Problem Solving
2-9 Sorting and Selection

MA Jun

Institute of Computer Software

April 23, 2020
Contents

1. Sorting

2. Selection
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
**QuickSort**

**Question**: What is the **KEY** idea of QuickSort?

To Be Sorted Recursively

- **pivot**
  - for any element in this segment, the key is **not greater than** pivot.
  - **small**
  - **large**
  - for any element in this segment, the key is **greater than** pivot.
**Quicksort**

**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.

**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: \textbf{PARTITION}

\begin{center}
\textbf{PARTITION}(A, p, r)
\begin{tabular}{l}
1. \(x = A[r]\) \\
2. \(i = p - 1\) \\
3. \textbf{for} \(j = p\ \text{to} \ r - 1\) \\
4. \quad \textbf{if} \(A[j] \leq x\) \\
5. \quad \quad \(i = i + 1\) \\
6. \quad \text{exchange} \(A[i]\) \text{ with } \(A[j]\) \\
7. \text{exchange} \(A[i + 1]\) \text{ with } \(A[r]\) \\
8. \textbf{return} \(i + 1\)
\end{tabular}
\end{center}

\textbf{Question} : How to prove the correctness of \textbf{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 \(A[k] = x\).
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:

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

\( n_1, n_2 \) vary and depend on \( q = \text{PARTITION}(A, p, r) \)
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

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!
Quicksort: Time Complexity

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.

\[ T(n) = O(n \lg n) \]
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 Partition over the entire execution of Quicksort on an $n$-element array. Then the running time of Quicksort is $O(n + X)$.

Proof.

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

**Randomized Quicksort**

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

1. \textbf{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)

**Randomized Partition**

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

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

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

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

**Goal:**

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

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

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

\[ X: \text{the TOTAL number of comparisons performed in all calls to Partition.} \]

- 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.

\( X_{ij} \): 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 $X$?

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

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

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

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

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

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

\[
= O(n \lg n)
\]
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}.
\]
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

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
- $n! \leq 2^h \iff h \geq \lg n! = \Omega(n \lg n)$
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 \( k \)
   - \( C[i] = 0 \)
3. for \( j = 1 \) to \( A.length \)
   - \( C[A[j]] = C[A[j]] + 1 \)
   - // \( C[i] \) now contains the number of elements equal to \( i \).
4. for \( i = 1 \) to \( k \)
   - \( C[i] = C[i] + C[i - 1] \)
   - // \( C[i] \) now contains the number of elements less than or equal to \( i \).
5. for \( j = A.length \) downto 1
   - \( B[C[A[j]]] = A[j] \)
   - \( C[A[j]] = C[A[j]] - 1 \)

---

**stable:** numbers with the same value appear in the output array in the same order as they do in the input array.

---

Matrix:

\[
\begin{array}{cccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
4 & 2 & 5 & 3 & 0 & 2 & 3 & 0 & 3 \\
\end{array}
\]
Sorting in Linear Time: Counting Sort

```
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 = [2, 0, 2, 3, 0, 1]

(b) B = [0, 0, 3]
    C = [2, 2, 4, 7, 7, 8]

(c) B = [0, 0, 3]
    C = [2, 2, 4, 6, 7, 8]

(d) B = [0, 0, 3]
    C = [1, 2, 4, 6, 7, 8]

(e) B = [0, 0, 3]
    C = [1, 2, 4, 5, 7, 8]
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

\[
\text{RADIX-SORT}(A, d) \\
1 \quad \text{for } i = 1 \text{ to } d \\
2 \quad \text{use a stable sort to sort array } A \text{ on digit } i
\]
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, 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 $\Theta(d(n + 2^r)) = \Theta((b/r)(n + 2^r))$. 

$\blacksquare$
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
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
```
Sorting in Linear Time: Bucket Sort

```plaintext
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

T(n) = Θ(n) + \sum_{i=0}^{n−1} O(n_i^2)
```

- 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

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

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

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

\[
= \Theta(n)
\]

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

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

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

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

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

\[
\text{Minimum}(A)
\]

1. \( \text{min} = A[1] \)
2. for \( i = 2 \) to \( A\.\text{length} \)
3. if \( \text{min} > A[i] \)
4. \( \text{min} = A[i] \)
5. return \( \text{min} \)
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*

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** \(i == k\) \quad // \text{the pivot value is the answer}
6. **return** \(A[q]\)
7. **elseif** \(i < k\)
8. **return** \(\text{RANDOMIZED-SELECT}(A, p, q - 1, i)\)
9. **else return** \(\text{RANDOMIZED-SELECT}(A, q + 1, r, i - k)\)

Similar to **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**?

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

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

\[
E[T(n)] \\
\leq E \left[ \sum_{k=1}^{n} X_k \cdot T(\max(k - 1, n - k)) + O(n) \right] \\
= \sum_{k=1}^{n} E[X_k \cdot T(\max(k - 1, n - k))] + O(n) \quad \text{(by linearity of expectation)} \\
= \sum_{k=1}^{n} E[X_k] \cdot E[T(\max(k - 1, n - k))] + O(n) \quad \text{(by equation (C.24))} \\
= \sum_{k=1}^{n} \frac{1}{n} \cdot E[T(\max(k - 1, n - k))] + O(n) \quad \text{(by equation (9.1))} .
\]

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

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

\textbf{SELECT}

1. Divide the input array into $\lceil n/5 \rceil$ 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 $\lceil n/5 \rceil$ groups with \textit{insertion-sort}.
3. Use \texttt{SELECT} recursively to find the median $m^*$ of the medians found in step 2.
4. \textbf{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:
   - 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 \([n/5]\) groups with INSERTION-SORT.

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

$> m^*$ or $< m^*$ are unknown only for elements in $A$ and $D$.
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 **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 **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 819
majun@nju.edu.cn