2. Efficiency of algorithms

Efficiency of Algorithms, Random Access Machine Model, Function Growth, Asymptotics [Cormen et al, Kap. 2.2,3,4.2-4.4 | Ottman/Widmayer, Kap. 1.1]

Efficiency of Algorithms

Goals

- Quantify the runtime behavior of an algorithm independent of the machine.
- Compare efficiency of algorithms.
- Understand dependece on the input size.

Technology Model

Random Access Machine (RAM)

- Execution model: instructions are executed one after the other (on one processor core).
- Memory model: constant access time.
- Fundamental operations: computations ($+$,$-$,$\cdot$,...) comparisons, assignment / copy, flow control (jumps)
- Unit cost model: fundamental operations provide a cost of 1.
- Data types: fundamental types like size-limited integer or floating point number.

Size of the Input Data

Typical: number of input objects (of fundamental type).

Sometimes: number bits for a *reasonable / cost-effective* representation of the data.

Asymptotic behavior

An exact running time can normally not be predicted even for small input data.

- We consider the asymptotic behavior of the algorithm.
- And ignore all constant factors.

Example

An operation with cost 20 is no worse than one with cost 1
Linear growth with gradient 5 is as good as linear growth with gradient 1.

2.1 Function growth

$\mathcal{O}$, Θ, Ω [Cormen et al, Kap. 3; Ottman/Widmayer, Kap. 1.1]

Superficially

Use the asymptotic notation to specify the execution time of algorithms.

We write $\Theta(n^2)$ and mean that the algorithm behaves for large n like n^2: when the problem size is doubled, the execution time multiplies by four.

More precise: asymptotic upper bound

provided: a function $f : \mathbb{N} \to \mathbb{R}$.

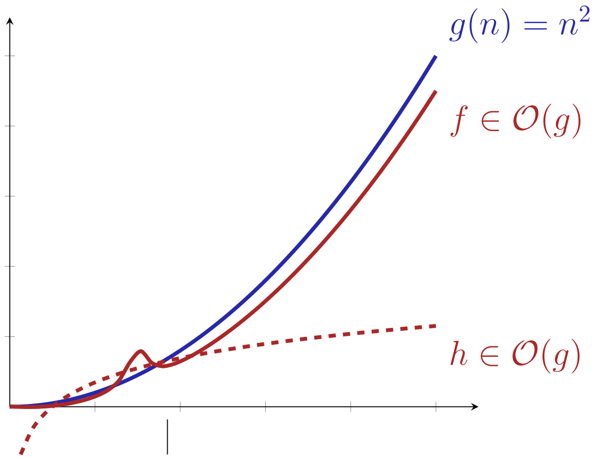
Definition:

$$\mathcal{O}(g) = \{f : \mathbb{N} \to \mathbb{R} \mid \exists c > 0, n_0 \in \mathbb{N} : 0 \le f(n) \le c \cdot g(n) \; \forall n \ge n_0\}$$

Notation:

$$\mathcal{O}(g(n)) := \mathcal{O}(g(\cdot)) = \mathcal{O}(g).$$

Graphic



Examples

$$\mathcal{O}(g) = \{f : \mathbb{N} \to \mathbb{R} | \ \exists c > 0, n_0 \in \mathbb{N} : 0 \leq f(n) \leq c \cdot g(n) \ \forall n \geq n_0\}$$

$f(n)$	$f \in \mathcal{O}(?)$	Example
$3n + 4$	$\mathcal{O}(n)$	$c = 4, n_0 = 4$
$2n$	$\mathcal{O}(n)$	$c = 2, n_0 = 0$
$n^2 + 100n$	$\mathcal{O}(n^2)$	$c = 2, n_0 = 100$
$n + \sqrt{n}$	$\mathcal{O}(n)$	$c = 2, n_0 = 1$

Property

$$f_1 \in \mathcal{O}(g), f_2 \in \mathcal{O}(g) \Rightarrow f_1 + f_2 \in \mathcal{O}(g)$$

