



# 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



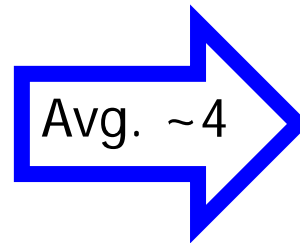
# Size of the Spot not Pre-Determined

---



# 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



-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

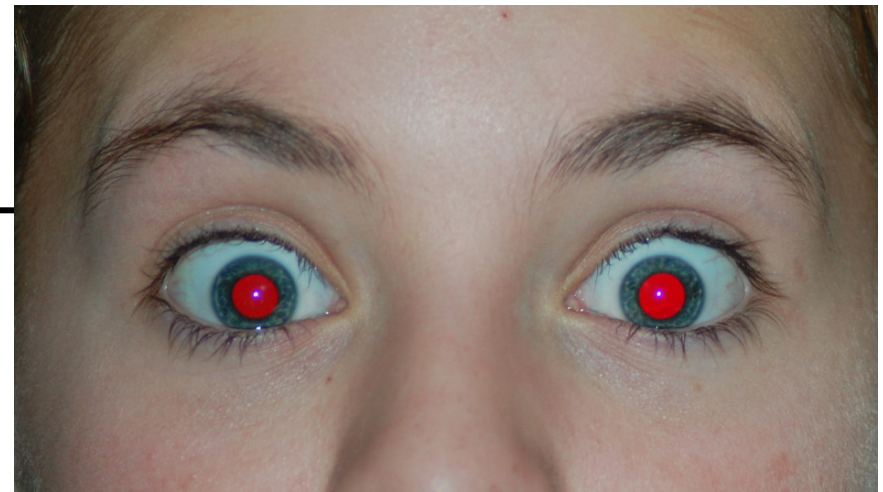
-3	2	4	2	1	-1
3	5	1	0	-2	-2
-2	3	2	-1	-2	-3
-3	-1	-4	-4	-4	-3
-2	0	4	4	-1	-2
-1	3	5	4	4	-1

-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)
- 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 **subarray  $A^*$**  of  $A$  such that the sum  $s^*$  of the values in  $A^*$  is **maximal** over all subarrays of  $A$ . If  $s^* < 0$ , return 0.*
- Remarks
  - We only want the **maximal value**, not the borders of  $A^*$
  - Cells may have positive or negative values (or 0)
  - **Length of the subarray  $A^*$**  is not fixed (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

