Algorithms and Data Structures

One Problem, Four Algorithms

Patrick Schäfer
(Folienbasis Ulf Leser)

Content of this Lecture

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

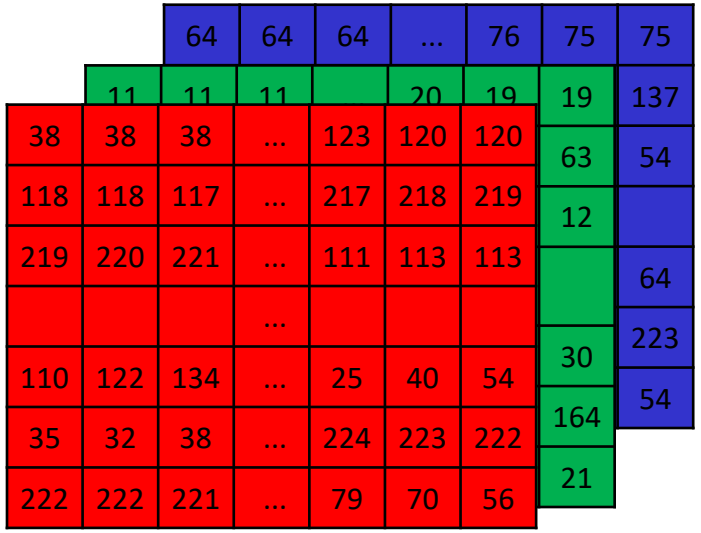
Where is the Sun?

Source: http://www.layoutsparks.com

Pixel RGB Representation

Image

2048×1324 Pixels



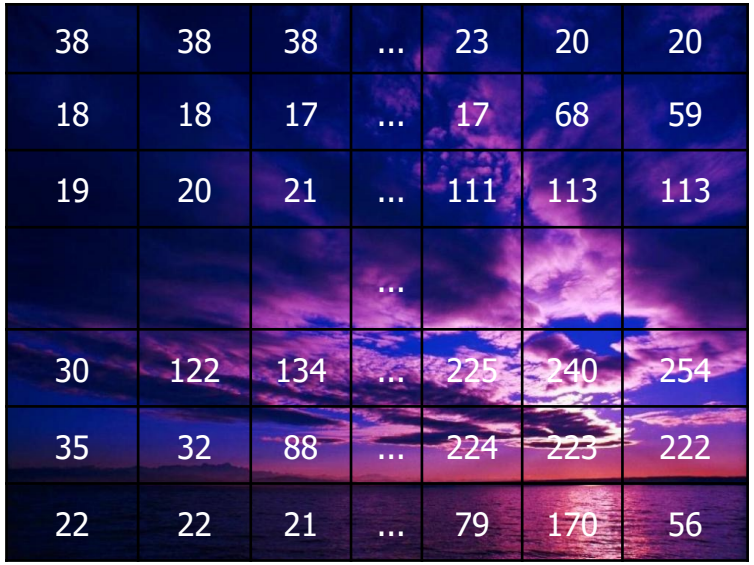
Internal: RGB representation

Red / Green / Blue Channels

3 x 2048×1324 Pixels

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



Brightness Scores

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

(subtract)

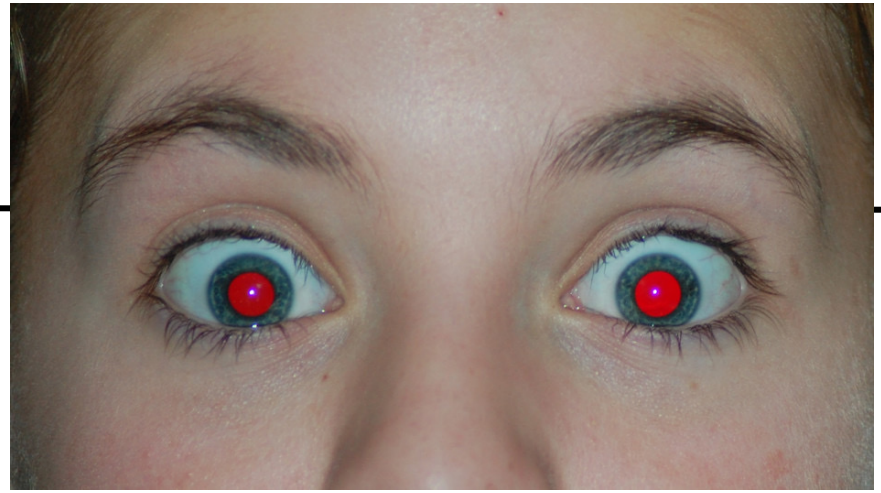
-3	2	4	2	1	-1
3	5	1	0	-2	-2
-2	3	2	-1	-2	-3
-3	-1	-4	-4	-4	-3
-2	0	4	4	-1	-2
-1	3	5	4	4	-1

-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?

- For now, we'll 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 offsets of A^*
  - Multiple subarrays A^* may share the same max sum-score
  - Length of the subarray A^* is not fixed (like shape of spot)

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

Max-Subarray – Example

What is the highest sum-score s^* for A?

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

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

$s^* = 11$ $s^* = 12$ $s^* = 15$

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

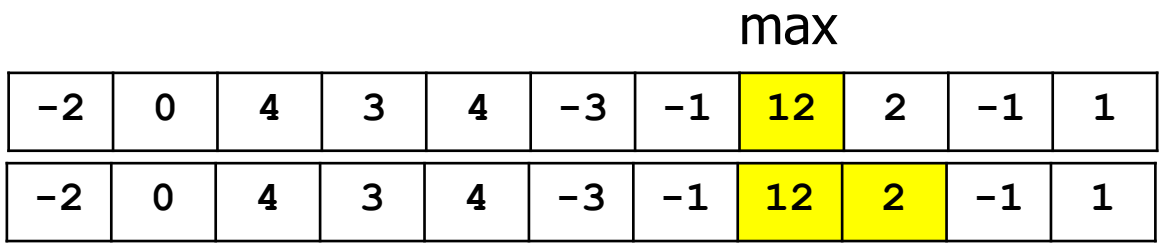
$s^* = 25$

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

$s^* = 29$

A Greedy Solution

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



A Greedy Solution

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

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

- Complexity? (Let $n = |A|$)
  - O(n) to find maximal value
  - O(n) expansion steps in worst case
  - O(n) together

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?

