Lectures 6 and 7: Merge-sort and Maximum Subarray Problem
COMS10007 - Algorithms

Dr. Christian Konrad

12.02.2020
Definition of the Sorting Problem

Sorting Problem

- **Input:** An array $A$ of $n$ numbers
- **Output:** A reordering of $A$ s.t. $A[0] \leq A[1] \leq \cdots \leq A[n-1]

Why is it important?

- Practical relevance: Appears almost everywhere
- Fundamental algorithmic problem, rich set of techniques
- There is a non-trivial lower bound for sorting (rare!)

Insertion Sort

- Worst-case and average-case runtime $O(n^2)$
- Surely we can do better?!
Insertion sort in Practice on Worst-case Instances

\[\begin{array}{c|c|c|c|c}
 n & 46929 & 102428 & 364178 & 1014570 \\
 secs & 1.03084 & 4.81622 & 61.2737 & 497.879 \\
\end{array}\]
**Definition** (in place)
A sorting algorithm is *in place* if at any moment at most $O(1)$ array elements are stored outside the array.

\[
\begin{array}{cccccccccc}
a_0 & a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 & a_9 & a_{10} \\
\end{array}
\]

\[
\quad \quad O(1)
\]

**Example:** Insertion-sort is in place

**Definition** (stability)
A sorting algorithm is *stable* if any pair of equal numbers in the input array appear in the same order in the sorted array.

**Example:** Insertion-sort is stable
Records, Keys, and Satellite Data

**Sorting Complex Data**

- In reality, data that is to be sorted is rarely entirely numerical (e.g. sort people in a database according to their last name)
- A data item is often also called a **record**
- The **key** is the part of the record according to which the data is to be sorted
- Data different to the key is also referred to as **satellite data**

<table>
<thead>
<tr>
<th>family name</th>
<th>first name</th>
<th>data of birth</th>
<th>role</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smith</td>
<td>Peter</td>
<td>02.10.1982</td>
<td>lecturer</td>
</tr>
<tr>
<td>Hills</td>
<td>Emma</td>
<td>05.05.1975</td>
<td>reader</td>
</tr>
<tr>
<td>Jones</td>
<td>Tom</td>
<td>03.02.1977</td>
<td>senior lecturer</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Observe:** Stability makes more sense when sorting complex data as opposed to numbers
Key Idea:
- Suppose that left half and right half of array is sorted
- Then we can merge the two sorted halves to a sorted array in $O(n)$ time:

Merge Operation
- Copy left half of $A$ to new array $B$
- Copy right half of $A$ to new array $C$
- Traverse $B$ and $C$ simultaneously from left to right and write the smallest element at the current positions to $A$
Example: Merge Operation

\[
\begin{array}{cccccccc}
A & 1 & 4 & 9 & 10 & 3 & 5 & 7 & 11 \\
\end{array}
\]
### Example: Merge Operation

<table>
<thead>
<tr>
<th>A</th>
<th>1</th>
<th>4</th>
<th>9</th>
<th>10</th>
<th>3</th>
<th>5</th>
<th>7</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>1</td>
<td>4</td>
<td>9</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example: Merge Operation

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1 4</td>
<td>3 5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9 10</td>
<td>7 11</td>
</tr>
</tbody>
</table>
Example: Merge Operation

<table>
<thead>
<tr>
<th>A</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>1</td>
<td>4</td>
<td>9</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

Dr. Christian Konrad
Lectures 6 and 7
Example: Merge Operation

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>B</th>
<th>1</th>
<th>4</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>3</th>
<th>5</th>
<th>7</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example: Merge Operation

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>4</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>11</td>
</tr>
</tbody>
</table>
**Example: Merge Operation**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 3 4</td>
<td>1 4 9 10</td>
<td>3 5 7 11</td>
</tr>
</tbody>
</table>

Dr. Christian Konrad
Example: Merge Operation

\[ A \]
\[ 1 \quad 3 \quad 4 \quad 5 \]

\[ B \]
\[ 1 \quad 4 \quad 9 \quad 10 \]

\[ C \]
\[ 3 \quad 5 \quad 7 \quad 11 \]
Example: Merge Operation

$A = \begin{bmatrix} 1 & 3 & 4 & 5 & 7 & \_ & \_ \end{bmatrix}$

$B = \begin{bmatrix} 1 & 4 & 9 & 10 \end{bmatrix}$

$C = \begin{bmatrix} 3 & 5 & 7 & 11 \end{bmatrix}$
### Example: Merge Operation

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>4</td>
<td>9</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Analysis: Merge Operation

Merge Operation

- **Input:** An array $A$ of integers of length $n$ ($n$ even) such that $A[0, \frac{n}{2} - 1]$ and $A[\frac{n}{2}, n - 1]$ are sorted
- **Output:** Sorted array $A$

Runtime Analysis:

1. Copy left half of $A$ to $B$: $O(n)$ operations
2. Copy right half of $A$ to $C$: $O(n)$ operations
3. Merge $B$ and $C$ and store in $A$: $O(n)$ operations

Overall: $O(n)$ time in worst case

How can we establish that left and right halves are sorted?

