Algorithms and Data Structures

One Problem, Four Algorithms

Ulf Leser
Content of this Lecture

- The Max-Subarray Problem
- Naïve Solution
- Better Solution
- Best Solution
Where is the Sun?

Source: http://www.layoutsparks.com
How can we find the Sun Algorithmically?

• Assume pixel (RGB) representation
• The sun obviously is bright
• RGB colors can be transformed into brightness scores
• The sun is the brightest spot
  – Compute an average brightness for entire picture
  – Subtract from each brightness value (will yield negative values)
  – Find the shape (spot) such that the sum of its brightness values is maximal
Size of the Spot not Pre-Determined
Example (Shapes: only Rectangles)

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>6</th>
<th>8</th>
<th>6</th>
<th>5</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>9</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>6</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>8</td>
<td>8</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>9</td>
<td>8</td>
<td>8</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

Avg. ~4

<table>
<thead>
<tr>
<th></th>
<th>-3</th>
<th>2</th>
<th>4</th>
<th>2</th>
<th>1</th>
<th>-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>-2</td>
<td>-2</td>
<td></td>
</tr>
<tr>
<td>-2</td>
<td>3</td>
<td>2</td>
<td>-1</td>
<td>-2</td>
<td>-3</td>
<td></td>
</tr>
<tr>
<td>-3</td>
<td>-1</td>
<td>-2</td>
<td>0</td>
<td>-3</td>
<td>-3</td>
<td></td>
</tr>
<tr>
<td>-2</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>-1</td>
<td>-2</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>-1</td>
<td></td>
</tr>
</tbody>
</table>
Max-Subarray Problem

• Today, we solve a simpler problem (1D versus 2D)
• Definition (Max-Subarray Problem)

Assume an array $A$ of integers. Find the subarray $A^*$ of $A$ such that the sum of the values in $A^*$ is maximal over all subarrays of $A$ and $|A^*| > 0$.

• Remarks
  - We only want the maximal value, not the borders of $A^*$
  - Cells have positive and negative values
  - Length of the subarray $A^*$ is not fixed

\[
\begin{array}{cccccccc}
-2 & 0 & 4 & 3 & 4 & -6 & -1 & 12 & -2 & 0 & 15 \\
\end{array}
\]
Types of Algorithms

• Creating an algorithm is between engineering and art
• Different **fundamental patterns** (non exhaustive list)
  - **Greedy**: Find some promising start point and expand aggressively until it complete solution is found
    • Usually fast, but doesn’t find the optimal solution
  - **Exhaustive**: Test all possible solutions and find the one that is best
    • Sometimes the only choice if optimality is asked for
  - **Divide & Conquer**: Break your problem into smaller ones until these are so easy that they can be solved directly; construct solutions for “bigger” problems from these small solutions
    - **Dynamic programming**
    - **Backtracking**
    - …
A Greedy Solution

- Promising start point: Find maximal value in $A$
- Aggressive expansion: Expand in both directions until sum decreases
- Complexity?
A Greedy Solution

- Promising start point: Find maximal value in array
- Aggressive expansion: Expand in both directions until sum decreases
- Complexity? (Let $n = |A|$)
  - $O(n)$ to find maximal value
  - $O(n)$ expansion steps in worst case
  - $O(n)$ together
- Correct?
A Greedy Solution

- Promising start point: Find maximal value in array
- Aggressive expansion: Expand in both directions until sum decreases
- Complexity? (Let $n=|A|$)
  - $O(n)$ together
- Correct?

\[
\begin{array}{cccccccc}
-2 & 0 & 4 & 3 & 4 & -3 & -1 & 12 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
-2 & 0 & 4 & 3 & 4 & -3 & -1 & 12 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
-2 & 0 & 4 & 3 & 4 & -3 & -1 & 12 \\
\end{array}
\]
Content of this Lecture

• The Max-Subarray Problem
• Naïve Solution
• Better Solution
• Best Solution
Exhaustive Solution

