if} $A[j] < x$
5. \hspace{1em} \texttt{i = i + 1}
6. \hspace{1em} \texttt{exchange A[i] with A[j]}
7. \texttt{exchange A[i + 1] with A[r]}
8. \textbf{return} $i + 1$
Lemma (7.1)

Let $X$ be the number of comparisons performed in line 4 of \texttt{Partition} over the entire execution of \texttt{Quicksort} on an $n$-element array. Then the running time of \texttt{Quicksort} is $O(n + X)$. 
Lemma (7.1)

Let \( X \) be the number of comparisons performed in line 4 of \texttt{Partition} over the entire execution of \texttt{Quicksort} on an \( n \)-element array. Then the running time of \texttt{Quicksort} is \( O(n + X) \).

Proof.

By the discussion above, the algorithm makes at most \( n \) calls to \texttt{Partition}, each of which does a constant amount of work and then executes the \texttt{for loop} some number of times. Each iteration of the \texttt{for loop} executes line 4.
**Randomized Quicksort**

**Randomized Quicksort**

```plaintext
RANDOMIZED-QUICKSORT(A, p, r)
1   if p < r
2       q = RANDOMIZED-PARTITION(A, p, r)
3   RANDOMIZED-QUICKSORT(A, p, q − 1)
4   RANDOMIZED-QUICKSORT(A, q + 1, r)
```

**Goal:**

To compute \( X \), the **TOTAL** number of comparisons performed in all calls to \( \text{PARTITION} \).

We will **NOT** attempt to analyze how many comparisons are made in **EACH** call to \( \text{PARTITION} \).
Randomized Quicksort: Expected Running Time

Q : How to compute the expected value of $X$?

$X$: the **TOTAL** number of comparisons performed in all calls to **Partition**.
Randomized Quicksort: Expected Running Time

**Q :** How to compute the expected value of $X$?

$X$: the **TOTAL** number of comparisons performed in all calls to **Partition**.

- We must understand *when the algorithm compares two elements of the array and when it does not.*
Randomized Quicksort: Expected Running Time

Q : How to compute the expected value of X?

\( X \): the **TOTAL** number of comparisons performed in all calls to **Partition**.

- We must understand *when the algorithm compares two elements of the array and when it does not*.
- For ease of analysis, we rename the elements of the array \( A \) as \( \{z_1, z_2, \ldots, z_n\} \), with \( z_i \) being the \( i \)th smallest element.
Randomized Quicksort: Expected Running Time

**Q:** How to compute the expected value of $X$?

$X$: the **TOTAL** number of comparisons performed in all calls to **Partition**.

- We must understand **when the algorithm compares two elements of the array and when it does not**.
- For ease of analysis, we rename the elements of the array $A$ as $\{z_1, z_2, \ldots, z_n\}$, with $z_i$ being the $i$th smallest element.
- $Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$: the set of elements between $z_i$ and $z_j$, inclusive.
Randomized Quicksort: Expected Running Time

Q: When does the algorithm compare $z_i$ and $z_j$?

- Each pair of elements is compared at most once.
- Elements are compared only to the pivot element.
- After a particular call of `PARTITION` finishes, the pivot element used in that call is never again compared to any other elements.
Randomized Quicksort: Expected Running Time

Q: When does the algorithm compare \( z_i \) and \( z_j \)?

- Each pair of elements is compared at most once
- Elements are compared only to the pivot element
- After a particular call of **Partition** finishes, the pivot element used in that call is never again compared to any other elements.

\( X_{ij} \): indicator random variables

\[
X_{ij} = I\{z_i \text{ is compared to } z_j\}
\]
Randomized Quicksort: Expected Running Time

**Q** : When does the algorithm compare \( z_i \) and \( z_j \)?

- Each pair of elements is compared **at most once**
- Elements are compared **only to the pivot element**
- After a particular call of \textsc{Partition} finishes, **the pivot element** used in that call is **never again** compared to any other elements.