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
Greedy	-2	0	4	3	4	-3	-1	12	2	-1	1
Optimal	-2	0	4	3	4	-3	-1	12	2	-1	1

$s^* = 21$

- Start point may already be wrong

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

$s^* = 11$

Content of this Lecture

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

Naive Solution: Look at all Subarrays

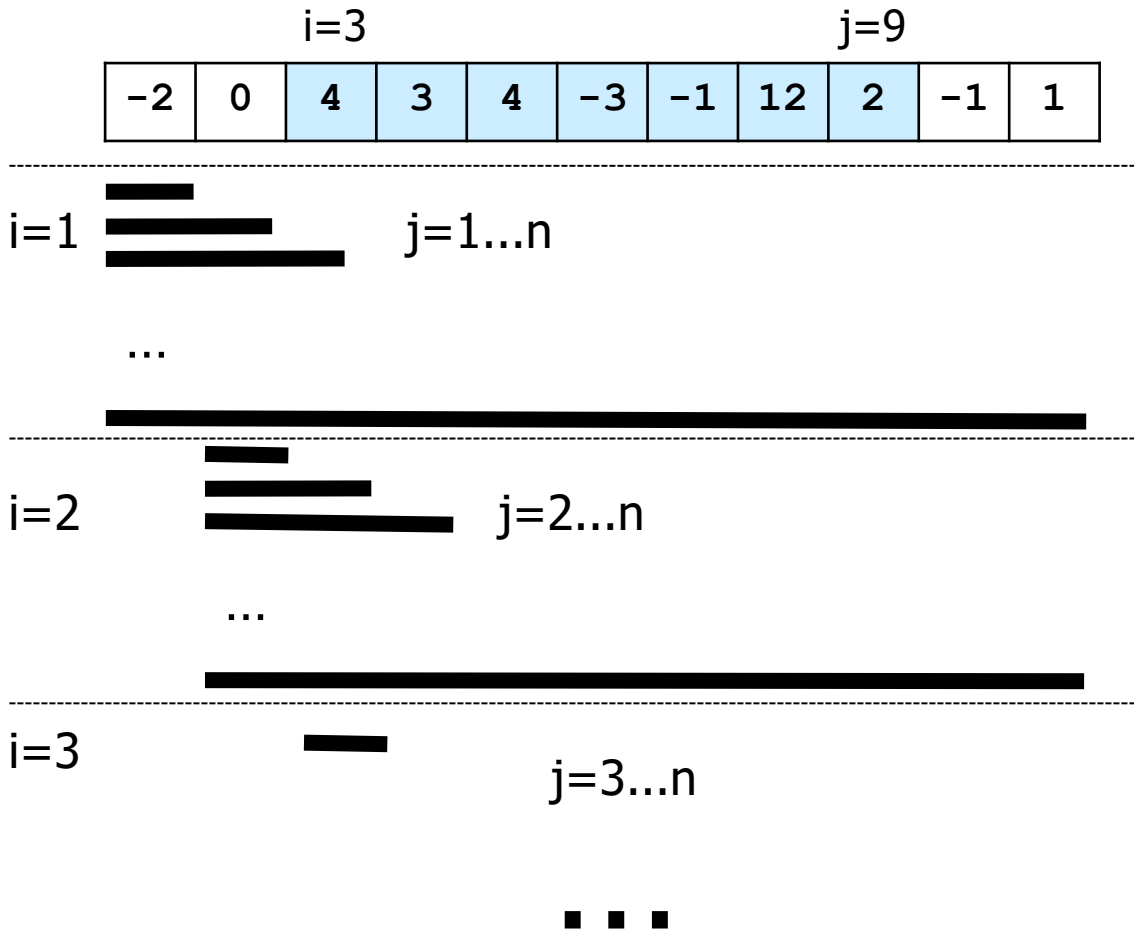
```
A: array_of_integer;
n := |A|;
m := 0;
for i := 1 … n do
  for j := i … n do
    s := 0;
    for k := i … j do
      s := s + A[k];
    end for;
    if s>m then
      m := s;
    end if;
  end for;
end for;
return m;
```

- m: highest sum-score s^*
- 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:

$$\sum_{k=i}^{j} A[k]$$

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;
```



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?
- i-loop: n times
- j-loop: runs 1 up to n times
  - Together: $\sum_{j=1}^{n} j = \frac{n(n+1)}{2}$ which is in $O(n^2)$
- Innermost k-loop: up to n times

- Together: $O(n^3)$

Complexity

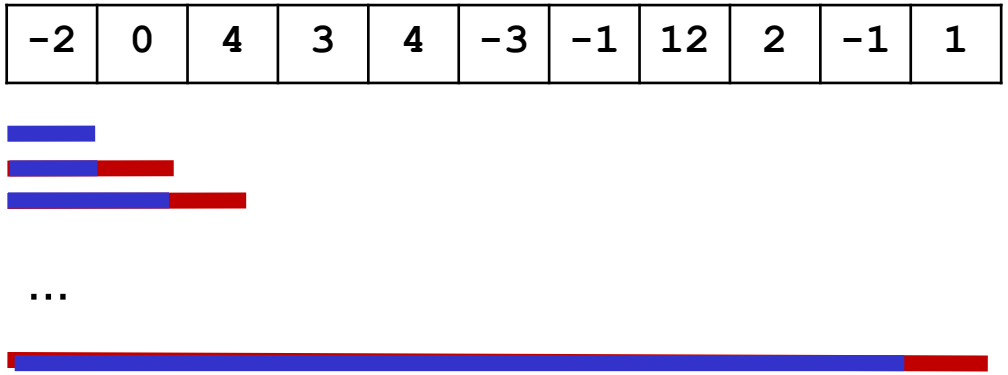
```
A: array_of_integer;
n := |A|;
m := 0;
for i := 1 … n do
  for j := i … n do
    s := 0;
    for k := i … j do
      s := s + A[k];
    end for;
    if s>m then
      m := s;
    end if;
  end for;
end for;
return m;
```

- 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]
- 2nd: A[1]+A[2]
- 3rd: A[1]+A[2]+A[3]
- 4th: …



- Every next sum is computed from the previous sum s plus the next cell A[j]

- We can reuse the previous sum

Exhaustive Solution, Improved

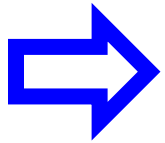
- First sum: s=A[1]
- 2nd: s=A[1]+A[2] = s + A[2]
- 3rd: s=A[1]+A[2]+A[3] = s + A[3]

- j-th sum: s= s + A[j]

- Every next sum s is computed from the previous sum s plus the next cell A[j]

Exhaustive Solution, Improved

