# 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 the entire picture
  - Subtract this 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)

| | | | | | |
|---|---|---|---|---|---|
| 1 | 6 | 8 | 6 | 5 | 3 |
| 7 | 9 | 5 | 4 | 2 | 2 |
| 2 | 7 | 6 | 3 | 2 | 1 |
| 1 | 3 | 0 | 0 | 0 | 1 |
| 2 | 4 | 8 | 8 | 3 | 2 |
| 3 | 7 | 9 | 8 | 8 | 3 |

Avg. ~4

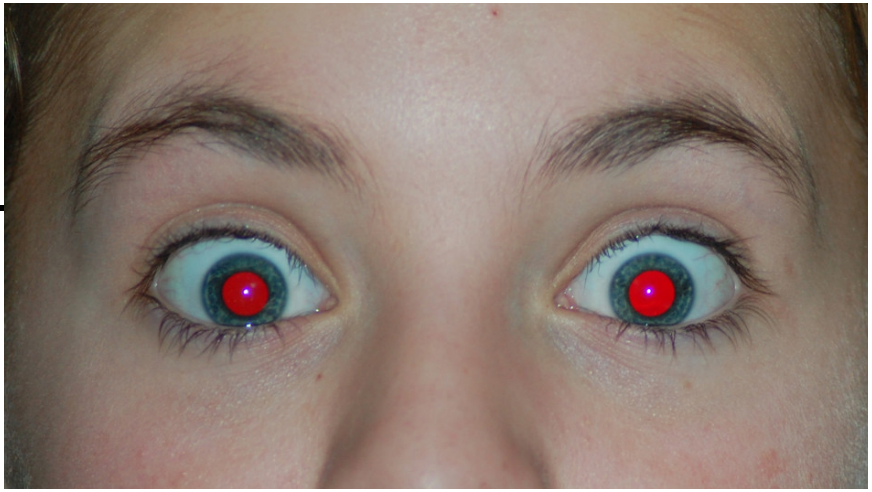
| | | | | | |
|---|---|---|---|---|---|
| -3 | 2 | 4 | 2 | 1 | -1 |
| 3 | 5 | 1 | 0 | -2 | -2 |
| -2 | 3 | 2 | -1 | -2 | -3 |
| -3 | -1 | -4 | -4 | -4 | -3 |
| -2 | 0 | 4 | 4 | -1 | -2 |
| -1 | 3 | 5 | 4 | 4 | -1 |

| | | | | | |
|---|---|---|---|---|---|
| -3 | 2 | 4 | 2 | 1 | -1 |
| 3 | 5 | 1 | 0 | -2 | -2 |
| -2 | 3 | 2 | -1 | -2 | -3 |
| -3 | -1 | -4 | -4 | -4 | -3 |
| -2 | 0 | 4 | 4 | -1 | -2 |
| -1 | 3 | 5 | 4 | 4 | -1 |

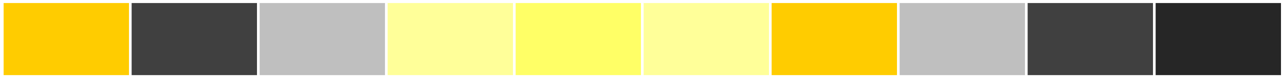
| | | | | | |
|---|---|---|---|---|---|
| -3 | 2 | 4 | 2 | 1 | -1 |
| 3 | 5 | 1 | 0 | -2 | -2 |
| -2 | 3 | 2 | -1 | -2 | -3 |
| -3 | -4 | -4 | -4 | -3 | -3 |
| -2 | 0 | 4 | 4 | -1 | -2 |
| -1 | 3 | 5 | 4 | 4 | -1 |

| | | | | | |
|---|---|---|---|---|---|
| -3 | 2 | 4 | 2 | 1 | -1 |
| 3 | 5 | 1 | 0 | -2 | -2 |
| -2 | 3 | 2 | -1 | -2 | -3 |
| -3 | -4 | -4 | -4 | -3 | -3 |
| -2 | 0 | 4 | 4 | -1 | -2 |
| -1 | 3 | 5 | 4 | 4 | -1 |

# Simpler Problem



- This is a bit complicated
  - Which shapes?
  - Shape should not be too big (sun is small compared to sky)
    - What if the sun is almost filling the picture?
  - Maximal sum of scores or maximal average score?
  - (see very last slide)
- For now, we look at a simpler problem: Max Subarray
  - Where is the sun?