\( X_{ij} \): indicator random variables

\[
X_{ij} = I\{z_i \text{ is compared to } z_j\}
\]

Then, we have:

\[
X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}
\]
Randomized Quicksort: Expected Running Time

**Q**: How to compute the expected value of $X$?

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

\[
E[X] = E \left[ \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij} \right] \\
= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}] \\
= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} Pr\{z_i \text{ is compared to } z_j\} \\
= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \\
< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} \\
= \sum_{i=1}^{n-1} O(\lg n) \\
= O(n \lg n)
\]
Randomized Quicksort: Expected Running Time

Q: What is $\Pr\{z_i \text{ is compared to } z_j\}$?

\[
\Pr\{z_i \text{ is compared to } z_j\} = \Pr\{z_i \text{ or } z_j \text{ is first pivot chosen from } Z_{ij}\} \\
= \Pr\{z_i \text{ is first pivot chosen from } Z_{ij}\} \\
+ \Pr\{z_j \text{ is first pivot chosen from } Z_{ij}\} \\
= \frac{1}{j-i+1} + \frac{1}{j-i+1} \\
= \frac{2}{j-i+1}.
\]
Q: What is $Pr\{z_i \text{ is compared to } z_j\}$?

\[
Pr\{z_i \text{ is compared to } z_j\} = Pr\{z_i \text{ or } z_j \text{ is first pivot chosen from } Z_{ij}\}
\]
\[
= Pr\{z_i \text{ is first pivot chosen from } Z_{ij}\} + Pr\{z_j \text{ is first pivot chosen from } Z_{ij}\}
\]
\[
= \frac{1}{j - i + 1} + \frac{1}{j - i + 1}
\]
\[
= \frac{2}{j - i + 1}.
\]
Top 10 Algorithms

The 10 Algorithms with the Greatest Influence on the Development and Practice of Science and Engineering in the 20th Century

- Metropolis Algorithm for Monte Carlo
- Simplex Method for Linear Programming
- Krylov Subspace Iteration Methods
- The Decompositional Approach to Matrix Computations
- The Fortran Optimizing Compiler
- QR Algorithm for Computing Eigenvalues
- **Quicksort Algorithm for Sorting**
- Fast Fourier Transform
- Integer Relation Detection
- Fast Multipole Method

Comparison-based Sort Algorithm

Theorem (8.1)

Any comparison sort algorithm requires $\Omega(n \lg n)$ comparisons in the worst case.

- $n!$ reachable leaves, each of which corresponds to a possible permutation
- $h$: the height of the decision (binary) tree
- $n! \leq 2^h \implies h \geq \lg n! = \Omega(n \lg n)$
Sorting in Linear Time

- Counting Sort
- Radix Sort
- Bucket Sort
Sorting in Linear Time: Counting Sort

Assumption

Each of the input elements is an integer in the range 0 to $k$.

$T(n) = \Theta(n + k)$, and if $k = O(n)$, $T(n) = \Theta(n)$.

```
COUNTING-SORT(A, B, k)
1   let C[0..k] be a new array
2   for $i$ = 0 to $k$
3       $C[i] = 0$
4   for $j$ = 1 to A.length
5       $C[A[j]] = C[A[j]] + 1$
6   // $C[i]$ now contains the number of elements equal to $i$.
7   for $i$ = 1 to $k$
8       $C[i] = C[i] + C[i - 1]$
9   // $C[i]$ now contains the number of elements less than or equal to $i$.
10  for $j$ = A.length downto 1
12     $C[A[j]] = C[A[j]] - 1$
```
Sorting in Linear Time: Counting Sort

Assumption

Each of the input elements is an integer in the range 0 to k.