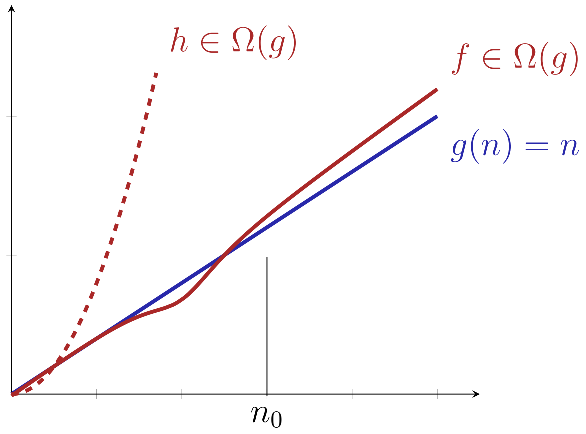
Converse: asymptotic lower bound

Given: a function $f : \mathbb{N} \to \mathbb{R}$.

Definition:

$$\Omega(g) = \{f : \mathbb{N} \to \mathbb{R} | \\ \exists c > 0, n_0 \in \mathbb{N} : 0 \leq c \cdot g(n) \leq f(n) \ \forall n \geq n_0\}$$

Example



Asymptotic tight bound

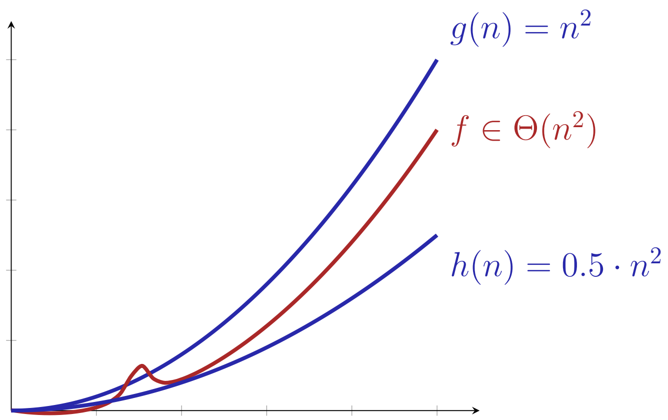
Given: function $f : \mathbb{N} \to \mathbb{R}$.

Definition:

$$\Theta(g) := \Omega(g) \cap \mathcal{O}(g).$$

Simple, closed form: exercise.