# Max-Subarray Problem

- Definition (Max-Subarray Problem)
  *Assume an array A of integers. Find the highest sum-score s* of all subarrays A* of A, where the sum-score of an array A* is the sum of all its values.*
  - *If s* is negative, return 0*
- Remarks
  - Cells may have positive or negative values (or 0)
  - We only want the maximal sum, not the borders of A*
  - There might be multiple A*, but only one max sum-score
  - Length of the subarray A* is not fixed (like shape of spot)

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| -2 | 0 | 4 | 3 | 4 | -6 | -1 | 12 | -2 | 0 | 15 |

# A Greedy Solution

- Promising start point: Find maximal value in array A
- Greedy: Expand in both directions until sum decreases
- Complexity?

# A Greedy Solution

- Promising start point: Find maximal value in array A
- Greedy: 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
- Do we optimally solve our problem?

# A Greedy Solution

- Promising start point: Find maximal value in array A
- Greedy: Expand in both directions until sum decreases
- Complexity? (Let n=|A|)
  - O(n) together
- Do we optimally solve our problem?

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| -2 | 0 | 4 | 3 | 4 | -3 | -1 | 12 | 2 | -1 | 1 |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| -2 | 0 | 4 | 3 | 4 | -3 | -1 | 12 | 2 | -1 | 1 |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| -2 | 0 | 4 | 3 | 4 | -3 | -1 | 12 | 2 | -1 | 1 |

# A Greedy Solution

- Promising start point: Find maximal value in array A
- Greedy: Expand in both directions until sum decreases
- Complexity? (Let n=|A|)
  - O(n) together
- Do we optimally solve our problem?

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| -2 | 0 | 4 | 3 | 4 | -3 | -1 | 12 | 2 | -1 | 1 |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| -2 | 0 | 4 | 3 | 4 | -3 | -1 | 12 | 2 | -1 | 1 |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| -2 | 0 | 4 | 3 | 4 | -3 | -1 | 12 | 2 | -1 | 1 |

- First step may already be wrong

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| -2 | 0 | 4 | 3 | 4 | -6 | -6 | 10 | -6 | -1 | 1 |

# Content of this Lecture

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

# Naive Solution: Look at all Subarrays

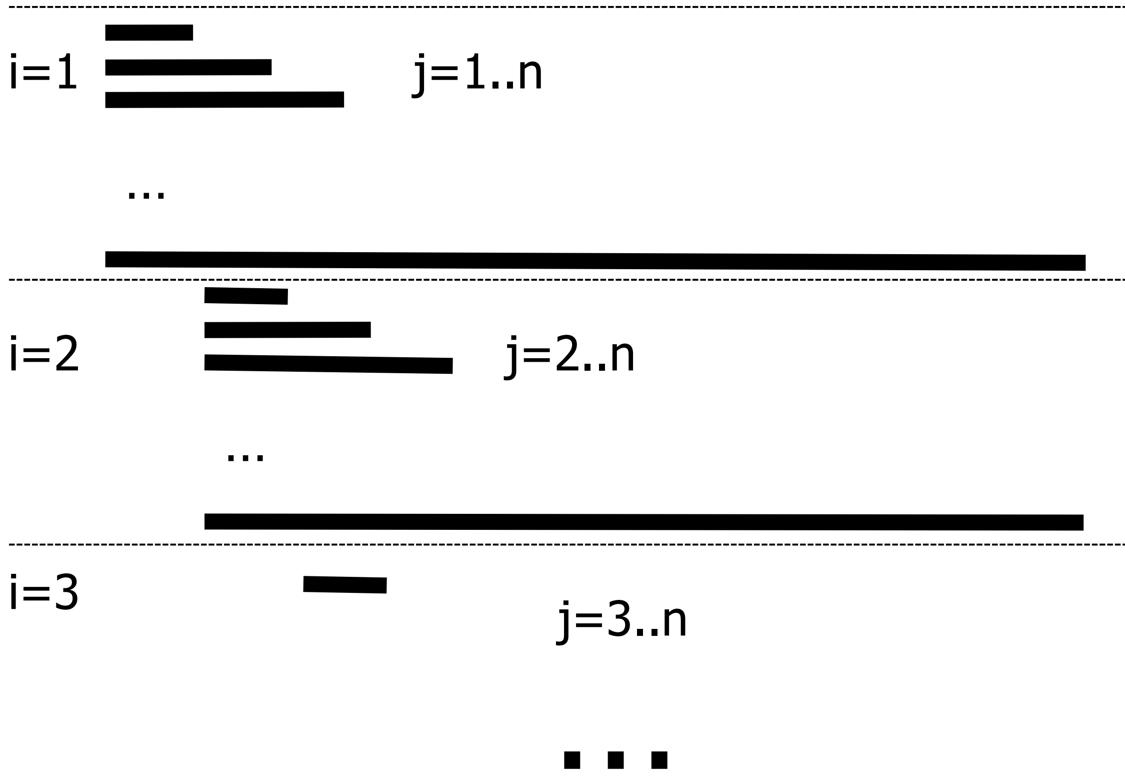
```
A: array_of_integer;
n := |A|;
m := 0;
for i := 1 … n do
  for j := i … n do
    s := 0;
    for k := i … j do
      s := s + A[k];
    end for;
    if s>m then
      m := s;
    end if;
  end for;
end for;
return m;
```