A: array_of_integer;

n := |A|;
m := -maxint;

for i := 1 ... n do
    for j := i ... n do
        s := 0;
        for k := i ... j do
            s := s + A[k];
        end for;
        if s>m then
            m := s;
        end if;
    end for;
end for;
return m;
Complexity

A: array_of_integer;
\[ n := |A|; \]
\[ m := -\text{maxint}; \]
for i := 1 \ldots n do
  for j := i \ldots n do
    s := 0;
    for k := i \ldots j do
      s := s + A[k];
    end for;
    if s > m then
      m := s;
    end if;
  end for;
end for;
return m;

- Outmost loop: \( n \) times
- j-loop: \( n \) times (worst-case)
- Inner loop: \( n \) times
- Together: \( \mathcal{O}(n^3) \)

- But: We are summing up the same numbers again and again
- We perform redundant work
- More clever ways?
Exhaustive Solution

- First sum: \( A[1] \)
- 4th: ...

- Every next sum actually is the previous sum plus the next cell
- How can we reuse the previous sum?
Exhaustive Solution, Improved

- Every next sum is the previous sum plus the next cell
- Complexity: $O(n^2)$
Content of this Lecture

• The Max-Subarray Problem
• Naïve Solution
• Better Solution
• Best Solution
Divide and Conquer

• Of course, we can break our problem into smaller ones by looking only at parts of the array

• One scheme: Assume $A = A_1 | A_2$
  - With “|” meaning array concatenation and $|A_1| = |A_2| (+0/1)$

• The max-subarray (msa) of $A$ …
  - either lies in $A_1$ – can be found by solving $msa(A_1)$
  - or in $A_2$ – can be found by solving $msa(A_2)$
  - or partly in $A_1$ and partly in $A_2$
    • Can be solved by summing-up the msa in $A_1/A_2$ that aligns with the right/left end of $A_1/A_2$

• We divide the problem into smaller ones and create the “bigger” solution from the “smaller” solutions
Algorithm (for simplicity, assume $|A|=2^x$ for some x)

```plaintext
function msa (A: array_of_integer) {
    n := |A|;
    if (n=1) then
        if A[1]>0 then
            return A[1]
        else
            return 0;
        end if;
    m := n/2;  # Assume even sizes
    A1 := A[1..m];
    A2 := A[m+1..n];
    l1 := rmax(A1);
    l2 := lmax(A2);
    m := max( msa(A1), l1 + l2, msa(A2));
    return m;
}

function rmax (A: array_of_integer) {
    n := |A|;
    s := 0;
    m := -maxint;
    for i := n .. 1 do
        s := s + A[i];
        if s>m then
            m := s;
        end if;
    end for;
    return m;
}
```
Example

-2 3 1 3 4 -3 -4 2

-2 3 1 3

4 -3 -4 2

-2 3

1 3

4 -3

-4 2

• Solution 11

• Solutions 7, 4
  - rmax/lmax: 7, 4

• Solutions 3, 4, 4, 2
  - rmax/lmax: 3, 4, 4, 0
Complexity

• This time it is not so easy …
• Complexity of lmax / rmax?

```plaintext
function rmax (A: array_of_integer) {
    n := |A|;
    s := 0;
    m := -maxint;
    for i := n .. 1 do
        s := s + A[i];
        if s > m then
            m := s;
        end if;
    end for;
    return m;
}
```
 Complexity

