# 3. Examples

Show Correctness, Recursion and Recurrences
[References to literatur at the examples]



## 3.1 Ancient Egyptian Multiplication

Ancient Egyptian Multiplication– Example on how to show correctness of algorithms.



## Ancient Egyptian Multiplication[3]

Compute $11 \cdot 9$

| | |
|---|---|
| 11 | 9 |
| ~~22~~ | ~~4~~ |
| ~~44~~ | ~~2~~ |
| 88 | 1 |
| 99 | – |

| | |
|---|---|
| 9 | 11 |
| 18 | 5 |
| ~~36~~ | ~~2~~ |
| 72 | 1 |
| 99 | |

1. Double left, integer division by 2 on the right
2. Even number on the right $\Rightarrow$ eliminate row.
3. Add remaining rows on the left.

[3] Also known as russian multiplication



## Advantages

- Short description, easy to grasp
- Efficient to implement on a computer: double = left shift, divide by 2 = right shift

**Beispiel**

*left shift* $9 = 01001_2 \to 10010_2 = 18$
*right shift* $9 = 01001_2 \to 00100_2 = 4$

## Questions

- For which kind of inputs does the algorithm deliver a correct result (in finite time)?
- How do you prove its correctness?
- What is a good measure for Efficiency?



## The Essentials

If $b > 1$, $a \in \mathbb{Z}$, then:

$$a \cdot b = \begin{cases} 2a \cdot \frac{b}{2} & \text{falls } b \text{ gerade,} \\ a + 2a \cdot \frac{b-1}{2} & \text{falls } b \text{ ungerade.} \end{cases}$$



## Termination

$$a \cdot b = \begin{cases} a & \text{falls } b = 1, \\ 2a \cdot \frac{b}{2} & \text{falls } b \text{ gerade,} \\ a + 2a \cdot \frac{b-1}{2} & \text{falls } b \text{ ungerade.} \end{cases}$$



## Recursively, Functional

$$f(a, b) = \begin{cases} a & \text{falls } b = 1, \\ f(2a, \frac{b}{2}) & \text{falls } b \text{ gerade,} \\ a + f(2a, \frac{b-1}{2}) & \text{falls } b \text{ ungerade.} \end{cases}$$

## Implemented as a function

```
// pre: b>0
// post: return a*b
int f(int a, int b){
  if(b==1)
    return a;
  else if (b%2 == 0)
    return f(2*a, b/2);
  else
    return a + f(2*a, (b-1)/2);
}
```



## Correctnes: Mathematical Proof

$$f(a,b) = \begin{cases} a & \text{if } b = 1, \\ f(2a, \frac{b}{2}) & \text{if } b \text{ even}, \\ a + f(2a \cdot \frac{b-1}{2}) & \text{if } b \text{ odd}. \end{cases}$$

Remaining to show: $f(a,b) = a \cdot b$ for $a \in \mathbb{Z}$, $b \in \mathbb{N}^+$.



## Correctnes: Mathematical Proof by Induction

Let $a \in \mathbb{Z}$, to show $f(a,b) = a \cdot b \quad \forall b \in \mathbb{N}^+$.

*Base clause:* $f(a,1) = a = a \cdot 1$
*Hypothesis:* $f(a,b') = a \cdot b' \quad \forall 0 < b' \leq b$
*Step:* $f(a,b') = a \cdot b' \quad \forall 0 < b' \leq b \overset{!}{\Rightarrow} f(a, b+1) = a \cdot (b+1)$

$$f(a, b+1) = \begin{cases} f(2a, \overbrace{\frac{b+1}{2}}^{0<\cdot\leq b}) \overset{i.H.}{=} a \cdot (b+1) & \text{if } b > 0 \text{ odd}, \\ a + f(2a, \underbrace{\frac{b}{2}}_{0<\cdot<b}) \overset{i.H.}{=} a + a \cdot b & \text{if } b > 0 \text{ even}. \end{cases}$$

■



## [Code Transformations: End Recursion]

The recursion can be writen as *end recursion*

```
// pre: b>0
// post: return a*b
int f(int a, int b){
  if(b==1)
    return a;
  else if (b%2 == 0)
    return f(2*a, b/2);
  else
    return a + f(2*a, (b-1)/2);
}
```