```
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;
```



j-th sum: s=s+A[j]

```
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;
```

Exhaustive Solution, Improved

- Every next sum s is the previous sum s plus the next cell A[j]
- Complexity:
  loop over j is repeated $\sum_{j=1}^{n} j$ times
  which is in $O(n^2)$

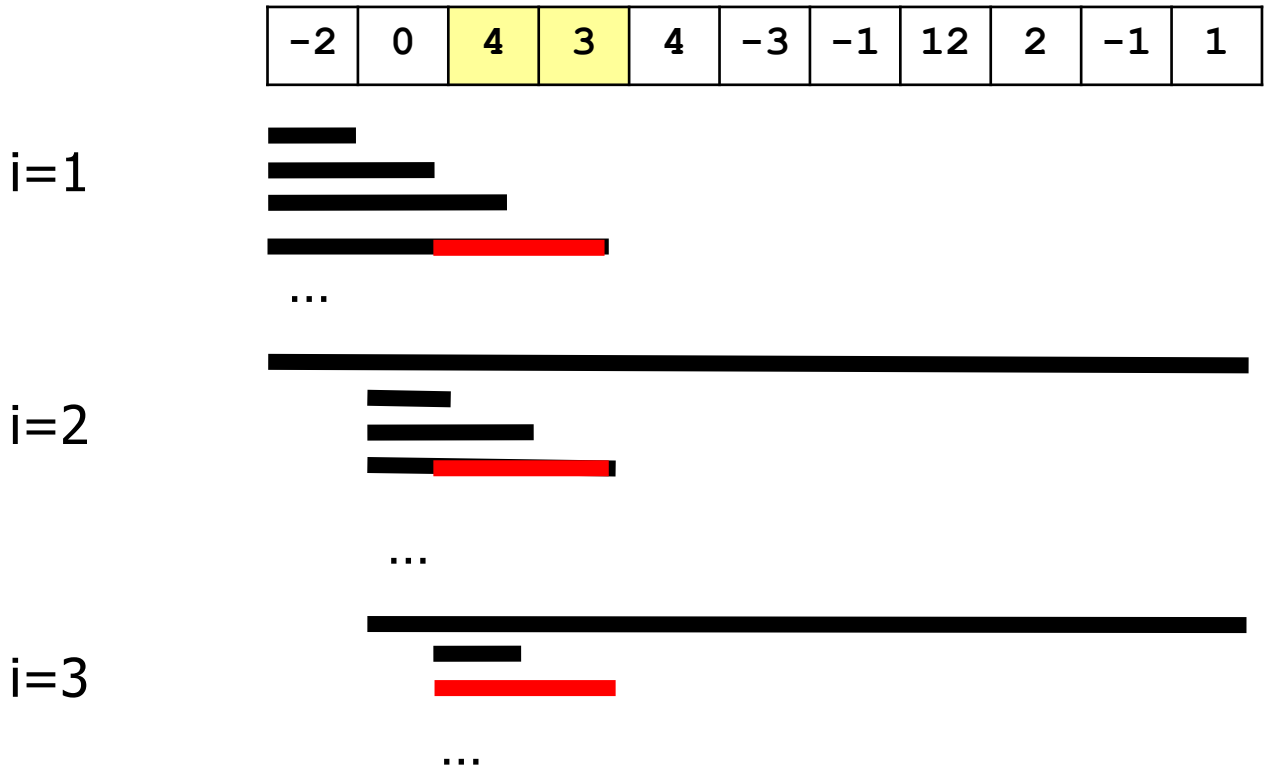
```
A: array_of_integer;
n := |A|;
m := 0;
for i := 1 … n do
  s := 0;
  for j := i … n do
    s := s + A[j];
    if s>m then
      m := s;
    end if;
  end for;
end for;
return m;
```

Content of this Lecture

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

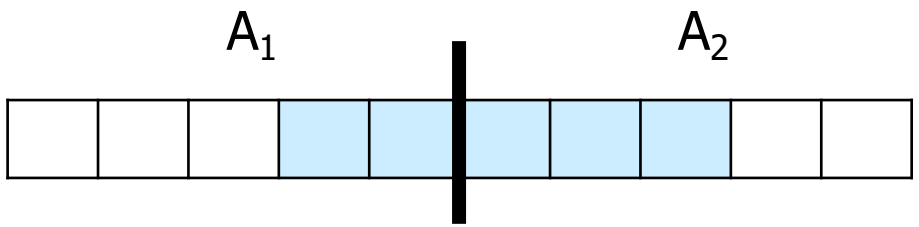
Observation

- We optimized the computation of sums in the j and k loops
- We still compute many sums multiple times – across the i loop