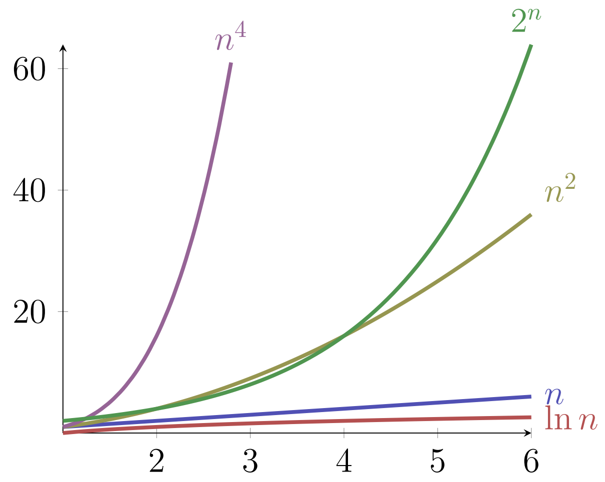
Example



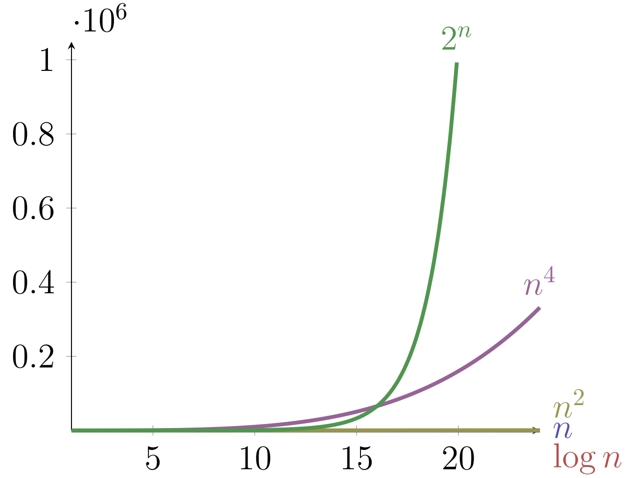
Notions of Growth

$\mathcal{O}(1)$	bounded	array access
$\mathcal{O}(\log \log n)$	double logarithmic	interpolated binary sorted sort
$\mathcal{O}(\log n)$	logarithmic	binary sorted search
$\mathcal{O}(\sqrt{n})$	like the square root	naive prime number test
$\mathcal{O}(n)$	linear	unsorted naive search
$\mathcal{O}(n \log n)$	superlinear / loglinear	good sorting algorithms
$\mathcal{O}(n^2)$	quadratic	simple sort algorithms
$\mathcal{O}(n^c)$	polynomial	matrix multiply
$\mathcal{O}(2^n)$	exponential	Travelling Salesman Dynamic Programming
$\mathcal{O}(n!)$	factorial	Travelling Salesman naively

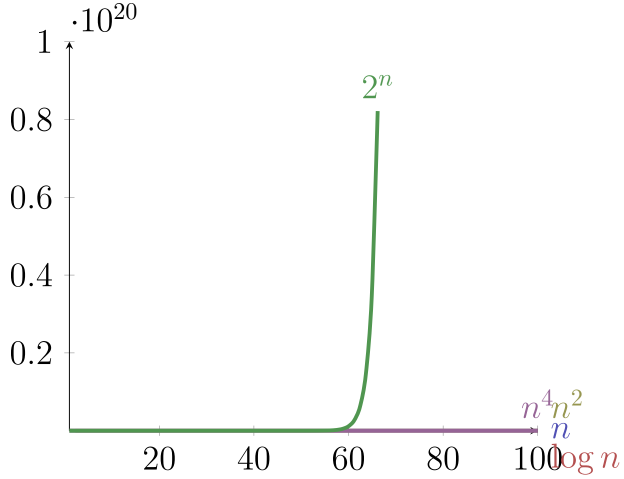
Small n



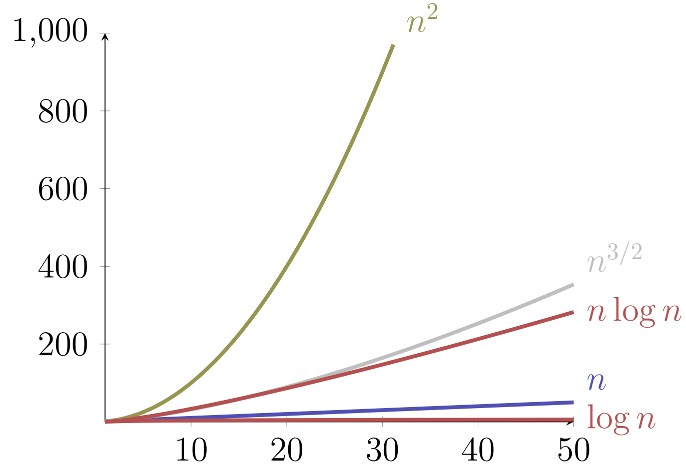
Larger n



"Large" n



Logarithms



Time Consumption

Assumption 1 Operation = $1\mu s$.

problem size	1	100	10000	10^6	10^9
$\log_2 n$	$1\mu s$	$7\mu s$	$13\mu s$	$20\mu s$	$30\mu s$
n	$1\mu s$	$100\mu s$	$1/100s$	$1s$	17 minutes
$n \log_2 n$	$1\mu s$	$700\mu s$	$13/100\mu s$	$20s$	8.5 hours
n^2	$1\mu s$	$1/100s$	1.7 minutes	11.5 days	317 centuries
2^n	$1\mu s$	10^{14} centuries	$\approx \infty$	$\approx \infty$	$\approx \infty$

A good strategy?

