Problem Solving
2-9 Sorting and Selection

MA Jun

Institute of Computer Software

April 23, 2020
Contents

1. Sorting

2. Selection
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
Quick sort

**Question**: What is the **KEY** idea of Quick sort?
**Question**: What is the **KEY** idea of Quicksort?
Question: What is the **KEY** idea of Quicksort?

For any element in this segment, the key is **not** greater than pivot.
Question: 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.
**Question**: 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
Question: What are the **SIMILARITIES** and **DIFFERENCES** between Quicksort and Mergesort?

**Quicksort**

\[
\text{QUICKSORT}(A, p, r)
\]

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

**Mergesort**

\[
\text{MERGE-SORT}(A, p, r)
\]

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. \( \text{MERGE}(A, p, q, r) \)

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

**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)
```

**Mergesort**

```plaintext
MERGE-SORT(A, p, r)
1   if p < r
2     q = ⌊(p + r)/2⌋
3     MERGE-SORT(A, p, q)
4     MERGE-SORT(A, q + 1, r)
5     MERGE(A, p, q, r)
```

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

**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><strong>Partition</strong></td>
<td>hard</td>
<td>easy</td>
</tr>
<tr>
<td><strong>Combination</strong></td>
<td>easy</td>
<td>hard</td>
</tr>
</tbody>
</table>
Quicksort: **PARTITION**

**Question**: 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. $i = i + 1$

7. exchange $A[i + 1]$ with $A[r]$  
8. return $i + 1$
QuickSort: **PARTITION**

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

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

1. \(x = A[r]\)
2. \(i = p - 1\)
3. \(\text{for } j = p \text{ to } r - 1\)
4. \(\text{if } A[j] \leq x\)
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

Question: 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. $i = i + 1$
7. exchange $A[i + 1]$ with $A[r]$
8. return $i + 1$
QuickSort: Partition

**Question**: 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 $r - 1$
   - if $A[j] \leq x$
     - $i = i + 1$
   - exchange $A[i]$ with $A[j]$
4. exchange $A[i + 1]$ with $A[r]$
5. return $i + 1$
QuickSort: Time Complexity

**Question**: 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

**Question**: 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:
Quicksort: Time Complexity

**Question**: 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:

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

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

**Question**: 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:

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

Initially, \( p = 1, r = n \)

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

**Question**: Which factor would affect the efficiency of QuickSort?

always produces a 9-to-1 split
**QuickSort: Time Complexity**

**Question**: 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**

**Question**: 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: $O(n \lg n)$
Quicksort: Time Complexity

**Question**: 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**

**Question:** Which factor would affect the efficiency of **QuickSort**?

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

What is the WORST CASE?

Any split of constant proportionality.
Quicksort: Time Complexity

Worst Case:

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

**Question**: When would the worst case happen?
Quicksort: Time Complexity

Worst Case:

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

**Question**: When would the worst case happen?

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

Worst Case:

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

**Question**: 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:

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

**Question**: 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!
Impression & Intuition:

Quick sort performs quite well in practice.
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.
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  if p < r
2    q = PARTITION(A, p, r)
3    QUICKSORT(A, p, q - 1)
4    QUICKSORT(A, q + 1, r)

PARTITION(A, p, r)
1  x = A[r]
2  i = p - 1
3  for j = p to r - 1
4      if A[j] <= x
5          i = i + 1
6      exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  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 $O(n + X)$.

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**

\[
\text{RANDOMIZED-QUICKSORT}(A, p, r)
\]

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

**Goal:**

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

We will **NOT** attempt to analyze how many comparisons are made in **EACH** call to **Partition**.

**Randomized Partition**

\[
\text{RANDOMIZED-PARTITION}(A, p, r)
\]

1. \( i = \text{RANDOM}(p, r) \)
2. exchange \( A[r] \) with \( A[i] \)
3. **return** \( \text{PARTITION}(A, p, r) \)

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

1. \( x = A[r] \)
2. \( i = p - 1 \)
3. **for** \( j = p \) **to** \( r - 1 \)
4. **if** \( A[j] < x \)
5. \( i = i + 1 \)
7. exchange \( A[i + 1] \) with \( A[r] \)
8. **return** \( i + 1 \)
Randomized Quicksort: Expected Running Time

**Question**: 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

**Question**: 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.*
Question: 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

**Question**: 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 $\{z_1, z_2, ..., z_n\}$, with $z_i$ being the $i$th smallest element.
- $Z_{ij} = \{z_i, z_{i+1}, ..., z_j\}$: the set of elements between $z_i$ and $z_j$, inclusive.
Randomized Quicksort: Expected Running Time

**Question**: 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

**Question**: 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

**Question**: 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

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

Then, we have:

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

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

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

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

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

$$ = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{ z_i \text{ is compared to } z_j \} $$

$$ = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} $$

$$ < \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} $$

$$ = \sum_{i=1}^{n-1} O(\lg n) $$

$$ = O(n \lg n) $$
Randomized Quicksort: Expected Running Time

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

\[
Pr\{z_i \text{ is compared to } z_j\} = Pr\{z_i \text{ or } z_j \text{ is first pivot chosen from } Z_{ij}\}
\]
\[
= Pr\{z_i \text{ is first pivot chosen from } Z_{ij}\}
\]
\[
+ Pr\{z_j \text{ is first pivot chosen from } Z_{ij}\}
\]
\[
= \frac{1}{j - i + 1} + \frac{1}{j - i + 1}
\]
\[
= \frac{2}{j - i + 1}.
\]
Randomized Quicksort: Expected Running Time

**Question**: 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 \( k \).

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