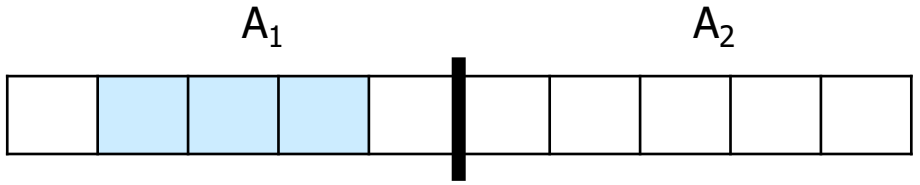
Divide and Conquer

- We can break up 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 $|A_1|=|A_2|(+0/1)=|A|/2$

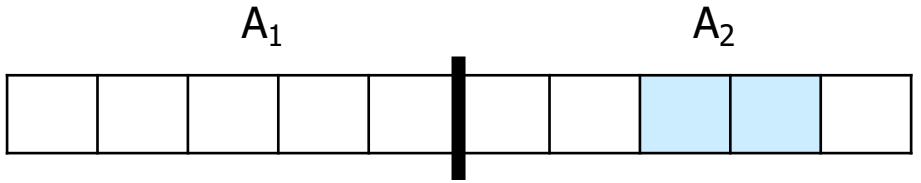


Divide and Conquer

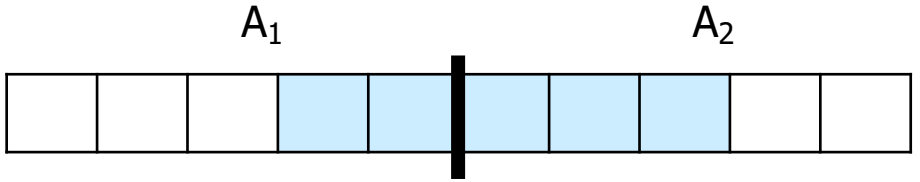
- The max-subarray (msa) of A …
  1. either lies in A_1 – can be found by solving msa(A_1)



  2. or in A_2 – can be found by solving msa(A_2)



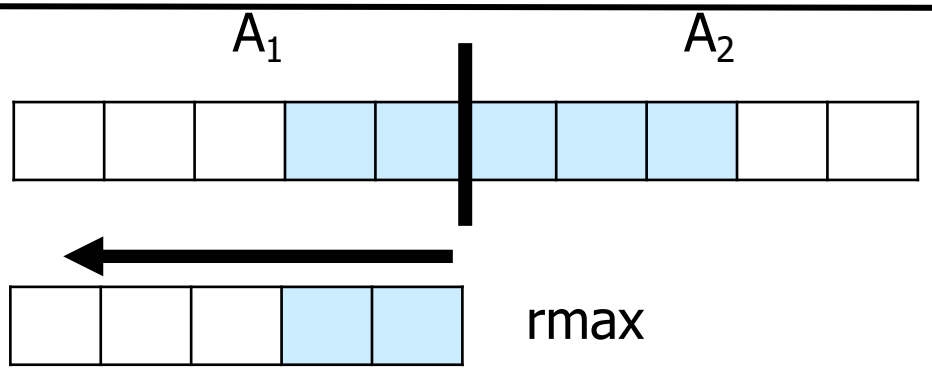
  3. or partly in A_1 and partly in A_2
     - Can be solved by summing-up the msa's in A_1/A_2 that align with the right/left end of A_1/A_2



- 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)

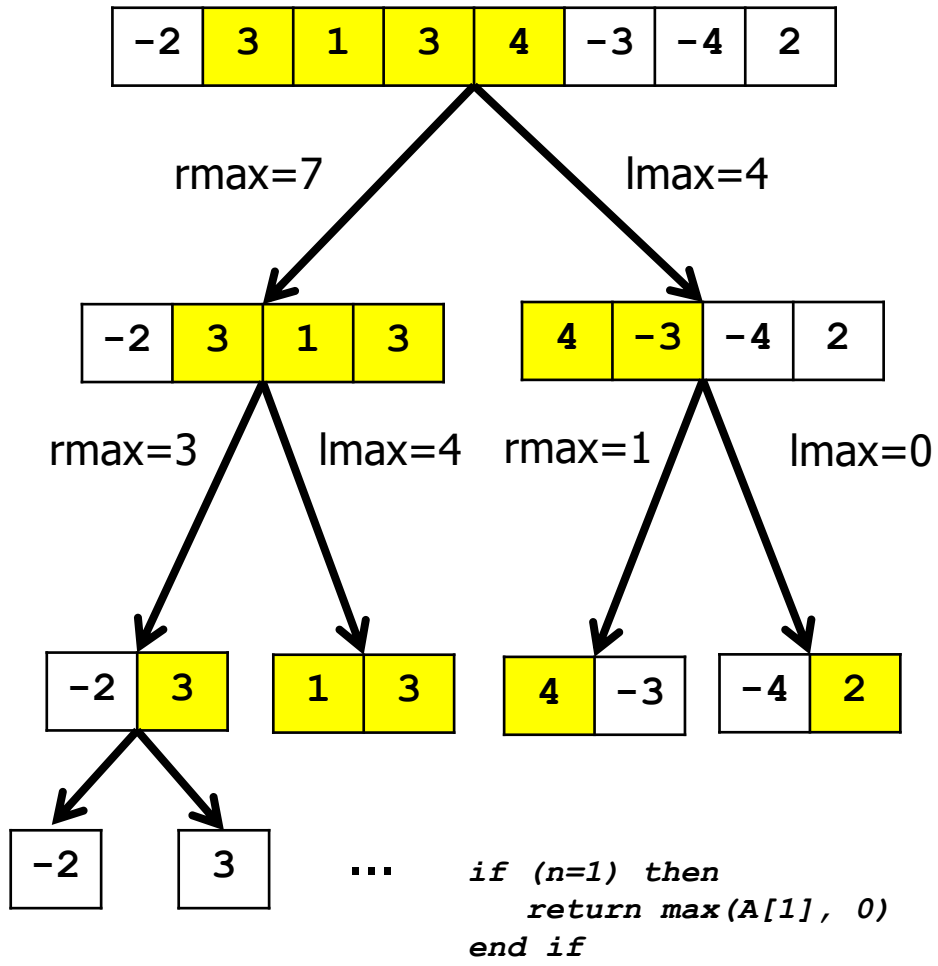
```
function msa (A: array_of_int) {
  n := |A|;
  if (n=1) then
    return max(A[1], 0)
  end if;