- i: Every start point of an array
- j: Every end point of an array
- k: Compute sum of values between start i and end j

# Illustration

```
A: array_of_integer;
n := |A|;
m := 0;
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;
```

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| -2 | 0 | 4 | 3 | 4 | -3 | -1 | 12 | 2 | -1 | 1 |



# Complexity

```
A: array_of_integer;
n := |A|;
m := 0;
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?

# Quiz

```
A: array_of_integer;
n := |A|;
m := 0;
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;
```

- Is this in $O(n)$?
- Is this in $O(n^2)$?
- Is this in $O(n^2 \log(n))$ but not $O(n^2)$?
- Is this in $O(n^3)$ but not $O(n^2)$?
- Is this in $O(n^4)$?

# Complexity

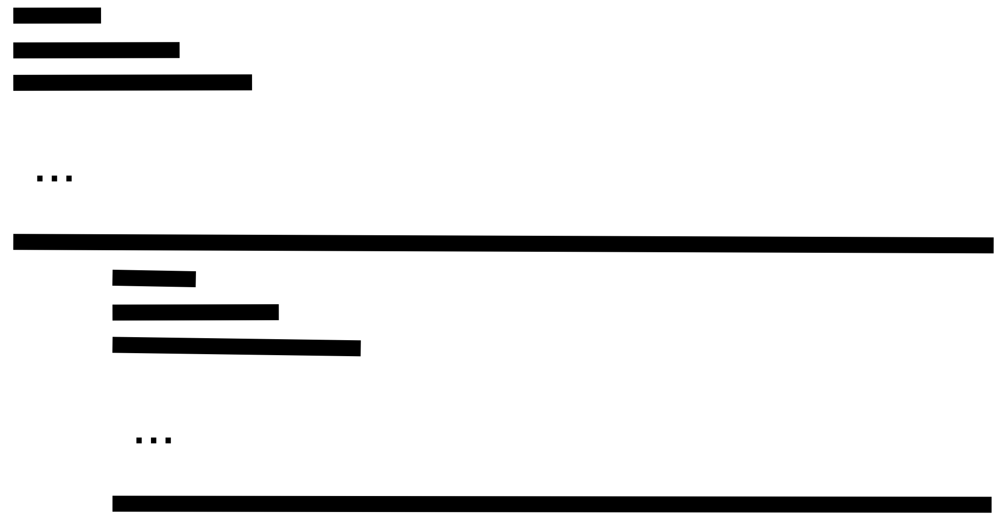
```
A: array_of_integer;
n := |A|;
m := 0;
for i := 1 … n do
  for j := i … n do
    s := 0;
    for k := i … j do
      s := s + A[k];
    end for;
    if s>m then
      m := s;
    end if;
  end for;
end for;
return m;
```

- i-loop: n times
- j-loop: n times (worst-case)
  - Together $\sim n^2/2$, which is in $O(n^2)$
- Inner loop: n times
- Together: $O(n^3)$
- But: We are summing up the same numbers again and again
- We perform redundant work
- More clever ways?

# Exhaustive Solution

- First sum: A[1]
- Second: A[1]+A[2]
- 3rd: A[1]+A[2]+A[3]
- 4th: …

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| -2 | 0 | 4 | 3 | 4 | -3 | -1 | 12 | 2 | -1 | 1 |




- For a given i, every next sum (k) is the previous sum plus the next cell (j)
- How can we reuse the previous sum?

# Exhaustive Solution, Improved

- Every next sum is the previous sum s plus the next cell A[j]
- Complexity: $O(n^2)$