Divide and Conquer!
Merge Sort: A Divide and Conquer Algorithm

```python
Require: Array A of n numbers
if n = 1 then
    return A
A[0, ⌊n/2⌋] ← MergeSort(A[0, ⌊n/2⌋])
A[⌊n/2⌋ + 1, n − 1] ← MergeSort(A[⌊n/2⌋ + 1, n − 1])
A ← Merge(A)
return A
```

**MergeSort**

Structure of a Divide and Conquer Algorithm

- **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
- **Conquer** the subproblems by solving them recursively. If the subproblems are small enough, just solve them in a straightforward manner.
- **Combine** the solutions to the subproblems into the solution for the original problem.
Analyzing MergeSort: An Example

```
12  9  7  2  3  8  15  7
```

```
12  9  7  2
3  8  15  7
```

```
12  9
7  2
3  8
15  7
```

```
12
9
7
2
3
8
15
7
```
Analyzing Merge Sort

Analysis Idea:
- We need to sum up the work spent in each node of the recursion tree.
- The recursion tree in the example is a complete binary tree.

Definition: A tree is a complete binary tree if every node has either 2 or 0 children.

Definition: A tree is a binary tree if every node has at most 2 children.

(we will talk about trees in much more detail later in this unit)

Questions:
- How many levels?
- How many nodes per level?
- Time spent per node?
Number of Levels

Level 1

Level 2

Level 3

Level 4
Number of Levels (2)

**Level i:**
- $2^{i-1}$ nodes (at most)
- Array length in level $i$ is $\left\lceil \frac{n}{2^{i-1}} \right\rceil$ (at most)
- Runtime of merge operation for each node in level $i$: $O\left(\frac{n}{2^{i-1}}\right)$

**Number of Levels:**
- Array length in last level $l$ is 1: $\left\lceil \frac{n}{2^{l-1}} \right\rceil = 1$

\[
\frac{n}{2^{l-1}} \leq 1 \Rightarrow n \leq 2^{l-1} \Rightarrow \log(n) + 1 \leq l
\]

- Array length in last but one level $l - 1$ is 2: $\left\lceil \frac{n}{2^{l-2}} \right\rceil = 2$

\[
\frac{n}{2^{l-2}} > 1 \Rightarrow n > 2^{l-2} \Rightarrow \log(n) + 2 > l
\]

\[
\log(n) + 1 \leq l < \log(n) + 2
\]

Hence, \( l = \left\lceil \log n \right\rceil + 1 \).
Runtime of Merge Sort

**Sum up Work:**
- Levels:
  \[ l = \lceil \log n \rceil + 1 \]
- Nodes on level \( i \):
  at most \( 2^{i-1} \)
- Array length in level \( i \):
  at most \( \left\lceil \frac{n}{2^{i-1}} \right\rceil \)

**Worst-case Runtime:**

\[
\sum_{i=1}^{\lceil \log n \rceil + 1} 2^{i-1} O \left( \left\lceil \frac{n}{2^{i-1}} \right\rceil \right) = \sum_{i=1}^{\lceil \log n \rceil + 1} 2^{i-1} O \left( \frac{n}{2^{i-1}} \right) = \sum_{i=1}^{\lceil \log n \rceil + 1} O(n) = (\lceil \log n \rceil + 1) O(n) = O(n \log n).
\]
Merge sort in Practice on Worst-case Instances

\[ n \begin{array}{cccc}
46929 & 102428 & 364178 & 1014570 \\
\text{secs} & 1.03084 & 4.81622 & 61.2737 & 497.879 \text{ (Insertion-sort)} \\
\text{secs} & 0.007157 & 0.015802 & 0.0645791 & 0.169165 \text{ (Merge-sort)}
\end{array} \]
Divide and Conquer Algorithm:

Let \( A \) be a divide and conquer algorithm with the following properties:

1. \( A \) performs two recursive calls on input sizes at most \( n/2 \)
2. The conquer operation in \( A \) takes \( O(n) \) time

Then:

\[
A \text{ has a runtime of } O(n \log n).
\]
Stability and In Place Property?