... Then I simply buy a new machine If today I can solve a problem of size n, then with a 10 or 100 times faster machine I can solve ...

Komplexität	(speed $\times 10$)	(speed $\times 100$)
$\log_2 n$	$n \to n^{10}$	$n \to n^{100}$
n	$n \to 10 \cdot n$	$n \to 100 \cdot n$
n^2	$n \to 3.16 \cdot n$	$n \to 10 \cdot n$
2^n	$n \to n + 3.32$	$n \to n + 6.64$

Examples

- $n \in \mathcal{O}(n^2)$ correct, but too imprecise: $n \in \mathcal{O}(n)$ and even $n \in \Theta(n)$.
- $3n^2 \in \mathcal{O}(2n^2)$ correct but uncommon: Omit constants: $3n^2 \in \mathcal{O}(n^2)$.
- $2n^2 \in \mathcal{O}(n)$ is wrong: $\frac{2n^2}{cn} = \frac{2}{c}n \underset{n\to\infty}{\to} \infty$!
- $\mathcal{O}(n) \subseteq \mathcal{O}(n^2)$ is correct
- $\Theta(n) \subseteq \Theta(n^2)$ is wrong $n \notin \Omega(n^2) \supset \Theta(n^2)$

Useful Tool

Theorem

Let $f, g : \mathbb{N} \to \mathbb{R}^+$ be two functions, then it holds that

1. $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f \in \mathcal{O}(g)$, $\mathcal{O}(f) \subsetneq \mathcal{O}(g)$.
2. $\lim_{n\to\infty} \frac{f(n)}{g(n)} = C > 0$ (C *constant*) $\Rightarrow f \in \Theta(g)$.
3. $\frac{f(n)}{g(n)} \underset{n\to\infty}{\to} \infty \Rightarrow g \in \mathcal{O}(f)$, $\mathcal{O}(g) \subsetneq \mathcal{O}(f)$.

About the Notation

Common notation

$$f = \mathcal{O}(g)$$

should be read as $f \in \mathcal{O}(g)$.

Clearly it holds that

$$f_1 = \mathcal{O}(g), f_2 = \mathcal{O}(g) \not\Rightarrow f_1 = f_2!$$

Beispiel

$n = \mathcal{O}(n^2), n^2 = \mathcal{O}(n^2)$ but naturally $n \neq n^2$.

Algorithms, Programs and Execution Time

Program: concrete implementation of an algorithm.

Execution time of the program: measurable value on a concrete machine. Can be bounded from above and below.

Beispiel

3GHz computer. Maximal number of operations per cycle (e.g. 8). $\Rightarrow$ lower bound.
A single operations does never take longer than a day $\Rightarrow$ upper bound.

From an *asymptotic* point of view the bounds coincide.

Complexity

Complexity of a problem P: minimal (asymptotic) costs over all algorithms A that solve P.

Complexity of the single-digit multiplication of two numbers with n digits is $\Omega(n)$ and $\mathcal{O}(n^{\log_3 2})$ (Karatsuba Ofman).

Example:

Problem	Complexity	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$
		$\uparrow$	$\uparrow$	$\uparrow$
Algorithm	Costs[2]	$3n-4$	$\mathcal{O}(n)$	$\Theta(n^2)$
		$\downarrow$	$\updownarrow$	$\updownarrow$
Program	Execution time	$\Theta(n)$	$\mathcal{O}(n)$	$\Theta(n^2)$

3. Design of Algorithms

Maximum Subarray Problem [Ottman/Widmayer, Kap. 1.3]
Divide and Conquer [Ottman/Widmayer, Kap. 1.2.2. S.9; Cormen et al, Kap. 4-4.1]

Algorithm Design

Inductive development of an algorithm: partition into subproblems, use solutions for the subproblems to find the overal solution.

Goal: development of the asymptotically most efficient (correct) algorithm.