\[ T(n) = \Theta(n + k), \text{ and if } k = O(n), \ T(n) = \Theta(n). \]

```
COUNTING-SORT(A, B, k)
1    let C[0..k] be a new array
2    for i = 0 to k
3        C[i] = 0
4    for j = 1 to A.length
5        C[A[j]] = C[A[j]] + 1
6    // C[i] now contains the number of elements equal to i.
7    for i = 1 to k
8        C[i] = C[i] + C[i - 1]
9    // C[i] now contains the number of elements less than or equal to i.
10   for j = A.length downto 1
12      C[A[j]] = C[A[j]] - 1
```

(a) A = [2 5 3 0 2 3 0 3]
    C = [0 1 2 3 4 5]

(b) A = [2 0 2 3 0 1]
    C = [2 2 4 7 7 8]
Assumption

Each of the input elements is an integer in the range 0 to \( k \).

\[ T(n) = \Theta(n + k), \text{ and if } k = O(n), T(n) = \Theta(n). \]

```
COUNTING-SORT(A, B, k)
1  let C[0..k] be a new array
2  for i = 0 to k
3      C[i] = 0
4  for j = 1 to A.length
5      C[A[j]] = C[A[j]] + 1
6     // C[i] now contains the number of elements equal to i.
7  for i = 1 to k
8      C[i] = C[i] + C[i - 1]
9     // C[i] now contains the number of elements less than or equal to i.
10     for j = A.length downto 1
12        C[A[j]] = C[A[j]] - 1
```
Sorting in Linear Time: Counting Sort

Assumption

Each of the input elements is an integer in the range 0 to \( k \).

\[
T(n) = \Theta(n + k), \text{ and if } k = O(n), \ T(n) = \Theta(n)
\]

```plaintext
COUNTING-SORT(A, B, k)
1 let C[0..k] be a new array
2 for i = 0 to k
3 \( C[i] = 0 \)
4 for j = 1 to A.length
5 \( C[A[j]] = C[A[j]] + 1 \)
6 // \( C[i] \) now contains the number of elements equal to \( i \).
7 for i = 1 to k
8 \( C[i] = C[i] + C[i - 1] \)
9 // \( C[i] \) now contains the number of elements less than or equal to \( i \).
10 for j = A.length downto 1
12 \( C[A[j]] = C[A[j]] - 1 \)
```

(a)  

\[
A = \begin{bmatrix} 2 & 5 & 3 & 0 & 2 & 3 & 0 & 3 \\ 0 & 1 & 2 & 3 & 4 & 5 \\ 2 & 0 & 2 & 3 & 0 & 1 \end{bmatrix}
\]

(b)  

\[
C = \begin{bmatrix} 2 & 2 & 4 & 7 & 7 & 8 \\
0 & 1 & 2 & 3 & 4 & 5 \\
2 & 2 & 4 & 7 & 7 & 8 \end{bmatrix}
\]
Assumption

Each of the input elements is an integer in the range 0 to \( k \).

\[ T(n) = \Theta(n + k), \text{ and if } k = O(n), \ T(n) = \Theta(n). \]
Sorting in Linear Time: Counting Sort

```plaintext
10 for j = A.length downto 1
12 \[ C[A[j]] = C[A[j]] - 1 \]
```

(a)

(b)

(c)

(d)

(e)
Sorting in Linear Time: Radix Sort

Assumption

- Each element in the $n$-element array $A$ has $d$ digits, where digit 1 is the lowest-order digit and digit $d$ is the highest-order digit.
- Each digit can take on up to $k$ possible values

```
RADIX-SORT(A, d)
1   for i = 1 to d
2       use a stable sort to sort array A on digit i
```
Assumption

- Each element in the $n$-element array $A$ has $d$ digits, where digit 1 is the lowest-order digit and digit $d$ is the highest-order digit.
- Each digit can take on up to $k$ possible values.