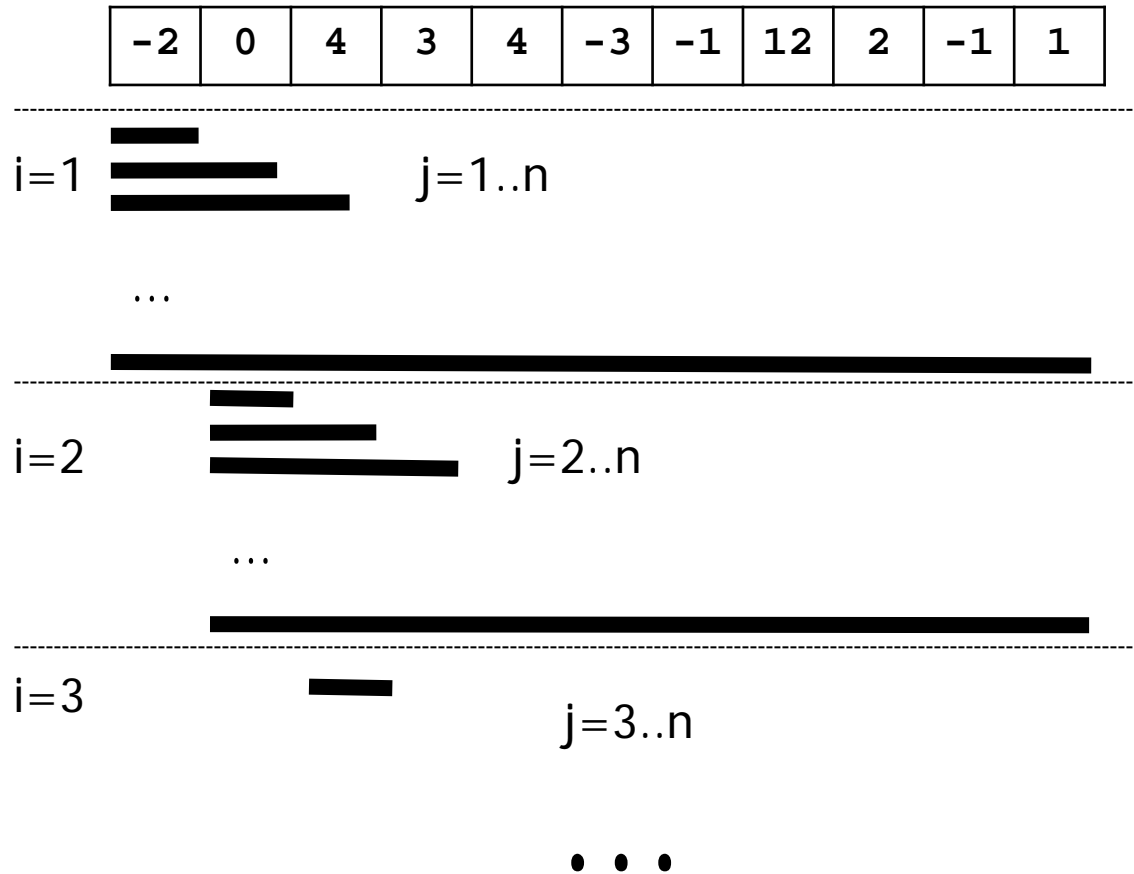
---

```
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 the **sum of the values** between start and end

# 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?
- Outmost loop: n times
- j-loop: n times (worst-case)
- 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?

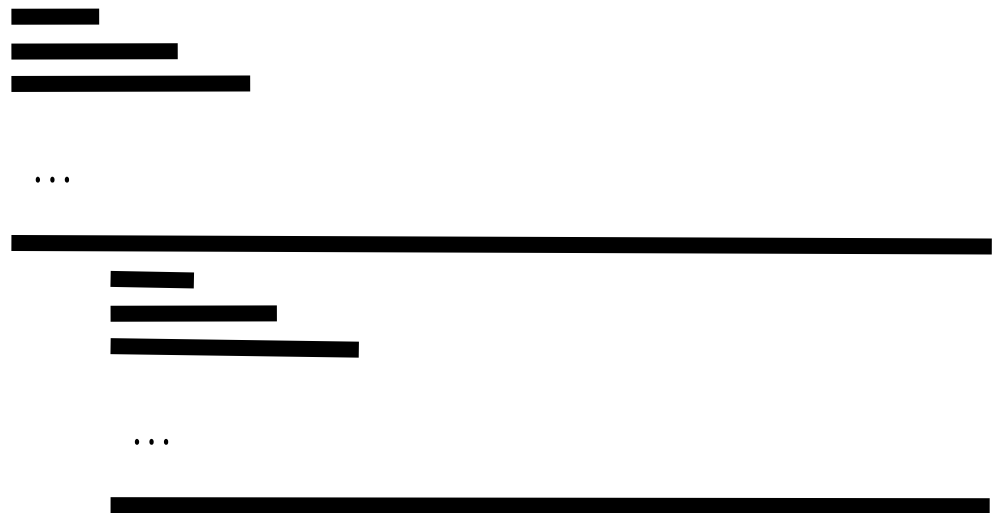


# Exhaustive Solution

---

- First sum:  $A[1]$
- Second:  $A[1]+A[2]$
- 3rd:  $A[1]+A[2]+A[3]$
- 4th: ...
  
- Every next sum is the **previous sum plus the next cell**
  
- How can we reuse the previous sum?

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



# Exhaustive Solution, Improved

---

- Every next sum is the previous sum plus the next cell
- 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