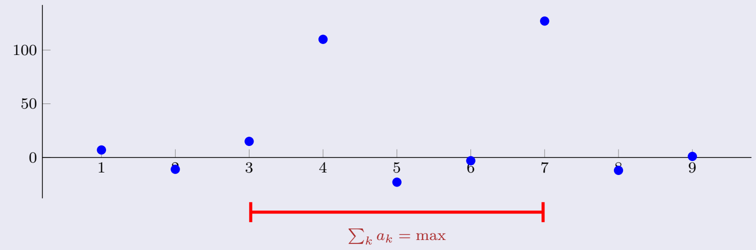
Efficiency towards run time costs (# fundamental operations) or /and memory consumption.

Maximum Subarray Problem

Given: an array of n rational numbers $(a_1, \dots, a_n)$.

Wanted: interval $[i, j]$, $1 \leq i \leq j \leq n$ with maximal positive sum $\sum_{k=i}^{j} a_k$.

Example: $a = (7, -11, 15, 110, -23, -3, 127, -12, 1)$



Naive Maximum Subarray Algorithm

Input : A sequence of n numbers $(a_1, a_2, \dots, a_n)$
Output : I, J such that $\sum_{k=I}^{J} a_k$ maximal.

$M \leftarrow 0$; $I \leftarrow 1$; $J \leftarrow 0$
for $i \in \{1, \dots, n\}$ **do**
  for $j \in \{i, \dots, n\}$ **do**
    $m = \sum_{k=i}^{j} a_k$
    if $m > M$ **then**
      $M \leftarrow m$; $I \leftarrow i$; $J \leftarrow j$

return I, J

Analysis

Theorem

The naive algorithm for the Maximum Subarray problem executes $\Theta(n^3)$ additions.

Beweis:

$$\sum_{i=1}^{n}\sum_{j=i}^{n}(j-i) = \sum_{i=1}^{n}\sum_{j=0}^{n-i} j = \sum_{i=1}^{n}\sum_{j=1}^{n-i} j == \sum_{i=1}^{n}\frac{(n-i)(n-i+1)}{2}$$

$$= \sum_{i=0}^{n-1}\frac{i\cdot(i+1)}{2} = \frac{1}{2}\left(\sum_{i=0}^{n-1} i^2 + \sum_{i=0}^{n-1} i\right)$$

$$= \frac{1}{2}\left(\Theta(n^3) + \Theta(n^2)\right) = \Theta(n^3).$$

■

Observation

$$\sum_{k=i}^{j} a_k = \underbrace{\left(\sum_{k=1}^{j} a_k\right)}_{S_j} - \underbrace{\left(\sum_{k=1}^{i-1} a_k\right)}_{S_{i-1}}$$

Prefix sums

$$S_i := \sum_{k=1}^{i} a_k.$$

Maximum Subarray Algorithm with Prefix Sums

Input : A sequence of n numbers $(a_1, a_2, \dots, a_n)$
Output : I, J such that $\sum_{k=J}^{J} a_k$ maximal.

$\mathcal{S}_0 \leftarrow 0$
for $i \in \{1, \dots, n\}$ **do** // prefix sum
  $\mathcal{S}_i \leftarrow \mathcal{S}_{i-1} + a_i$
$M \leftarrow 0$; $I \leftarrow 1$; $J \leftarrow 0$
for $i \in \{1, \dots, n\}$ **do**
  for $j \in \{i, \dots, n\}$ **do**
    $m = \mathcal{S}_j - \mathcal{S}_{i-1}$
    if $m > M$ **then**
      $M \leftarrow m$; $I \leftarrow i$; $J \leftarrow j$

Analysis

Theorem

The prefix sum algorithm for the Maximum Subarray problem conducts $\Theta(n^2)$ additions and subtractions.