**Radix-Sort**

```plaintext
RADIX-SORT($A$, $d$)
1  for $i = 1$ to $d$
2      use a stable sort to sort array $A$ on digit $i$
```
Lemma (8.3)

Given $n$ $d$-digit numbers in which each digit can take on up to $k$ possible values, \textsc{Radix-Sort} correctly sorts these numbers in $\Theta(d(n + k))$ time if the \textit{stable sort} it uses takes $\Theta(n + k)$ time.
Sorting in Linear Time: Radix Sort

Lemma (8.3)

Given $n$ $d$-digit numbers in which each digit can take on up to $k$ possible values, Radix-Sort correctly sorts these numbers in $\Theta(d(n + k))$ time if the stable sort it uses takes $\Theta(n + k)$ time.

Lemma (8.4)

Given $n$ $b$-bit numbers and any positive integer $r \leq b$, Radix-Sort correctly sorts these numbers in $\Theta((b/r)(n + 2^r))$ time if the stable sort it uses takes $\Theta(n + k)$ time for inputs in the range 0 to $k$.

Proof  For a value $r \leq b$, we view each key as having $d = \lceil b/r \rceil$ digits of $r$ bits each. Each digit is an integer in the range 0 to $2^r - 1$, so that we can use counting sort with $k = 2^r - 1$. (For example, we can view a 32-bit word as having four 8-bit digits, so that $b = 32$, $r = 8$, $k = 2^r - 1 = 255$, and $d = b/r = 4$.) Each pass of counting sort takes time $\Theta(n + k) = \Theta(n + 2^r)$ and there are $d$ passes, for a total running time of $\Theta(d(n + 2^r)) = \Theta((b/r)(n + 2^r))$. 

Ma Jun (Institute of Computer Software)
Assumption

The input is drawn from a **uniform distribution**

**BUCKET-SORT(A)**

1. let $B[0..n-1]$ be a new array
2. $n = A\.length$
3. for $i = 0$ to $n - 1$
   4. make $B[i]$ an empty list
5. for $i = 1$ to $n$
   6. insert $A[i]$ into list $B[[nA[i]]]$
7. for $i = 0$ to $n - 1$
   8. sort list $B[i]$ with **insertion sort**
9. concatenate the lists $B[0], B[1], \ldots, B[n - 1]$ together in order
Sorting in Linear Time: Bucket Sort

**Algorithm 3.1** Bucket-Sort($A$)

1. let $B[0..n-1]$ be a new array
2. $n = A.length$
3. for $i = 0$ to $n-1$
4. make $B[i]$ an empty list
5. for $i = 1$ to $n$
6. insert $A[i]$ into list $B[[nA[i]]]$
7. for $i = 0$ to $n-1$
8. sort list $B[i]$ with insertion sort
9. concatenate the lists $B[0], B[1], \ldots, B[n-1]$ together in order

- All lines except line 8 take $O(n)$ time in the worst case.
- $n_i$: the number of elements placed in bucket $B[i]$. 

**Note:**

- $T(n) = \Theta(n) + n - 1 \sum_{i=0}^{n-1} O(n^2 i)$
- All lines except line 8 take $O(n)$ time in the worst case.
- $n_i$: the number of elements placed in bucket $B[i]$. 

**Problem Solving**

Ma Jun (Institute of Computer Software)

April 19, 2022 25/42
Sorting in Linear Time: Bucket Sort

**BUCKET-SORT**(A)
1. let B[0..n-1] be a new array
2. \(n = A\.length\)
3. for \(i = 0\) to \(n-1\)
   4. make \(B[i]\) an empty list
5. for \(i = 1\) to \(n\)
   6. insert \(A[i]\) into list \(B[[nA[i]]]\)
7. for \(i = 0\) to \(n-1\)
   8. sort list \(B[i]\) with **insertion sort** \(O(n_i^2)\)
9. concatenate the lists \(B[0], B[1], \ldots, B[n-1]\) together in order