---

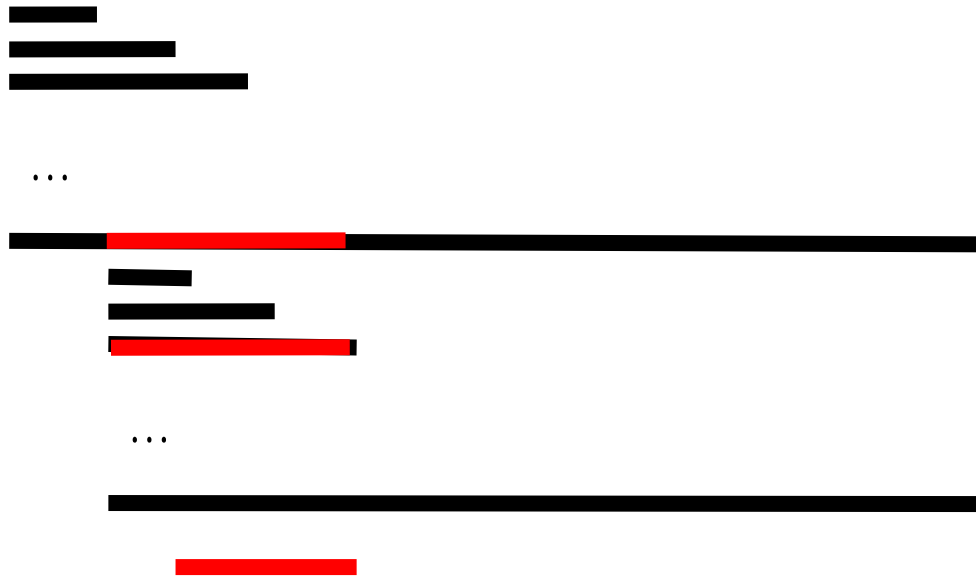
- The Max-Subarray Problem
- Naïve Solution
- Better Solution
- Best Solution

# 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

---

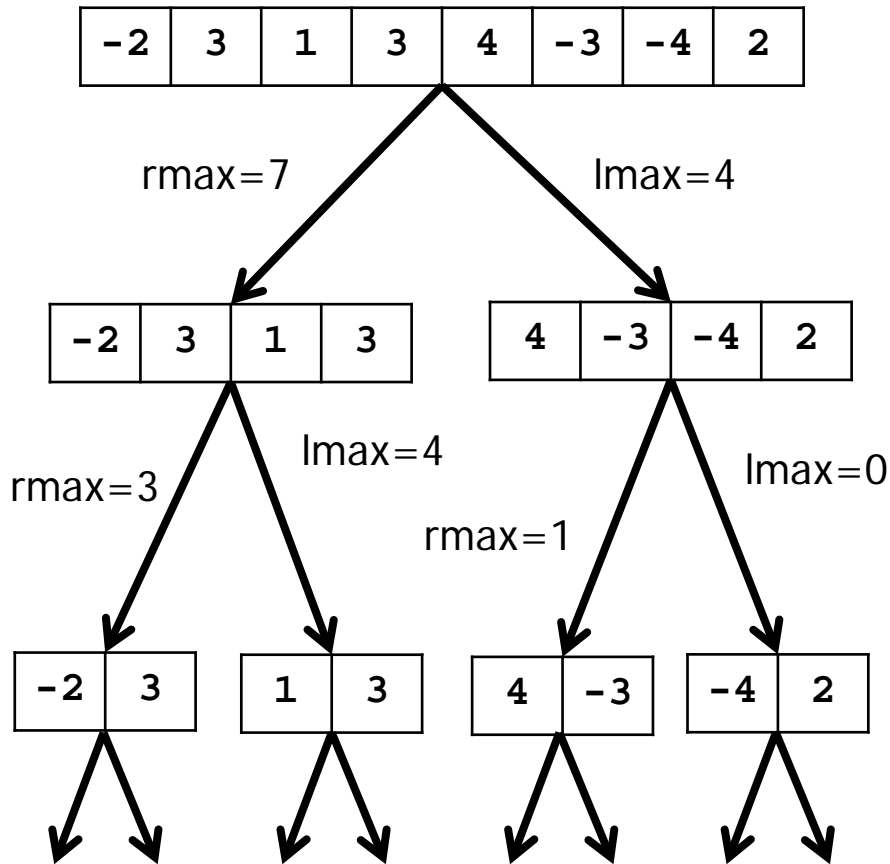
- We can break up our **problem into smaller ones** by looking only at parts of the array
- One scheme: Assume  $A = A_1 | A_2$ 
  - With “|” meaning array concatenation and  $|A_1| = |A_2| (+0/1)$
- The max-subarray (msa) of  $A$  ...
  - either lies in  $A_1$  – can be found by solving  $\text{msa}(A_1)$
  - or in  $A_2$  – can be found by solving  $\text{msa}(A_2)$
  - 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



- 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:  $T(n) \sim O(1)+O(n)+O(n)+T(n/2)+T(n/2)$