```
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
```
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 \( i \).
5. for \( i = 1 \) to \( k \)
   6. \( C[i] = C[i] + C[i-1] \)
   7. // \( C[i] \) now contains the number of elements less than or equal to \( i \).
10. for \( j = A.length \) downto 1
12. \( C[A[j]] = C[A[j]] - 1 \)
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

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

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. \( C[A[j]] = C[A[j]] - 1 \)

**Stable**: numbers with the same value appear in the output array in the same order as they do in the input array.
Sorting in Linear Time: Counting Sort

10  for  \( j = A \cdot \text{length downto } 1 \)
12        \( C[A[j]] = C[A[j]] - 1 \)

(a)

(b)

(c)

(d)

(e)
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

\[
\text{RADIX-SORT}(A, d)
\]

1. for \( i = 1 \) to \( d \)
2. use a stable sort to sort array \( A \) on digit \( i \)

<table>
<thead>
<tr>
<th>329</th>
<th>720</th>
<th>720</th>
<th>329</th>
</tr>
</thead>
<tbody>
<tr>
<td>457</td>
<td>355</td>
<td>329</td>
<td>355</td>
</tr>
<tr>
<td>657</td>
<td>436</td>
<td>436</td>
<td>436</td>
</tr>
<tr>
<td>839</td>
<td>457</td>
<td>839</td>
<td>457</td>
</tr>
<tr>
<td>436</td>
<td>657</td>
<td>355</td>
<td>657</td>
</tr>
<tr>
<td>720</td>
<td>329</td>
<td>457</td>
<td>720</td>
</tr>
<tr>
<td>355</td>
<td>839</td>
<td>657</td>
<td>839</td>
</tr>
</tbody>
</table>
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
```
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, \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.

Lemma (8.4)

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

\textit{Proof} \quad For a value $r \leq b$, we view each key as having $d = \lfloor b/r \rfloor$ 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 $\Theta(d(n + 2^r)) = \Theta((b/r)(n + 2^r))$. \hfill $\blacksquare$
Sorting in Linear Time: Bucket Sort

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 $n - 1$
   - make $B[i]$ an empty list
4. for $i = 1$ to $n$
   - insert $A[i]$ into list $B[[nA[i]]]$
5. for $i = 0$ to $n - 1$
   - sort list $B[i]$ with insertion sort
6. concatenate the lists $B[0], B[1], \ldots, B[n-1]$ together in order
Sorting in Linear Time: Bucket Sort