- All lines except line 8 take \(O(n)\) time in the worst case.
- \(n_i\): the number of elements placed in bucket \(B[i]\).
Sorting in Linear Time: Bucket Sort

**BUCKET-SORT(A)**
1. let \( B[0..n-1] \) be a new array
2. \( n = A.length \)
3. for \( i = 0 \) to \( n-1 \)
4. make \( B[i] \) an empty list
t5. for \( i = 1 \) to \( n \)
6. insert \( A[i] \) into list \( B[[nA[i]]] \)
7. for \( i = 0 \) to \( n-1 \)
8. sort list \( B[i] \) with *insertion sort* \( O(n_i^2) \)
9. concatenate the lists \( B[0], B[1], \ldots, B[n-1] \) together in order

- All lines except line 8 take \( O(n) \) time in the worst case.
- \( n_i \): the number of elements placed in bucket \( B[i] \).
Sorting in Linear Time: Bucket Sort

\[ E[T(n)] = E\left[ \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2) \right] \]

\[ = \Theta(n) + \sum_{i=0}^{n-1} E \left[ O(n_i^2) \right] \]

\[ = \Theta(n) + \sum_{i=0}^{n-1} O(E[n_i^2]) \]

\[ = \Theta(n) \]
Sorting in Linear Time: Bucket Sort

\[ E \left[ T(n) \right] = E \left[ \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2) \right] \]

\[ = \Theta(n) + \sum_{i=0}^{n-1} E \left[ O(n_i^2) \right] \]

\[ = \Theta(n) + \sum_{i=0}^{n-1} O\left( E \left[ n_i^2 \right] \right) \]

\[ = \Theta(n) \]

\[ X_{ij} = I\{A[j] \text{ falls in bucket } i\} \]

for \( i = 0, 1, \ldots, n - 1 \) and \( j = 1, 2, \ldots, n \). Thus,

\[ n_i = \sum_{j=1}^{n} X_{ij} . \]

\[ \sum_{i=0}^{n-1} O\left( E \left[ n_i^2 \right] \right) = 2 - \frac{1}{n} \]
## Complexity of Sorting Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Worst-case running time</th>
<th>Average-case/expected running time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion sort</td>
<td>$Θ(n^2)$</td>
<td>$Θ(n^2)$</td>
</tr>
<tr>
<td>Merge sort</td>
<td>$Θ(n \lg n)$</td>
<td>$Θ(n \lg n)$</td>
</tr>
<tr>
<td>Heapsort</td>
<td>$O(n \lg n)$</td>
<td>—</td>
</tr>
<tr>
<td>Quicksort</td>
<td>$Θ(n^2)$</td>
<td>$Θ(n \lg n)$ (expected)</td>
</tr>
<tr>
<td>Counting sort</td>
<td>$Θ(k + n)$</td>
<td>$Θ(k + n)$</td>
</tr>
<tr>
<td>Radix sort</td>
<td>$Θ(d(n + k))$</td>
<td>$Θ(d(n + k))$</td>
</tr>
<tr>
<td>Bucket sort</td>
<td>$Θ(n^2)$</td>
<td>$Θ(n)$ (average-case)</td>
</tr>
</tbody>
</table>
Contents

1 Sorting
   - Quicksort
   - Randomized Quicksort
   - Comparison-based Sort
   - Sorting in Linear Time

2 Selection
   - Minimum and Maximum
   - Selection in Expected Linear Time
   - Selection in Worst-case Linear Time
Minimum and Maximum

Problem (Minimum or Maximum)

Given a subset of a total-order set, find the maximum or minimum element of the subset.
Problem (Minimum or Maximum)

*Given a subset of a total-order set, find the maximum or minimum element of the subset.*

- requires **at least** \( n - 1 \) comparisons

```python
def MINIMUM(A):
    1. \( min = A[1] \)
    2. for \( i = 2 \) to \( A.length \):
        3. if \( min > A[i] \):
            4. \( min = A[i] \)
    5. return \( min \)
```

Minimum and Maximum
Problem (Maximum & minimum)

Given a subset of a total-order set, find both the maximum and minimum elements of the subset.
Minimum and Maximum

Problem (Maximum & minimum)

Given a subset of a total-order set, find both the maximum and minimum elements of the subset.