Beweis:

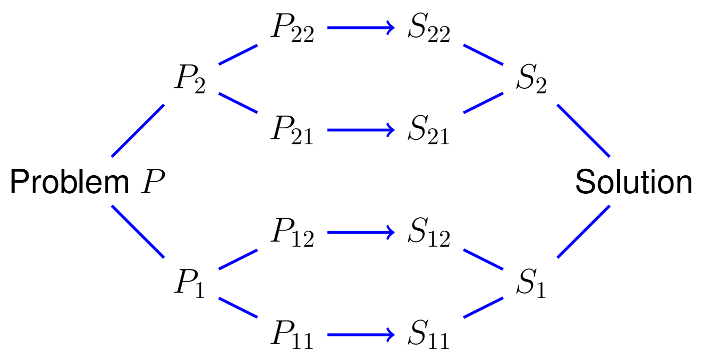
$$\sum_{i=1}^{n} 1 + \sum_{i=1}^{n}\sum_{j=i}^{n} 1 = n + \sum_{i=1}^{n}(n-i+1) = n + \sum_{i=1}^{n} i = \Theta(n^2)$$

■

divide et impera

Divide and Conquer

Divide the problem into subproblems that contribute to the simplified computation of the overal problem.



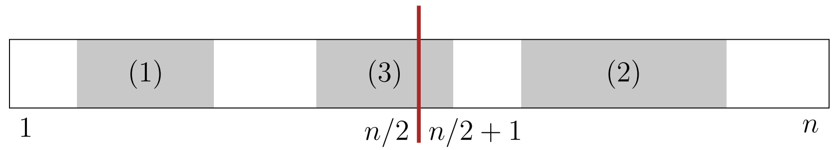
Maximum Subarray – Divide

- Divide: Divide the problem into two (roughly) equally sized halves:
  $(a_1, \dots, a_n) = (a_1, \dots, a_{\lfloor n/2 \rfloor}, \quad a_{\lfloor n/2 \rfloor + 1}, \dots, a_1)$
- Simplifying assumption: $n = 2^k$ for some $k \in \mathbb{N}$.

Maximum Subarray – Conquer

If i and j are indices of a solution $\Rightarrow$ case by case analysis:

1. Solution in left half $1 \le i \le j \le n/2 \Rightarrow$ Recursion (left half)
2. Solution in right half $n/2 < i \le j \le n \Rightarrow$ Recursion (right half)
3. Solution in the middle $1 \le i \le n/2 < j \le n \Rightarrow$ Subsequent observation



Maximum Subarray – Observation

Assumption: solution in the middle $1 \le i \le n/2 < j \le n$

$$
\begin{aligned}
S_{\max} &= \max_{\substack{1 \le i \le n/2 \\ n/2 < j \le n}} \sum_{k=i}^{j} a_k = \max_{\substack{1 \le i \le n/2 \\ n/2 < j \le n}} \left(\sum_{k=i}^{n/2} a_k + \sum_{k=n/2+1}^{j} a_k \right) \\
&= \max_{1 \le i \le n/2} \sum_{k=i}^{n/2} a_k + \max_{n/2 < j \le n} \sum_{k=n/2+1}^{j} a_k \\
&= \max_{1 \le i \le n/2} \underbrace{S_{n/2} - S_{i-1}}_{\text{suffix sum}} + \max_{n/2 < j \le n} \underbrace{S_j - S_{n/2}}_{\text{prefix sum}}
\end{aligned}
$$

Maximum Subarray Divide and Conquer Algorithm

Input : A sequence of n numbers $(a_1, a_2, \dots, a_n)$
Output : Maximal $\sum_{k=i'}^{j'} a_k$.
if $n = 1$ **then**
  return $\max\{a_1, 0\}$
else
  Divide $a = (a_1, \dots, a_n)$ in $A_1 = (a_1, \dots, a_{n/2})$ und $A_2 = (a_{n/2+1}, \dots, a_n)$
  Recursively compute best solution W_1 in A_1
  Recursively compute best solution W_2 in A_2
  Compute greatest suffix sum S in A_1
  Compute greatest prefix sum P in A_2
  Let $W_3 \leftarrow S + P$
  return $\max\{W_1, W_2, W_3\}$

Analysis

Theorem

The divide and conquer algorithm for the maximum subarray sum problem conducts a number of $\Theta(n \log n)$ additions and comparisons.