- This time it is not so easy ...
- Complexity of $\text{lmax}/\text{lmax}$?
  - $O(n)$
- Let $T(n)$ be the number of steps necessary to execute the algorithm for $|A|=n$
  - In each level, $n'=n/2$
  - The two sub-solutions require $T(n')$ each
- How does $T(n)$ depend on $T(n/2)$?

```plaintext
function msa (A: array_of_integer) {
  n := |A|;
  if (n=1) then
    if A[1]>0 then
      return A[1]
    else
      return 0;
  end if;
  m := n/2;  # Assume even sizes
  A1 := A[1...m];
  A2 := A[m+1...n];
  l1 := rmax(A1);
  l2 := lmax(A2);
  m := max( msa(A1), l1+l2, msa(A2));
  return m;
}
```
function msa (A: array_of_integer) {
    n := |A|;
    if (n=1) then
        if A[1]>0 then
            return A[1]
        else
            return 0;
    end if;
    m := n/2;  # Assume even sizes
    A1 := A[1..m];
    A2 := A[m+1..n];
    l1 := rmax(A1);
    l2 := lmax(A2);
    m := max( msa(A1), l1+l2, msa(A2));
    return m;
}

• For constants $c_1$, $c_2$
• $T(n) = 2 \cdot T(n/2) + c_1 \cdot n$
• Further: $T(1) = c_2$
• Iterative substitution yields
  
  $T(n) = 2 \cdot T(n/2) + c_2 \cdot n =$
  $= 2(2 \cdot T(n/4) + c_1 n/2) + c_1 n = 4T(n/4) + c_1 n + c_1 n =$
  $= 4(2 \cdot T(n/8) + c_1 n/2) + 2c_1 n = 8T(n/8) + 3c_1 n = ...$
  $2^{\log(n)} \cdot c_2 + c_1 n \cdot \log(n) =$
  $c_2 n + c_1 n \cdot \log(n) = O(n \cdot \log(n))$
Same Problem, Different Algorithms

- **Naive:** $O(n^3)$
- **Less naive, but still exhaustive:** $O(n^2)$
- **Divide & Conquer:** $O(n \cdot \log(n))$

- **The problem:** $O(n)$
Content of this Lecture

• The Max-Subarray Problem
• Naïve Solution
• Better Solution
• Linear Solution
Let’s Think again – More Carefully

- Let’s use another strategy for dividing the problem
- Let’s look at the solution for A[1], A[1..2], A[1..3], …
- What can we say about the msa for \( A^{i+1} = A[1..i+1] \), given the msa of \( A^i = A[1..i] \)?

\[
\begin{array}{ccccccc}
-2 & 0 & 4 & 3 & 4 & -3 & -1 & 6 \\
\end{array}
\]
Let’s Think again – More Carefully

- Let’s use another strategy for dividing the problem
- Let’s look at the solution for A[1], A[1..2], A[1..3], …
- What can we say about the msa for A\textsuperscript{i+1}=A[1..i+1], given the msa of A\textsuperscript{i}=A[1..i]? 

\begin{center} 
\begin{tabular}{|c|c|c|c|c|} 
\hline 
-2 & 0 & 4 & 3 & 4 \\
\hline 
-3 & -1 & 6 \\
\hline 
\end{tabular} 
\end{center} 

- msa(A\textsuperscript{i+1}) is … 
  - either somewhere within A\textsuperscript{i}, which means msa(A\textsuperscript{i}) 
  - or is formed by rmax(A\textsuperscript{i})+A[i+1] 
- Thus, we only need to keep msa and rmax while scanning once through A
Algorithm & Complexity

- Analyses is simple
- Obviously: O(n)
- Asymptotically optimal
  - We only look a constant number of times at every element of A
  - But we need to look at least once on every element of A
  - Thus, we need at least O(n) operations – problem is Ω(n)
Example

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-2</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>-3</td>
<td>-4</td>
<td>2</td>
<td>-2</td>
</tr>
<tr>
<td>-2</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>-3</td>
<td>-4</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>-2</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>-3</td>
<td>-4</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>-2</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>-3</td>
<td>-4</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>-2</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>-3</td>
<td>-4</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>-2</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>-3</td>
<td>-4</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>-2</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>-3</td>
<td>-4</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>-2</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>-3</td>
<td>-4</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>-2</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>-3</td>
<td>-4</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>-2</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>-3</td>
<td>-4</td>
<td>2</td>
<td>6</td>
</tr>
</tbody>
</table>