- does not require $2n - 2$ comparisons
Problem (Maximum & minimum)

Given a subset of a total-order set, find both the maximum and minimum elements of the subset.
- does not require $2n - 2$ comparisons

A possible way for finding both maximum & minimum.
- compare pairs of elements from the input first with each other
- then compare the smaller with the current minimum and the larger to the current maximum
- at most $3\lceil n/2 \rceil$ comparisons
General Selection Problem

Problem (General Selection)

*Given a subset of a total-order set, find the $i$-th smallest element of the subset.*
Selection in Expected Linear Time:

**RANDOMIZED-SELECT**

\[
\text{RANDOMIZED-SELECT}(A, p, r, i) \\
1. \textbf{if } p == r \\
2. \quad \text{return } A[p] \\
3. q = \text{RANDOMIZED-PARTITION}(A, p, r) \\
4. k = q - p + 1 \\
5. \textbf{if } i == k \quad \text{// the pivot value is the answer} \\
6. \quad \text{return } A[q] \\
7. \textbf{elseif } i < k \\
8. \quad \text{return } \text{RANDOMIZED-SELECT}(A, p, q - 1, i) \\
9. \textbf{else} \text{return } \text{RANDOMIZED-SELECT}(A, q + 1, r, i - k)
\]
# Selection in Expected Linear Time: \textsc{Randomized-Select}

\begin{algorithm}
\textbf{Randomized-Select}(A, p, r, i)
\begin{algorithmic}[1]
\State \textbf{if} \( p == r \)
\State \hspace{1em} \textbf{return} \( A[p] \)
\State \( q = \text{Randomized-Partition}(A, p, r) \)
\State \( k = q - p + 1 \)
\State \textbf{if} \( i == k \) \quad // \text{the pivot value is the answer}
\State \hspace{1em} \textbf{return} \( A[q] \)
\State \textbf{elseif} \( i < k \)
\State \hspace{1em} \textbf{return} \text{Randomized-Select}(A, p, q - 1, i)
\State \textbf{else} \textbf{return} \text{Randomized-Select}(A, q + 1, r, i - k)
\end{algorithmic}
\end{algorithm}

Similar to \textsc{Randomized-Quicksort}, but only have to handle exact one sub-problem in each step of the recursion.
**RANDOMIZED-SELECT: Expected Running Time**

**Q**: What is the expected running time of **RANDOMIZED-SELECT**?
**Randomized-Select: Expected Running Time**

**Q : What is the expected running time of Randomized-Select?**

Indicator random variable $X_k$:

- $X_k = I\{\text{the subarray } A[p..q] \text{ has exactly } k \text{ elements}\}$
- Assuming the elements are distinct, we have $E[X_k] = 1/n$

---

```plaintext
Randomized-Select(A, p, r, i)
1 if p == r
2    return A[p]
3 q = Randomized-Partition(A, p, r)
4 k = q - p + 1
5 if i == k              // the pivot value is the answer
6    return A[q]
7 elseif i < k
8    return Randomized-Select(A, p, q - 1, i)
9 else return Randomized-Select(A, q + 1, r, i - k)
```
**Randomized-Select**: Expected Running Time

**Q**: What is the expected running time of **Randomized-Select**?

indicator random variable $X_k$:

- $X_k = I\{\text{the subarray } A[p..q] \text{ has exactly } k \text{ elements}\}$
- assuming the elements are distinct, we have $E[X_k] = 1/n$

