Introduction

• Algorithms and Data Structures
• Peaks
  • Algorithm 1
  • Algorithm 2
  • Algorithm 3
Introduction

• Algorithms and Data Structures
  • Peaks
    • Algorithm 1
    • Algorithm 2
    • Algorithm 3
Algorithms and Data Structures

- **Algorithmic problem.** Precisely defined relation between input and output.

- **Algorithm.** Method to solve an algorithm problem.
  - *Discrete* and *unambiguous* steps.
  - Mathematical abstraction of a program.

- **Data structure.** Method for organizing data to enable queries and updates.
Example: Find max

- **Find max.** Given a table A[0..n-1], find an index i, such that A[i] is maximal.
  - **Input.** Table A[0..n-1].
  - **Output.** An index i such that A[i] ≥ A[j] for all indices j ≠ i.

- **Algorithm.**
  - Process A from left-to-right and maintain value and index of maximal value seen so far.
  - Return index.

```
0  1  2  3  4  5  6  7  8  9  10 11 12 13 14
 1  3  7 15 17 11  2  3  6  8  7  5  9  5 23
```
Description of Algorithms

• Natural language.
  • Process A from left-to-right and maintain value and index of maximal value seen so far.
  • Return index.

• Program.

• Pseudocode.

```
public static int findMax(int[] A) {
    int max = 0;
    for(i = 0; i < A.length; i++)
        if (A[i] > A[max]) max = i;
    return max;
}
```

```
FINDMAX(A, n)
    max = 0
    for i = 0 to n-1
        if (A[i] > A[max]) max = i
    return max
```
Introduction

• Algorithms and Data Structures
• Peaks
  • Algorithm 1
  • Algorithm 2
  • Algorithm 3
Peaks

• **Peak.** A[i] is a peak if A[i] is as least as large as it's neighbors:
  • A[i] is a peak if A[i-1] ≤ A[i] ≥ A[i+1] for i ∈ {1, .., n-2}
  • A[0] is a peak if A[0] ≥ A[1].

- **Peak finding.** Given a table A[0..n-1], find an index i such that A[i] is a peak.
  • **Input.** A table A[0..n-1].
  • **Output.** An index i such that A[i] is a peak.
Introduction

• Algorithms and Data Structures
• Peaks
  • Algorithm 1
  • Algorithm 2
  • Algorithm 3
Algorithm 1

- **Algorithm 1.** For each entry check if it is a peak. Return the index of the first peak.

```
PEAK1(A, n)
    for i = 1 to n-2
```

- **Pseudocode.**

- **Challenge.** How do we analyze the algorithm?
Theoretical Analysis

- **Running time/time complexity.**
  - \(T(n) = \text{number of steps}\) that the algorithm performs on input of size \(n\).
- **Steps.**
  - Read/write to memory (\(x := y, A[i], i = i + 1, \ldots\))
  - Arithmetic/boolean operations (\(+, -, *, /, \%, &&, ||, &, |, ^, ~\))
  - Comparisons (\(<, >, <=, >=, =, \neq\))
  - Program flow (if-then-else, while, for, goto, funktion call, ..)
- **Worst-case time complexity.** Maximal running time over all input of size \(n\).
Theoretical Analysis

• **Running time.** What is the running time $T(n)$ for algorithm 1?

```
PEAK1(A, n)
    for i = 1 to n-2
```

$T(n) = c_1 + (n-2)c_2 + c_3$

• $T(n)$ is a linear function of $n$: $T(n) = an + b$
• **Asymptotic notation.** $T(n) = \Theta(n)$
• **Experimental analysis.**
  • What is the experimental running time of algorithm 1?
  • How does the experimental analysis compare to the theoretical analysis?
Peaks

- Algorithm 1 finds a peak in $\Theta(n)$ time.
- Theoretical and experimental analysis agrees.
- **Challenge.** Can we do better?
Introduction

• Algorithms and Data Structures
• Peaks
  • Algorithm 1
  • Algorithm 2
  • Algorithm 3
Algorithm 2

- **Observation.** A maximal entry $A[i]$ is a peak.
- **Algorithm 2.** Find a maximal entry in $A$ med $\text{FINDMAX}(A, n)$.

```
\text{FINDMAX}(A, n)
max = 0
for i = 0 to n-1
    if (A[i] > A[max]) max = i
return max
```
Theoretical Analysis