```
A: array_of_integer;
n := |A|;
m := 0;
for i := 1 … n do
  s := 0;
  for j := i … n do
    s := s + A[j];
    if s>m then
      m := s;
    end if;
  end for;
end for;
return m;
```

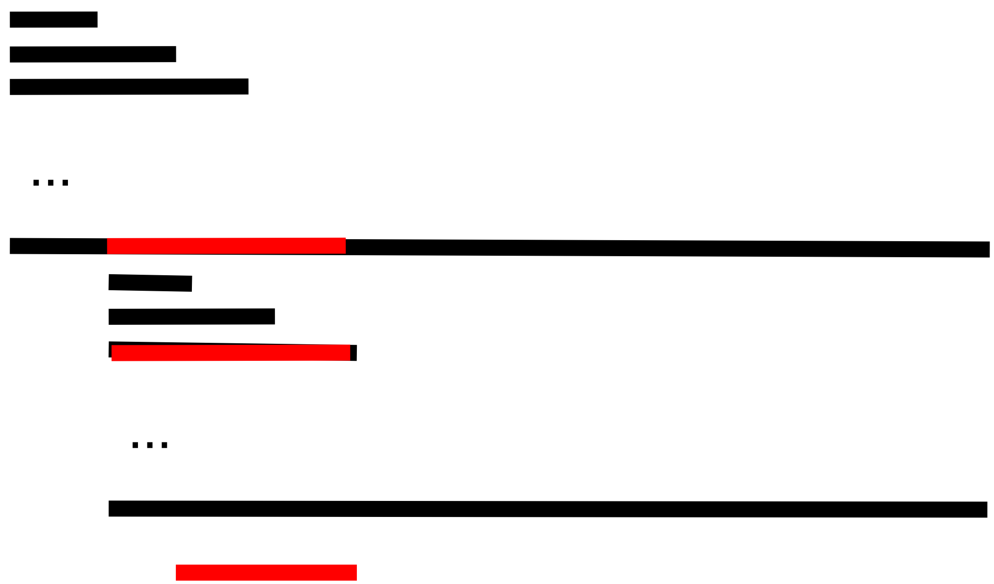
# Content of this Lecture

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

# Observation

- We optimized computation of sums in the j/k looks
- We still compute many sums multiple times – across i's

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| -2 | 0 | 4 | 3 | 4 | -3 | -1 | 12 | 2 | -1 | 1 |



# Divide and Conquer

- We can break up our problem into smaller ones by looking only at parts of the array
- A very common trick: Break into two equally-sized pieces
- One scheme: Assume $A=A_1|A_2$
  - With "|" meaning array concatenation and $|A_1|=|A_2|(+0/1)=|A|/2$
- 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's in $A_1/A_2$ that align 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

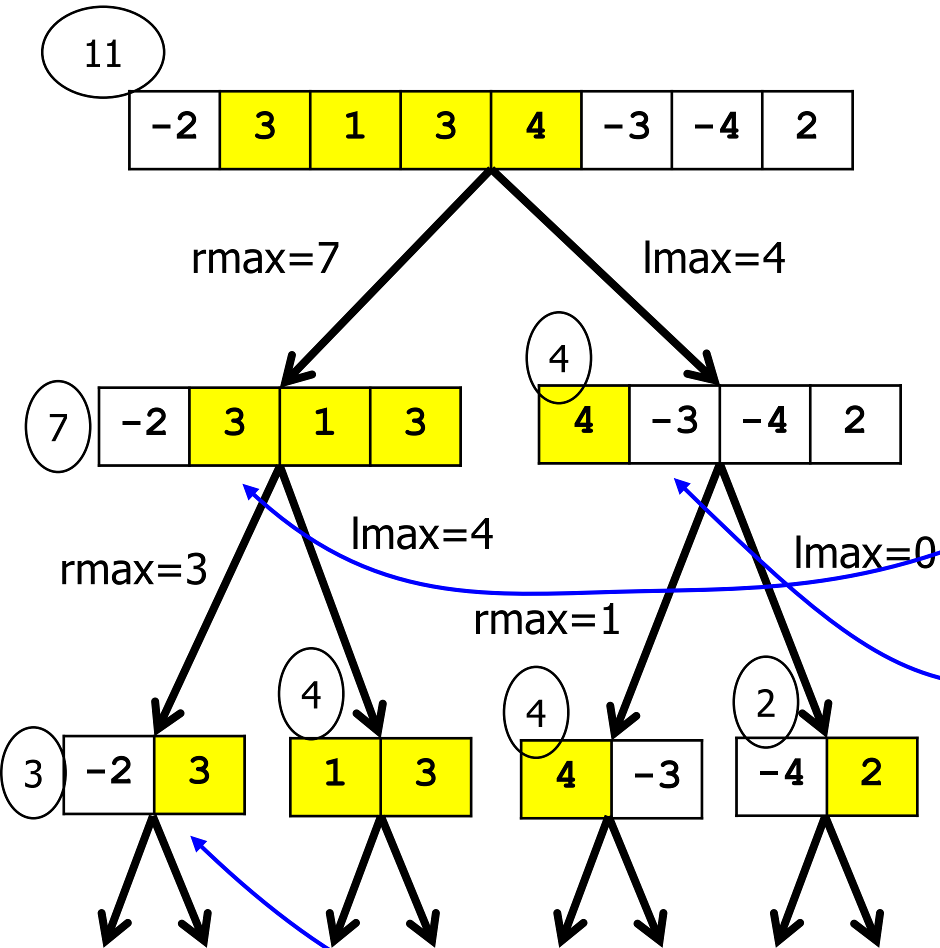
# Algorithm (for simplicity, assume $|A|=2^x$ for some x)