BEGIN 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], . . . , B[n−1] together in order
END

- 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

\begin{algorithm}
\textbf{BUCKET-SORT}$(A)$
\begin{algorithmic}[1]
\State let $B[0..n-1]$ be a new array
\State $n = A.length$
\For{$i = 0$ to $n-1$}
\State make $B[i]$ an empty list
\EndFor
\For{$i = 1$ to $n$}
\State insert $A[i]$ into list $B[[nA[i]]]$\endFor
\For{$i = 0$ to $n-1$}
\State sort list $B[i]$ with insertion sort $O(n_i^2)$\endFor
\State concatenate the lists $B[0], B[1], \ldots, B[n-1]$ together in order
\end{algorithmic}
\end{algorithm}

- 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
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], …, 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

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

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

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

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

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

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

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

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

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

```plaintext
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
```
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 $3\lfloor n/2 \rfloor$ 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**

**RANDOMIZED-SELECT**($A, p, r, i$)

1. **if** $p == r$
   2. **return** $A[p]$
3. $q = \text{RANDOMIZED-PARTITION}(A, p, r)$
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**($A, q + 1, r, i - k$)
Selection in Expected Linear Time: \textsc{Randomized-Select}

\begin{verbatim}
\textsc{Randomized-Select}(A, p, r, i)
1     if \( p == r \)
2         return \( A[p] \)
3     \( q = \) \textsc{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 \textsc{Randomized-Select}(A, p, q - 1, i)
9     else return \textsc{Randomized-Select}(A, q + 1, r, i - k)
\end{verbatim}

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

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

**Question:** 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 $E[X_k] = 1/n$

```
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

**Question**: 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 \( E[X_k] = \frac{1}{n} \)

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

**Question:** 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**

$E[T(n)]:$ 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**

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

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

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

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

\textbf{SELECT}

1. Divide the input array into $\lceil n/5 \rceil$ groups of 5 elements each
   \begin{itemize}
   \item at most one group made up of the remaining $n \mod 5$ elements.
   \end{itemize}
2. Find the median of each of the $\lceil n/5 \rceil$ groups with \texttt{insertion-sort}.
3. Use \texttt{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 \texttt{SELECT} recursively:
   \begin{itemize}
   \item if $i < k$, find the $i$th smallest element on the low side
   \item if $i > k$, find the $(i - k)$th smallest element on the high side
   \end{itemize}
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.
Step 3: Use $\text{SELECT}$ recursively to find the median $m^*$ of the medians found in step 2.
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 \textbf{median} $m^*$ of the medians found in step 2.
Step 4: **Partition** the input array around \( m^* \).

- **A** and **C** are the lower half of the array.
- **B** and **D** are the upper half of the array.
- **A** and **D** are sorted by their medians.
- The median of medians is used to partition the array.
- Elements in **A** and **D** are less than or equal to \( m^* \) and greater than or equal to \( m^* \), respectively.
- Elements in **B** and **C** are greater than \( m^* \).
**Step 4: Partition** the input array around $m^*$. 

> $m^*$ or $< m^*$ are unknown only for elements in $A$ and $D$
**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 23, 2020

38 / 40
Step 5: Assume that $m^*$ is the $k$th smallest element.

- If $i = k$, then return $m^*$.
- Otherwise, use \texttt{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

\begin{itemize}
  \item $|C| \geq 3n/10 - 6$
  \item $|B| \geq 3n/10 - 6$
  \item median of medians
\end{itemize}
**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

\[ |C| \geq 3n/10 - 6 \]

\[ |B| \geq 3n/10 - 6; \]

**calls **SELECT** recursively on at most $7n/10 + 6$ elements.**
The \textbf{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
- \( T(7n/10 + 6) \): maximum cost for calling \texttt{Select} recursively.
- \( O(n) \):
  - divide the input array into 5-elements groups
  - find medians of all 5-elements groups, about \( 6 \times \lceil n/5 \rceil \)
  - \texttt{PARTITION} with the pivot \( m^* \)

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

Questions?

Office 819
majun@nju.edu.cn