• **Running time.** What is the running time $T(n)$ for algorithm 2?

```plaintext
FINDMAX(A, n)
  max = 0
  for i = 0 to n-1
    if (A[i] > A[max]) max = i
  return max
```

$T(n) = c_4 + n \cdot c_5 + c_6 = \Theta(n)$

• **Experimental analysis.** Better constants?
Peaks

• Theoretical analysis.
  • Algorithm 1 and 2 find a peak in $\Theta(n)$ time.
• Experimental analysis.
  • Algorithm 1 and 2 run in $\Theta(n)$ time in practice.
  • Algorithm 2 is a constant factor faster than algorithm 1.
• Challenge. Can we do significantly better?
Introduction

• Algorithms and Data Structures
• Peaks
  • Algorithm 1
  • Algorithm 2
  • Algorithm 3
Algorithm 3

• Clever idea.
  • Consider any entry \( A[i] \) and its neighbors \( A[i-1] \) and \( A[i+1] \).
  • Where can a peak be relative to \( A[i] \)?
    • Neighbor are \( \leq A[i] \) \( \implies \) \( A[i] \) is a peak.
    • Otherwise \( A \) is increasing in at least one direction \( \implies \) peak must exist in that direction.

\[
\begin{array}{ccc}
  i-1 & i & i+1 \\
  3 & 10 & 7 \\
\end{array}
\quad
\begin{array}{ccc}
  i-1 & i & i+1 \\
  12 & 10 & 7 \\
\end{array}
\]

\[
\begin{array}{ccc}
  i-1 & i & i+1 \\
  3 & 10 & 15 \\
\end{array}
\quad
\begin{array}{ccc}
  i-1 & i & i+1 \\
  12 & 10 & 15 \\
\end{array}
\]

• Challenge. How can we turn this into a fast algorithm?
Algorithm 3

- Algorithm 3.
  - Consider the middle entry $A[m]$ and neighbors $A[m-1]$ and $A[m+1]$.
  - If $A[m]$ is a peak, return $m$.
  - Otherwise, continue search recursively in half with the increasing neighbor.
Algorithm 3

- **Algorithm 3.**
  - Consider the *middle* entry $A[m]$ and neighbors $A[m-1]$ and $A[m+1]$.
  - If $A[m]$ is a peak, return $m$.
  - Otherwise, continue search recursively in half with the increasing neighbor.

```
PEAK3(A, i, j)
    m = ⌊(i+j)/2⌋
    if $A[m] ≥$ neighbors return $m$
    elseif $A[m-1] > A[m]$
        return PEAK3(A, i, m-1)
    elseif $A[m] < A[m+1]$
        return PEAK3(A, m+1, j)
```

<table>
<thead>
<tr>
<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>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>7</td>
<td>15</td>
<td>17</td>
<td>11</td>
<td>2</td>
<td>3</td>
<td>6</td>
<td>8</td>
<td>7</td>
<td>5</td>
<td>9</td>
<td>5</td>
<td>23</td>
</tr>
</tbody>
</table>
• **Running time.**
• A recursive call takes constant time.
• How many recursive calls?
  • A recursive call **halves** size of interval. We stop when table has size 1.
    • 1. recursive call: $n/2$
    • 2. recursive call: $n/4$
    • …. 
    • $k^{th}$. recursive call: $n/2^k$
    • …. 
• $\implies$ After $\sim \log_2 n$ recursive call table has size $\leq 1$.
• $\implies$ Running time is $\Theta(\log n)$
• **Experimental analysis.** Significantly better?

```python
PEAK3(A,i,j)
    m = \lfloor (i+j)/2 \rfloor
    if A[m] \geq \text{neighbors return } m
    elseif A[m-1] > A[m]
        return PEAK3(A,i,m-1)
    elseif A[m] < A[m+1]
        return PEAK3(A,m+1,j)
```

Peaks

• **Theoretical analysis.**
  - Algorithm 1 and 2 finds a peak in $\Theta(n)$ time.
  - Algorithm 3 finds a peak in $\Theta(\log n)$ time.

• **Experimental analysis.**
  - Algorithm 1 and 2 run in $\Theta(n)$ time in practice.
  - Algorithm 2 is a constant factor faster than algorithm 1.
  - Algorithm 3 is much, much faster than algorithm 1 and 3.
Introduction

- Algorithms and Data Structures
- Peaks
  - Algorithm 1
  - Algorithm 2
  - Algorithm 3