```
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

- 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:  $T(n) \sim O(1)+O(n)+T(n/2)+T(n/2)$

```
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

---

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

```
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;
}
```

# Complexity

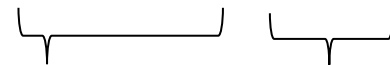
```
function msa (A: array_of_integer) {
  n := |A|;
  if (n=1) then
    if A[1]>0 then
      return A[1]
    else
      return 0;
  end if;
  m := n/2;      # Assume even sizes
  A1 := A[1..m];
  A2 := A[m+1..n];
  l1 := rmax(A1);
  l2 := lmax(A2);
  m := max( msa(A1), l1+l2, msa(A2));
  return m;
}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

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

$$T(n) = 2 \cdot T(n/2) + c_1 n =$$

$$= 2(2T(n/4) + c_1 n/2) + c_1 n = 4T(n/4) + c_1 n + c_1 n =$$

$$= 4(2T(n/8) + c_1 n/4) + 2c_1 n = 8T(n/8) + 3c_1 n = \dots$$



$$2^{\log(n)} \cdot c_2 + c_1 n \cdot \log(n) =$$

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

# Same Problem, Different Algorithms

---

- Naive:  $O(n^3)$
- Less naive, still redundant:  $O(n^2)$
- 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

---

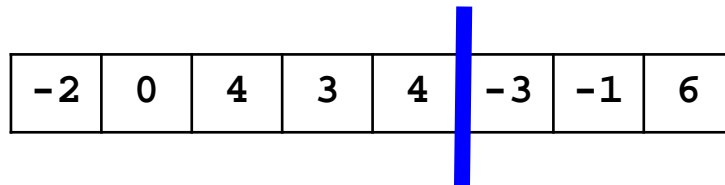
- Let's use **another strategy for dividing** the problem
- Let's look at the solutions for  $A[1]$ ,  $A[1..2]$ ,  $A[1..3]$ , ...
- What can we say about the **msa for  $A^{i+1}=A[1..i+1]$** , given the msa of  $A^i=A[1..i]$ ?

-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  $A^i=A[1..i]$ ?



- $\text{msa}(A^{i+1})$  is ...
  - either somewhere within  $A^i$ , which means the same as  $\text{msa}(A^i)$
  - or is formed by  $\text{rmax}(A^i) + A[i+1]$
- Thus: 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( A[i],  
                rmax+A[i]);  
    m := max( rmax, m);  
end for;
```

- 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  $\Omega(n)$**
- Example of **dynamic programming**: Build larger solutions from smaller ones

# Example

---

										rmax	m
-2	3	1	3	4	-3	-4	2			-2	-2
-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

# 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

# Types of Algorithms

---

- Different fundamental patterns (non exhaustive list)
  - **Greedy**: Find some promising start point and expand aggressively until a complete solution is found
    - Usually fast, but usually 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
  - ...

# Types of Algorithms

---

- For the max subarray problem
  - Greedy:  $O(n)$ , but wrong
  - Exhaustive:  $O(n^3)$ 
    - With pruning  $O(n^2)$
  - Divide & Conquer:  $O(n \cdot \log(n))$
  - Dynamic programming:  $O(n)$
  - Backtracking
  - ...
- Notes
  - No sharp way to differentiate alg patterns – just ideas
  - Usually there are many greedy, many exhaustive, many ... 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