  m := n/2;

  A1 := A[1…m];
  A2 := A[m+1…n];

  l1 := rmax(A1);
  l2 := lmax(A2);
  m := max(msa(A1),     (1)
           l1+l2,       (3)
           msa(A2));    (2)
  return m;
}
```




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

Example

```
m := max(msa(A1), rmax+lmax, msa(A2));
```



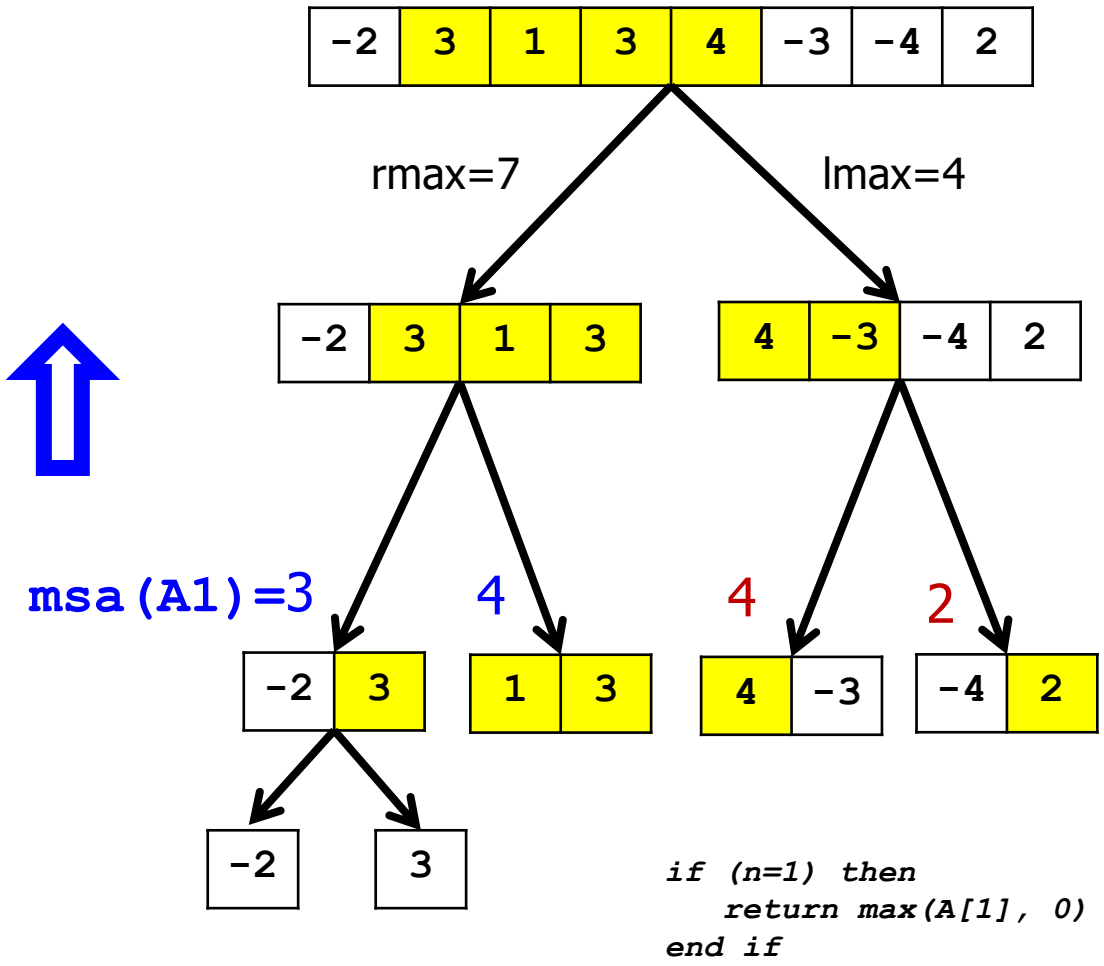

```
m := max(msa(A1), 7+4, msa(A2));
```

```
m := max(msa(A1), 3+4, msa(A2));
m := max(msa(A1), 1+0, msa(A2));
```

```
m := max(msa(A1), 0+3, msa(A2));
m := max(msa(A1), 1+3, msa(A2));
m := max(msa(A1), 4+0, msa(A2));
m := max(msa(A1), 0+2, msa(A2));
```

Example

```
m := max(msa(A1), rmax+lmax, msa(A2));
```



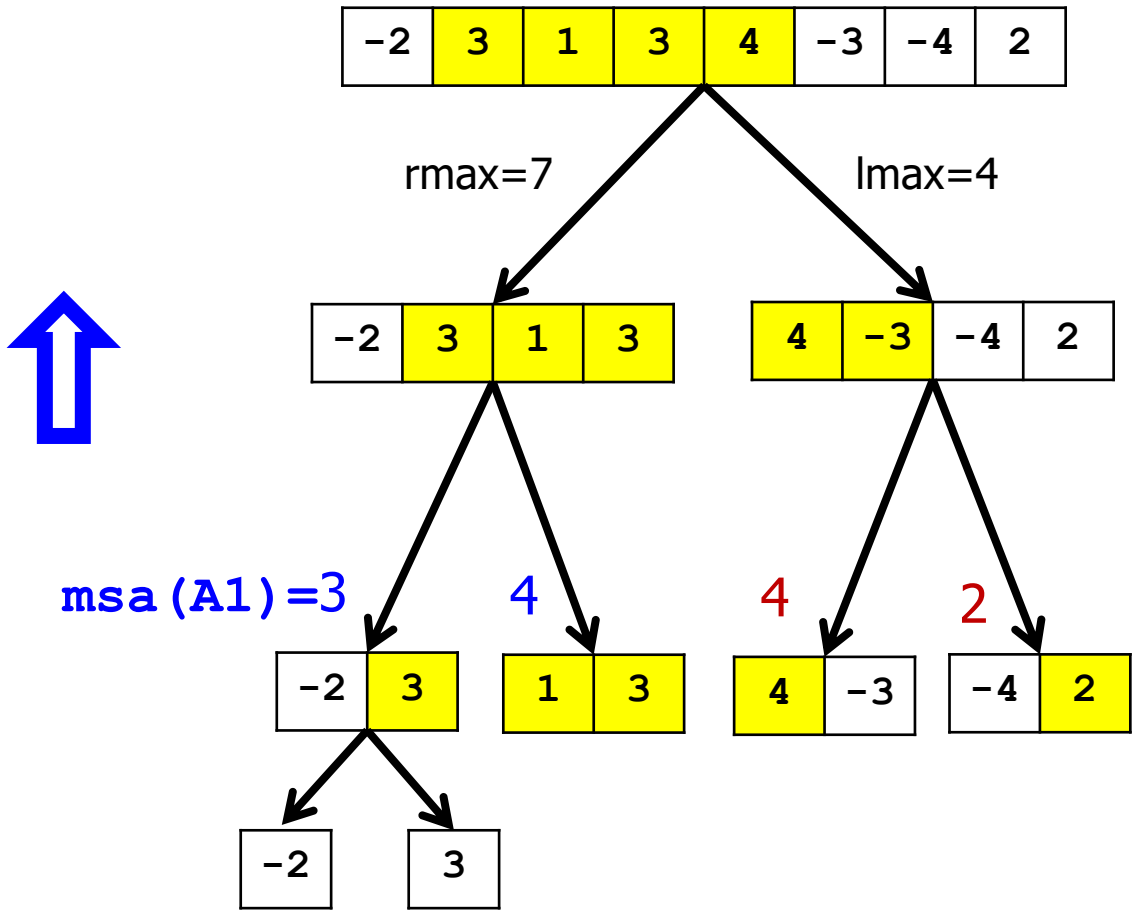