$t(n)$: the running time on an input array of size $n$

$$T(n) \leq \sum_{k=1}^{n} X_k \cdot (T(\max(k-1, n-k)) + O(n))$$

$$= \sum_{k=1}^{n} X_k \cdot T(\max(k-1, n-k)) + O(n).$$

```python
RANDOMIZED-SELECT(A, p, r, i)
1    if p == r
2        return A[p]
3    q = RANDOMIZED-PARTITION(A, p, r)
4    k = q - p + 1
5    if i == k        // the pivot value is the answer
6        return A[q]
7    elseif i < k
8        return RANDOMIZED-SELECT(A, p, q - 1, i)
9    else return RANDOMIZED-SELECT(A, q + 1, r, i - k)
```
**RANDOMIZED-SELECT:** Expected Running Time

**Q:** What is the expected running time of **RANDOMIZED-SELECT**?

Indicator random variable $X_k$:
- $X_k = I\{\text{the subarray } A[p..q] \text{ has exactly } k \text{ elements}\}$
- assuming the elements are distinct, we have $E[X_k] = 1/n$

$T(n)$: the running time on an input array of size $n$

$$T(n) \leq \sum_{k=1}^{n} X_k \cdot (T(\max(k-1, n-k)) + O(n))$$

$$= \sum_{k=1}^{n} X_k \cdot T(\max(k-1, n-k)) + O(n).$$
**Randomized-Select: Expected Running Time**

\[ E[T(n)]: \text{the expected running time on an input array of size } n \]

\[
E[T(n)] \\
\leq E \left[ \sum_{k=1}^{n} X_k \cdot T(\max(k-1, n-k)) + O(n) \right] \\
= \sum_{k=1}^{n} E[X_k] \cdot E[T(\max(k-1, n-k))] + O(n) \quad \text{(by linearity of expectation)} \\
= \sum_{k=1}^{n} \frac{1}{n} \cdot E[T(\max(k-1, n-k))] + O(n) \quad \text{(by equation (9.1))}.
\]
**Randomized-Select:** Expected Running Time

$E[T(n)]$: the expected running time on an input array of size $n$

Let us consider the expression $\max(k - 1, n - k)$. We have

$$\max(k - 1, n - k) = \begin{cases} k - 1 & \text{if } k > \lfloor n/2 \rfloor, \\ n - k & \text{if } k \leq \lfloor n/2 \rfloor. \end{cases}$$

If $n$ is even, each term from $T(\lfloor n/2 \rfloor)$ up to $T(n - 1)$ appears exactly twice in the summation, and if $n$ is odd, all these terms appear twice and $T(\lfloor n/2 \rfloor)$ appears once. Thus, we have

$$E[T(n)] \leq \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} E[T(k)] + O(n).$$
**Randomized-Select: Expected Running Time**

\[ E[T(n)]: \text{the expected running time on an input array of size } n \]

Let us consider the expression \( \max(k - 1, n - k) \). We have

\[
\max(k - 1, n - k) = \begin{cases} 
  k - 1 & \text{if } k > \lfloor n/2 \rfloor, \\
  n - k & \text{if } k \leq \lfloor n/2 \rfloor.
\end{cases}
\]

If \( n \) is even, each term from \( T(\lfloor n/2 \rfloor) \) up to \( T(n - 1) \) appears exactly twice in the summation, and if \( n \) is odd, all these terms appear twice and \( T(\lfloor n/2 \rfloor) \) appears once. Thus, we have

\[
E[T(n)] \leq \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} E[T(k)] + O(n).
\]