```
function msa (A: array_of_int) {
  n := |A|;
  if (n=1) then
    if A[1]>0 then
      return A[1]
    else
      return 0;
  end if;
  m := n/2;
  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_int){
  n := |A|;
  s := 0;
  m := 0;
  for i := n .. 1 do
    s := s + A[i];
    if s>m then
      m := s;
    end if;
  end for;
  return m;
}
```

# Example

m=(msa(A1),rmax(A1)+lmax(a2),msa(A2))




- Solution: max(7,7+4,4)
- Left array: max(3,3+4,4)
- Right array: max(4,1+0,2)
- Left-most: max(0,0+3,3)
- …

# Complexity

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

```
function rmax (A: array_of_int){
  n := |A|;
  s := 0;
  m := 0;
  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 lmax / rmax?
  - O(n)
- Function msa
  - 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
  - This yields: $T(n) \sim O(1)+O(n)+T(n')+T(n')$

```
function msa (A: array_of_int) {
  n := |A|;
  if (n=1) then
    if A[1]>0 then
      return A[1]
    else
      return 0;
  end if;
  m := n/2;    # …
  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;
}
```

# Complexity

- This time it is not so easy …
- Complexity of lmax / rmax?
  - O(n)
- Function msa
  - 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
  - This yields: $T(n) \sim O(1)+O(n)+T(n')+T(n')$
    $= O(1)+O(n)+T(n/2)+T(n/2)$
    $= O(n)+2*T(n/2)$

```
function msa (A: array_of_int) {
  n := |A|;
  if (n=1) then
    if A[1]>0 then
      return A[1]
    else
      return 0;
  end if;
  m := n/2;    # …
  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;
}
```

# Complexity

```
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 $c_1$, $c_2$
- $T(n) = 2*T(n/2)+c_1*n$
- Further: $T(1) = c_2$

# Complexity

```
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 $c_1$, $c_2$
- $T(n) = 2*T(n/2)+c_1*n$
- Further: $T(1) = c_2$
- Iterative substitution:

$$T(n)= 2*T(n/2)+c_1 n =$$
$$= 2(2T(n/4)+c_1 n/2)+c_1 n \quad = 4T(n/4)+2c_1 n=$$
$$= 4(2T(n/8)+c_1 n/4)+2c_1 n = \underbrace{8T(n/8)}_{2^{\log(n)}*c_2}+\underbrace{3c_1 n}_{\log(n)*c_1 n} = \ldots$$

$$2^{\log(n)}*c_2 + \log(n)*c_1 n =$$
$$n*c_2+\log(n)*c_1 n \;\; = O(n*\log(n))$$

# Same Problem, Different Algorithms

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

- 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 solutions 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]?