```
if (n=1) then
   return max(A[1], 0)
end if
```

```
m := max(msa(A1), 7, msa(A2));
m := max(msa(A1), 1, msa(A2));
```

```
m := max(0, 3, 3) = 3
m := max(1, 4, 3) = 4
m := max(4, 4, -3) = 4
m := max(-4, 2, 2) = 2
```

Example

```
m := max(msa(A1), rmax+lmax, msa(A2));
```



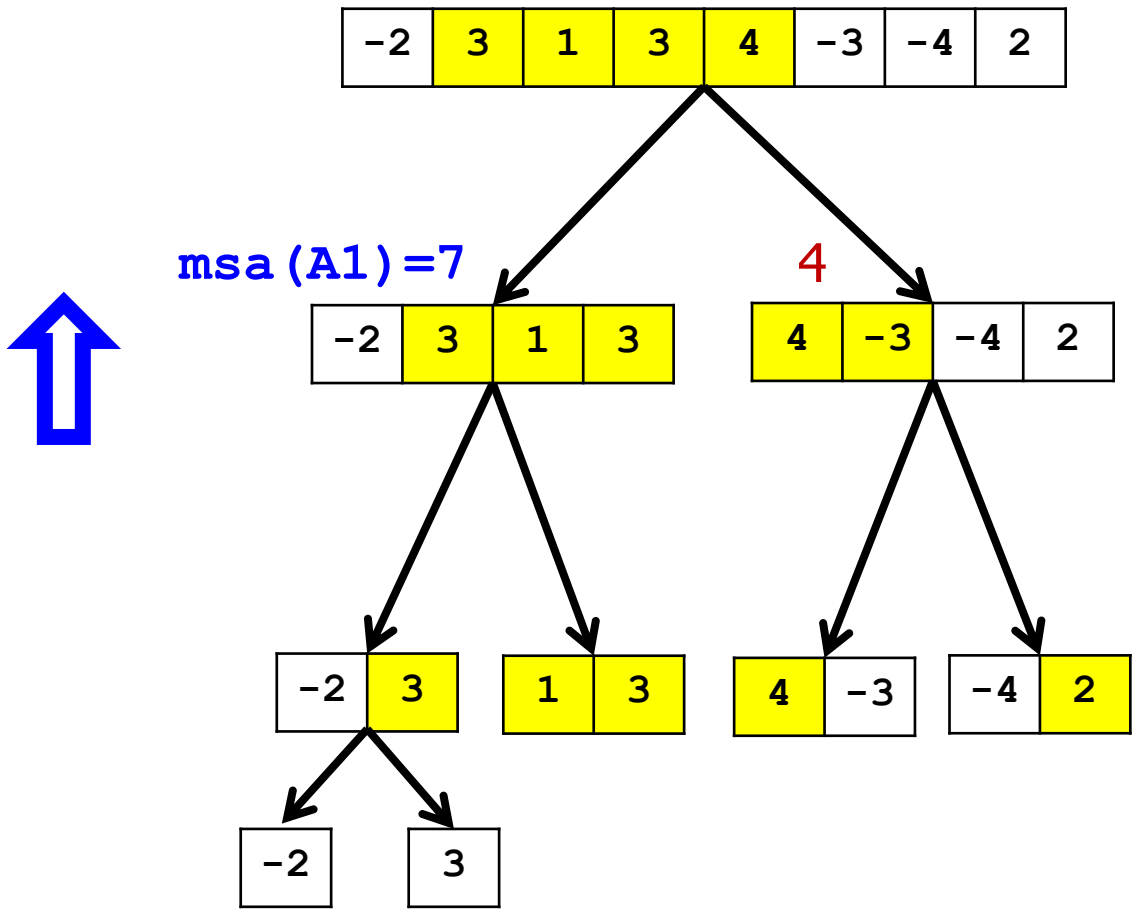

```
m := max(msa(A1), 11, msa(A2));
```

```
m := max(3, 7, 4) = 7
m := max(4, 1, 2) = 4
```

```
m := max(0, 3, 3) = 3
m := max(1, 4, 3) = 4
m := max(4, 4, -3) = 4
m := max(-4, 2, 2) = 2
```

Example

```
m := max(msa(A1), rmax+lmax, msa(A2));
```



Solution :

```
m := max(7, 11, 4) = 11
```

```
m := max(3, 7, 4) = 7
m := max(4, 1, 2) = 4
```

Complexity

- This time analysis is not so easy …
- Complexity of lmax / rmax?
  - O(n)

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

Complexity

- This time analysis 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

```
function msa (A: array_of_int) {
  n := |A|;
  if (n=1) then
    return max(A[1], 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 analysis 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: $T(n) \sim O(1) + O(n) + 2 \cdot T\left(\frac{n}{2}\right) = 2 \cdot T\left(\frac{n}{2}\right) + O(n)$

```
function msa (A: array_of_int) {                          
  n := |A|;                                  O(1)
  if (n=1) then                              O(1)
    return max(A[1], 0)                      O(1)
  end if;
  m := n/2;     # …                          O(1)
  A1 := A[1…m];                              O(1)
  A2 := A[m+1…n];                            O(1)
  l1 := rmax(A1);                            O(n/2) = O(n)
  l2 := lmax(A2);                            O(n/2) = O(n)
  m := max(msa(A1),l1+l2,msa(A2));           2 · T(n/2)
  return m;
}
```

Complexity

- For constants c_1, c_2:
- $T(1) = c_2$
- $T(n) = 2 \cdot T(n/2) + c_1 \cdot n$

```
function msa (A: array_of_integer) {
  n := |A|;
  if (n=1) then
    return max(A[1], 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;
}
```