Analysis

Input : A sequence of n numbers $(a_1, a_2, \ldots, a_n)$
Output : Maximal $\sum_{k=i'}^{j'} a_k$.

if $n = 1$ **then**
$\Theta(1)$ **return** $\max\{a_1, 0\}$
else
$\Theta(1)$ Divide $a = (a_1, \ldots, a_n)$ in $A_1 = (a_1, \ldots, a_{n/2})$ und $A_2 = (a_{n/2+1}, \ldots, a_n)$
$T(n/2)$ Recursively compute best solution W_1 in A_1
$T(n/2)$ Recursively compute best solution W_2 in A_2
$\Theta(n)$ Compute greatest suffix sum S in A_1
$\Theta(n)$ Compute greatest prefix sum P in A_2
$\Theta(1)$ Let $W_3 \leftarrow S + P$
$\Theta(1)$ **return** $\max\{W_1, W_2, W_3\}$

Analysis

Recursion equation

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T(\frac{n}{2}) + a \cdot n & \text{if } n > 1 \end{cases}$$

Analysis

Mit $n = 2^k$:

$$\overline{T}(k) = \begin{cases} c & \text{if } k = 0 \\ 2\overline{T}(k-1) + a \cdot 2^k & \text{if } k > 0 \end{cases}$$

Solution:

$$\overline{T}(k) = 2^k \cdot c + \sum_{i=0}^{k-1} 2^i \cdot a \cdot 2^{k-i} = c \cdot 2^k + a \cdot k \cdot 2^k = \Theta(k \cdot 2^k)$$

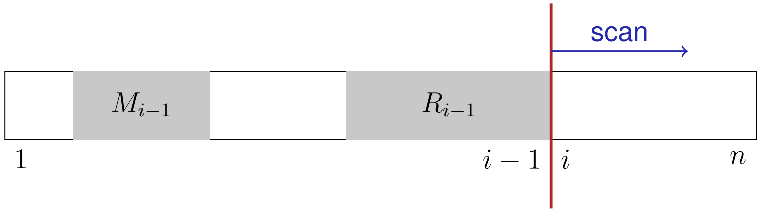
also

$$T(n) = \Theta(n \log n)$$

■

Maximum Subarray Sum Problem – Inductively

Assumption: maximal value M_{i-1} of the subarray sum is known for $(a_1, \dots, a_{i-1})$ $(1 < i \leq n)$.



a_i: generates at most a better interval at the right bound (prefix sum).

$R_{i-1} \Rightarrow R_i = \max\{R_{i-1} + a_i, 0\}$

Inductive Maximum Subarray Algorithm

Input : A sequence of n numbers $(a_1, a_2, \dots, a_n)$.
Output : $\max\{0, \max_{i,j} \sum_{k=i}^{j} a_k\}$.

$M \leftarrow 0$
$R \leftarrow 0$
for $i = 1 \dots n$ **do**
  $R \leftarrow R + a_i$
  if $R < 0$ **then**
    $R \leftarrow 0$
  if $R > M$ **then**
    $M \leftarrow R$
return M;

Analysis

Theorem

The inductive algorithm for the Maximum Subarray problem conducts a number of $\Theta(n)$ additions and comparisons.

Complexity of the problem?

Can we improve over $\Theta(n)$?

Every correct algorithm for the Maximum Subarray Sum problem must consider each element in the algorithm.

Assumption: the algorithm does not consider a_i.

1. The algorithm provides a solution including a_i. Repeat the algorithm with a_i so small that the solution must not have contained the point in the first place.
2. The algorithm provides a solution not including a_i. Repeat the algorithm with a_i so large that the solution must have contained the point in the first place.

Complexity of the maximum Subarray Sum Problem

Theorem

The Maximum Subarray Sum Problem has Complexity $\Theta(n)$.

Beweis: Inductive algorithm with asymptotic execution time $\mathcal{O}(n)$.
Every algorithm has execution time $\Omega(n)$.
Thus the complexity of the problem is $\Omega(n) \cap \mathcal{O}(n) = \Theta(n)$. ■