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| -2 | 0 | 4 | 3 | 4 | -3 | -1 | 6 |

# Let‘s Think again – More Carefully

- Let‘s use another strategy for dividing the problem
- Let‘s look at the solutions 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]?

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| -2 | 0 | 4 | 3 | 4 | -3 | -1 | 6 |

- msa($A^{i+1}$) is …
  - either somewhere within $A^i$, which means the same as msa($A^i$)
  - or includes A[i+1], i.e., it is rmax($A^i$)+A[i+1]
- Idea: Keep msa and rmax while scanning once through A

# Algorithm & Complexity

```
A: array_of_integer;
rmax:= 0;
m := 0;
for i:= 1 to n do
  rmax := max( 0, rmax+A[i]);
  m := max( rmax, m);
end for;
return m;
```

- 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 at every element of A to solve msa
  - Thus, the problem is $\Omega(n)$
- Example of dynamic programming: Build larger solutions from smaller ones

# Example

| | | | | | | | | | rmax | m |
|---|---|---|---|---|---|---|---|---|---|---|
| -2 | 3 | 1 | 3 | 4 | -3 | -4 | 2 | | 0 | 0 |
| -2 | 3 | 1 | 3 | 4 | -3 | -4 | 2 | | 3 | 3 |
| -2 | 3 | 1 | 3 | 4 | -3 | -4 | 2 | | 4 | 4 |
| -2 | 3 | 1 | 3 | 4 | -3 | -4 | 2 | | 7 | 7 |
| -2 | 3 | 1 | 3 | 4 | -3 | -4 | 2 | | 11 | 11 |
| -2 | 3 | 1 | 3 | 4 | -3 | -4 | 2 | | 8 | 11 |
| -2 | 3 | 1 | 3 | 4 | -3 | -4 | 2 | | 4 | 11 |
| -2 | 3 | 1 | 3 | 4 | -3 | -4 | 2 | | 6 | 11 |

# Why Better?

```
A: array_of_integer;
rmax:= 0;
m := 0;
for i:= 1 to n do
  rmax := max( 0, A[i],
                 rmax+A[i]);
  m := max( rmax, m);
end for;
```

```
function msa (A: array_of_integer) {
  n := |A|;
  …
  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;
}
```

- Algorithm to the right: Breaks in the middle
  - log(n) recursive calls
  - But we need to compute new rmax and lmax values all the time
- Algorithm to the left: Break iteratively
  - n extensions
  - But we can compute the next rmax from the previous rmax

# Optimization Problems

- Optimization – find the best among all possible solutions
  - Shortest path, best schedule, best query plan, best superstring, …
- Issues
  - Find solutions: Simple for msa, but sometimes hard
  - Score solutions: Simple for msa, but sometimes hard
  - Search space pruning: Do we need to look at all possible solutions?
    - msa: No need to break array at position 3/7
    - Be aware: Sometimes all possible "divides" are necessary
- Typical pattern
  - Enumerate solutions in a systematic manner
  - Often generates a tree of partial and finally complete solutions
  - Prune parts of the search space where no optimal solution can be
  - If possible, stop early

# Fundamental Types of Algorithms

- Greedy: Find some promising start point and expand aggressively until a complete solution is found
  - Fast, but usually doesn't find the optimal solution
- Exhaustive: Test all solutions and find the one that is best
  - No pruning - slow, sometimes the only choice
- Divide & Conquer: Break problem into smaller ones until these can be solved "directly"; construct solutions for bigger problems from these small solutions
- Dynamic programming
  - Special case of d&c involving tables of intermediate results
- Backtracking
  - Special case of d&c: Move from smaller solutions to larger, but allow undoing previous moves

# Types of Algorithms

- For the max subarray problem
  - Greedy: O(n), but wrong
  - Exhaustive: $O(n^3)$
    - With pruning $O(n^2)$
  - Divide & Conquer: O(n*log(n))
  - Other Divide & Conquer: O(n)
- Notes
  - Usually there are different greedy, exhaustive, … solutions

# Exemplary Questions

- Give an optimal algorithm for the max-subarray problem and prove its optimality
- Assume the max-subarray problem with the additional restriction that the length of sub-array must be short-or-equal a constant k. Give a linear algorithm solving this problem.
- Give an algorithm for the max-subarray problem in 2D, where |A| is quadratic and the subarray must be a square. Analyze its worst-case complexity.
  - Hint: For improvements, store intermediate results