Complexity

```
function msa (A: array_of_integer) {
  n := |A|;
  if (n=1) then
    return max(A[1], 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
- $T(1) = c_2$
- $T(n) = 2 \cdot T(n/2) + c_1 \cdot n$

- Iterative substitution:

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + c_1 n$$

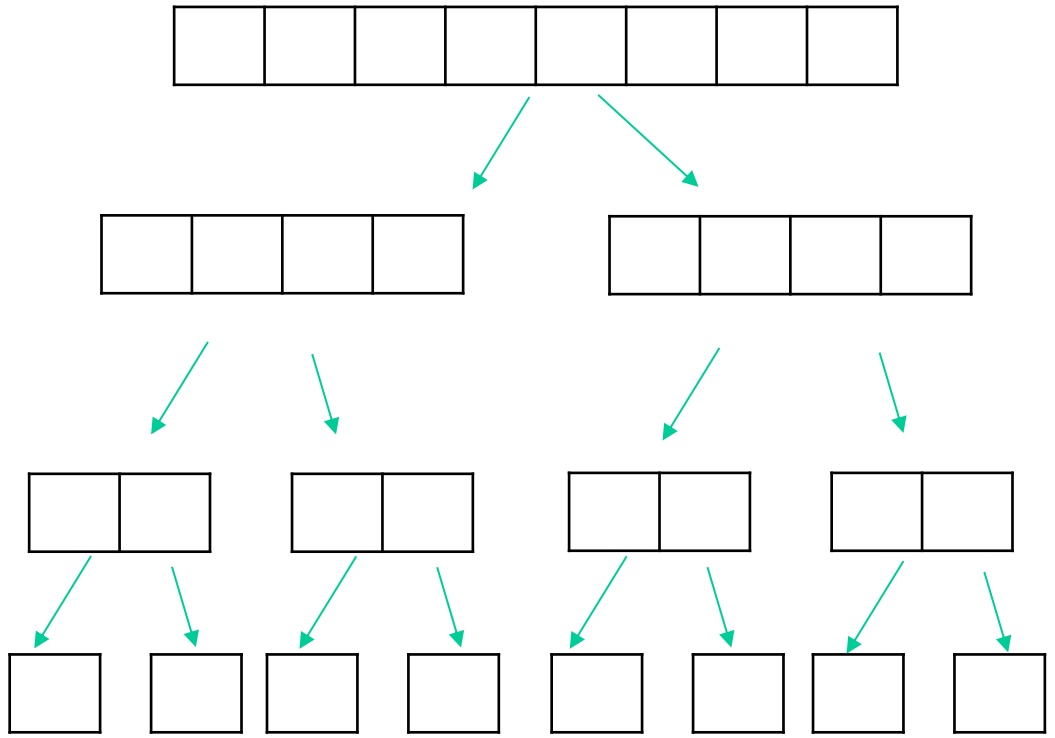
$$= 2\left(2 \cdot T\left(\frac{n}{4}\right) + c_1 \frac{n}{2}\right) + c_1 n = 4 \cdot T\left(\frac{n}{4}\right) + 2c_1 n$$

$$= 4\left(2 \cdot T\left(\frac{n}{8}\right) + c_1 \frac{n}{4}\right) + 2c_1 n = 8 \cdot T\left(\frac{n}{8}\right) + 3c_1 n$$

$$= \cdots$$

Iterative Solution – Tree Illustration

Level k	Subproblems
0	$T(n)$
1	$2^1 \cdot T\left(\frac{n}{2^1}\right)$
2	$2^2 \cdot T\left(\frac{n}{2^2}\right)$
3	$2^3 \cdot T\left(\frac{n}{2^3}\right)$



$2^k = n$
$\Leftrightarrow \mathrm{k} = \log_2(n)$

$2^{\log_2(n)} \cdot T(1)$
$= n \cdot T(1)$

Complexity

```
function msa (A: array_of_integer) {
  n := |A|;
  if (n=1) then
    return max(A[1], 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
- $T(1) = c_2$
- $T(n) = 2 \cdot T(n/2) + c_1 \cdot n$

- Iterative substitution:

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + c_1 n = \cdots = \underbrace{2^3 \cdot T\left(\frac{n}{2^3}\right)}_{} + \underbrace{3 \cdot c_1 n}_{} = \cdots = n\,T(1) + k\,c_1 n$$

k: levels

$$= n \cdot c_2 + \log_2(n) \cdot c_1 n$$
$$\in O(n \cdot \log(n))$$

Same Problem, Different Algorithms

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

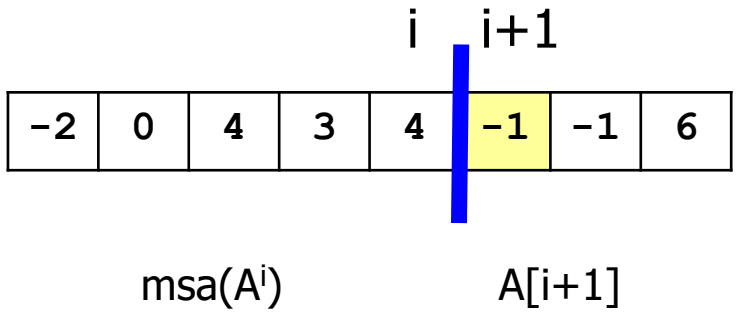
- The problem: $O(n)$

Content of this Lecture

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

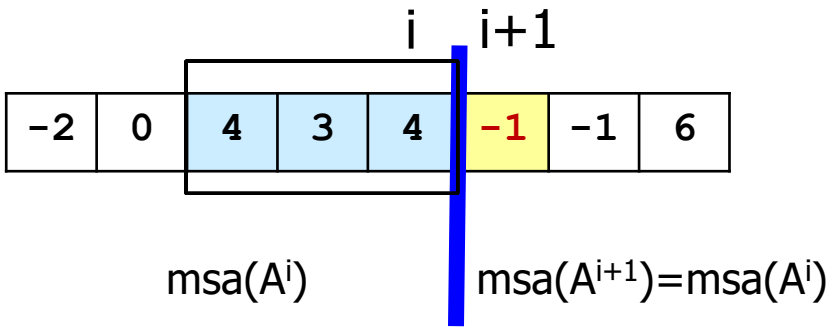
Let's Think again – More Carefully

- 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]?