Then, we could prove \( E[T(n)] = O(n) \) by substitution. Assuming:

\[
E[T(n)] \leq cn
\]
Selection in Expected Linear Time: **SELECT**

**SELECT**

1. Divide the input array into \([n/5]\) groups of 5 elements each
   - at most one group made up of the remaining \(n \mod 5\) elements.
2. Find the **median** of each of the \([n/5]\) groups with **insertion-sort**.
3. Use **SELECT** recursively to find the **median** \(m^*\) of the medians found in step 2.
4. **Partition** the input array around the **median-of-medians** \(m^*\).
5. Assume that \(m^*\) is the \(k\)th smallest element. If \(i = k\), then return \(m^*\). Otherwise, use **SELECT** recursively:
   - if \(i < k\), find the \(i\)th smallest element on the low side
   - if \(i > k\), find the \((i - k)\)th smallest element on the high side
Step 1: Divide the input array into $\lceil n/5 \rceil$ groups of 5 elements each
Step 2: Find the **median** of each of the \(\lceil n/5 \rceil\) groups with INSERTION-SORT.
**Select**

Step 3: Use `SELECT` recursively to find the **median** $m^*$ of the medians found in step 2.
Step 3: Use $\textit{SELECT}$ recursively to find the median $m^*$ of the medians found in step 2.
Step 3: Use \texttt{SELECT} recursively to find the median $m^*$ of the medians found in step 2.
Step 4: Partition the input array around $m^*$. 

Increasing by medians

Increasing

Median of medians

Elements in $A$ and $D$ are unknown.
Step 4: **Partition** the input array around $m^*$. 

The elements in the array are divided into four partitions: 

- **A** (purple): Elements less than $m^*$.
- **B** (red): Elements greater than $m^*$.
- **C** (orange): Elements less than $m^*$.
- **D** (green): Elements greater than $m^*$.

The median of medians is used to decide the partition point $m^*$. 

- Elements greater than $m^*$ or less than $m^*$ are only unknown for elements in partitions **A** and **D**.

The diagram shows the partition process with arrows indicating the direction and order of elements increasing by medians.
**Select**

Step 5: Assume that $m^*$ is the $k$th smallest element.

- If $i = k$, then return $m^*$.
- Otherwise, use `Select` recursively:
  - if $i < k$, find the $i$th smallest element on the low side
  - if $i > k$, find the $(i-k)$th smallest element on the high side

---

**Diagram:**

- **A** and **B** represent decreasing sequences.
- **C** and **D** represent increasing sequences.
- **m** is the median of medians.
- The sequences are increasing by medians.
**Select**

Step 5: Assume that $m^*$ is the $k$th smallest element.

- If $i = k$, then return $m^*$.
- Otherwise, use Select recursively:
  - if $i < k$, find the $i$th smallest element on the low side
  - if $i > k$, find the $(i - k)$th smallest element on the high side

![Diagram showing four regions labeled A, B, C, and D. Region A contains elements less than $m^*$, region B contains elements greater than $m^*$, region C contains elements less than $m^*$, and region D contains elements greater than $m^*$, with a middle horizontal line dividing B into two regions. The diagram also shows an arrow labeled "increasing by medians." The text $|C| \geq 3n/10 - 6$ and $|B| \geq 3n/10 - 6$ is present.]
**SELECT**

Step 5: Assume that $m^*$ is the $k$th smallest element.

- If $i = k$, then return $m^*$.
- Otherwise, use SELECT recursively:
  - if $i < k$, find the $i$th smallest element on the low side
  - if $i > k$, find the $(i - k)$th smallest element on the high side

---

Problem Solving  
April 19, 2022  40 / 42
The **SELECT** algorithm: Running Time in Worst-case

Counting the total number of comparisons

\[ T(n) \leq T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n) \]

- \( T(\lceil n/5 \rceil) \): find the median of the medians via calling **SELECT**
- \( T(7n/10 + 6) \): maximum cost for calling **SELECT** recursively.
- \( O(n) \):
  - divide the input array into 5-elements groups
  - find medians of all 5-elements groups, about \( 6 \times \lceil n/5 \rceil \)
  - **PARTITION** with the pivot \( m^* \)

We could show that the running time \( T(n) = O(n) \) by substitution
Thank You!

Questions?

Office 541
majun@nju.edu.cn