- Merge sort is stable
- Merge sort does not sort in place
Buy Low, Sell High Problem

- **Input:** An array of \( n \) integers
- **Output:** Indices \( 0 \leq i < j \leq n - 1 \) such that \( A[j] - A[i] \) is maximized
Buy Low, Sell High Problem

- **Input:** An array of $n$ integers
- **Output:** Indices $0 \leq i < j \leq n - 1$ such that $A[j] - A[i]$ is maximized
Focus on Array of Changes:

<table>
<thead>
<tr>
<th>Day</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>100</td>
<td>113</td>
<td>110</td>
<td>85</td>
<td>105</td>
<td>102</td>
<td>86</td>
<td>63</td>
<td>81</td>
<td>101</td>
<td>94</td>
<td>106</td>
</tr>
<tr>
<td>$\Delta$</td>
<td>13</td>
<td>-3</td>
<td>-25</td>
<td>20</td>
<td>-3</td>
<td>-16</td>
<td>-23</td>
<td>18</td>
<td>20</td>
<td>-7</td>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

Maximum Subarray Problem

- **Input:** Array $A$ of $n$ numbers
- **Output:** Indices $0 \leq i \leq j \leq n - 1$ such that $\sum_{l=i}^{j} A[l]$ is maximum.

**Trivial Solution:** $O(n^3)$ runtime

- Compute subarrays for every pair $i, j$
- There are $O(n^2)$ pairs, computing the sum takes time $O(n)$. 

Dr. Christian Konrad
Lectures 6 and 7
Focus on Array of Changes:

<table>
<thead>
<tr>
<th>Day</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>100</td>
<td>113</td>
<td>110</td>
<td>85</td>
<td>105</td>
<td>102</td>
<td>86</td>
<td>63</td>
<td>81</td>
<td>101</td>
<td>94</td>
<td>106</td>
</tr>
<tr>
<td>Δ</td>
<td>13</td>
<td>-3</td>
<td>-25</td>
<td>20</td>
<td>-3</td>
<td>-16</td>
<td>-23</td>
<td>18</td>
<td>20</td>
<td>-7</td>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

Maximum Subarray Problem

- **Input:** Array $A$ of $n$ numbers
- **Output:** Indices $0 \leq i \leq j \leq n - 1$ such that $\sum_{i=0}^{j} A[l]$ is maximum.

**Trivial Solution:** $O(n^3)$ runtime

- Compute subarrays for every pair $i, j$
- There are $O(n^2)$ pairs, computing the sum takes time $O(n)$.
Divide and Conquer:
Compute maximum subarrays in left and right halves of initial array

\[ A = L \circ R \]

Combine:
Given maximum subarrays in \( L \) and \( R \), we need to compute
maximum subarray in \( A \)

Three cases:
1. Maximum subarray is entirely included in \( L \) ✓
2. Maximum subarray is entirely included in \( R \) ✓
3. Maximum subarray crosses midpoint, i.e., \( i \) is included in \( L \) and \( j \) is included in \( R \)
Maximum Subarray Crosses Midpoint:

- Find maximum subarray \( A[i, j] \) such that \( i \leq \frac{n}{2} \) and \( j > \frac{n}{2} \) (assume that \( n \) is even)

- Observe that:
  \[
  \sum_{l=i}^{j} A[l] = \sum_{l=i}^{\frac{n}{2}} A[i] + \sum_{l=\frac{n}{2}+1}^{j} A[l].
  \]

Two Independent Subproblems:

- Find index \( i \) such that \( \sum_{l=i}^{\frac{n}{2}} A[i] \) is maximized

- Find index \( j \) such that \( \sum_{l=\frac{n}{2}+1}^{j} A[l] \) is maximized

We can solve these subproblems in time \( O(n) \). (how?)
Maximum Subarray Problem - Summary

**Require:** Array $A$ of $n$ numbers

if $n = 1$ then
    return $A$
Recursively compute max. subarray $S_1$ in $A[0, \lfloor \frac{n}{2} \rfloor]$
Recursively compute max. subarray $S_2$ in $A[\lfloor \frac{n}{2} \rfloor + 1, n - 1]$
Compute maximum subarray $S_3$ that crosses midpoint
return Heaviest of the three subarrays $S_1, S_2, S_3$

Recursive Algorithm for the Maximum Subarray Problem

**Analysis:**
- Two recursive calls with inputs that are only half the size
- Conquer step requires $O(n)$ time
- Identical to Merge Sort, runtime $O(n \log n)$!