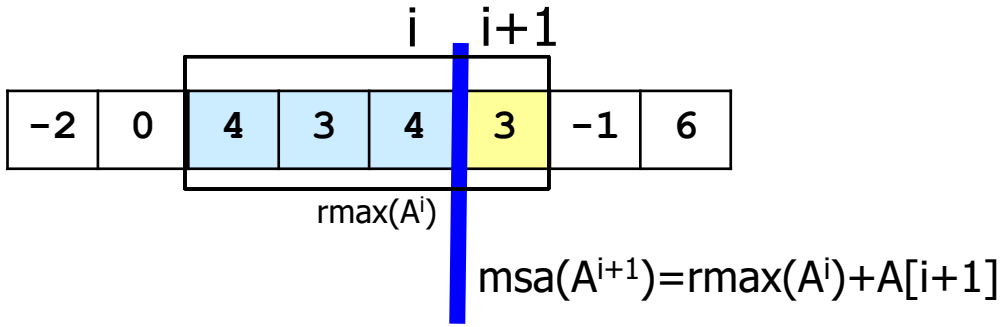
Let's Think again – More Carefully

- msa(A^{i+1}) is ...
  1. either somewhere within A^i, which means the same as msa(A^i)



Let's Think again – More Carefully

- msa(A^{i+1}) is …
  1. either somewhere within A^i, which means the same as msa(A^i)
  2. 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

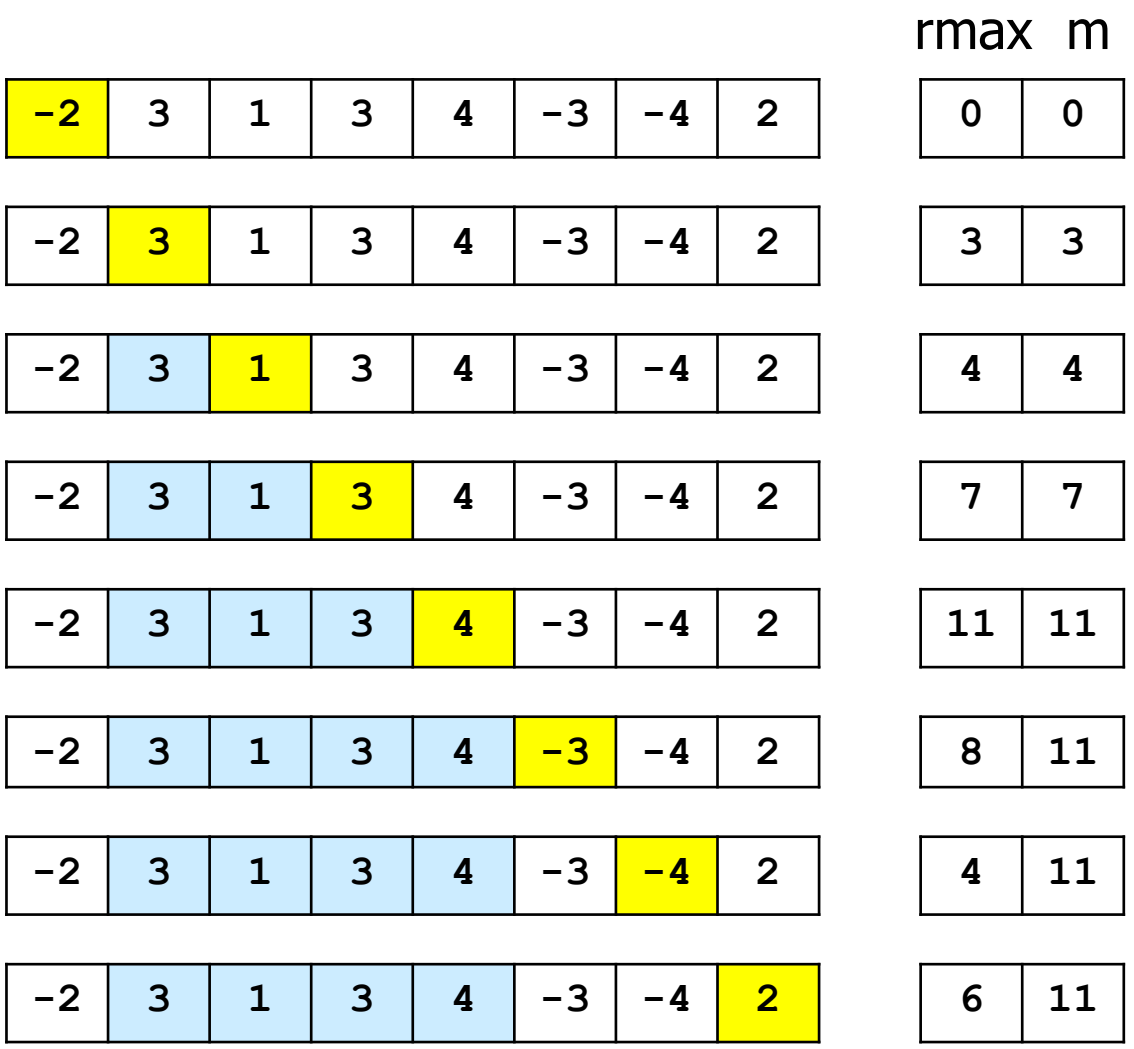
Algorithm & Complexity

```
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
  - Thus, the problem is also $\Omega(n)$
- Example of dynamic programming: Build larger solutions from smaller ones

Example

```
rmax := max(0, rmax+A[i]);
m := max(rmax, m);
```



Optimization Problems

- Optimization – find the best among all possible solutions
- Issues
  - Find solutions: Simple here, but sometimes hard
  - Score solutions: Simple here, but sometimes hard
  - Search space pruning: Do we need to look at all possible solutions?

- 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

- Different fundamental patterns exist
  - 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
    - Sometimes the only choice if optimality is asked for
  - Divide & Conquer: Break problem into smaller ones until these become so easy that they can be solved "directly"; construct solutions for bigger problems from these small solutions
  - Dynamic programming
  - Backtracking
  - ...

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 multiple 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