→

```
// pre: b>0
// post: return a*b
int f(int a, int b){
  if(b==1)
    return a;
  int z=0;
  if (b%2 != 0){
    --b;
    z=a;
  }
  return z + f(2*a, b/2);
}
```

## [Code-Transformation: End-Recursion ⇒ Iteration]

```
// pre: b>0
// post: return a*b
int f(int a, int b){
  if(b==1)
    return a;
  int z=0;
  if (b%2 != 0){
    --b;
    z=a;
  }
  return z + f(2*a, b/2);
}
```

→

```
int f(int a, int b) {
  int res = 0;
  while (b != 1) {
    int z = 0;
    if (b % 2 != 0){
      --b;
      z = a;
    }
    res += z;
    a *= 2; // neues a
    b /= 2; // neues b
  }
  res += a; // Basisfall b=1
  return res;
}
```



## [Code-Transformation: Simplify]

```
int f(int a, int b) {
  int res = 0;
  while (b != 1) {
    int z = 0;
    if (b % 2 != 0){
      --b;     ⟶ Teil der Division
      z = a;   ⟶ Direkt in res
    }
    res += z;
    a *= 2;
    b /= 2;
  }
  res += a;    ⟶ in den Loop
  return res;
}
```

→

```
// pre: b>0
// post: return a*b
int f(int a, int b) {
  int res = 0;
  while (b > 0) {
    if (b % 2 != 0)
      res += a;
    a *= 2;
    b /= 2;
  }
  return res;
}
```



## Correctness: Reasoning using Invariants!

```
// pre: b>0
// post: return a*b
int f(int a, int b) {
  int res = 0;
  ------------------------
  while (b > 0) {
    if (b % 2 != 0){
  ------------------------
      res += a;
      --b;
  ------------------------
    }
  ------------------------
    a *= 2;
    b /= 2;
  ------------------------
  }
  ------------------------
  return res;
}
```

Sei $x := a \cdot b$.

here: $x = \boxed{a \cdot b + res}$

if here $x = a \cdot b + res$ ...

... then also here $x = a \cdot b + res$

$b$ even

here: $x = a \cdot b + res$

here: $x = a \cdot b + res$ und $b = 0$

Also $res = x$.



## Conclusion

The expression $a \cdot b + res$ is an *invariant*

- Values of $a$, $b$, $res$ change but the invariant remains basically unchanged: The invariant is only temporarily discarded by some statement but then re-established. If such short statement sequences are considered atomiv, the value remains indeed invariant
- In particular the loop contains an invariant, called *loop invariant* and it operates there like the induction step in induction proofs.
- Invariants are obviously powerful tools for proofs!

# [Further simplification]

```
// pre: b>0
// post: return a∗b
int f(int a, int b) {
  int res = 0;
  while (b > 0) {
    if (b % 2 != 0){
      res += a;
      −−b;
    }
    a ∗= 2;
    b /= 2;
  }
  return res;
}
```

```
// pre: b>0
// post: return a∗b
int f(int a, int b) {
  int res = 0;
  while (b > 0) {
    res += a ∗ (b%2);
    a ∗= 2;
    b /= 2;
  }
  return res;
}
```



# [Analysis]

```
// pre: b>0
// post: return a∗b
int f(int a, int b) {
  int res = 0;
  while (b > 0) {
    res += a ∗ (b%2);
    a ∗= 2;
    b /= 2;
  }
  return res;
}
```

Ancient Egyptian Multiplication corresponds to the school method with radix $2$.

$$
\begin{array}{ccccccccccc}
1 & 0 & 0 & 1 & \times & 1 & 0 & 1 & 1 & \\
\hline
 & & & & & 1 & 0 & 0 & 1 & (9) \\
 & & & & 1 & 0 & 0 & 1 & & (18) \\
\hline
 & & & & 1 & 1 & 0 & 1 & 1 & \\
 & & 1 & 0 & 0 & 1 & & & & (72) \\
\hline
 & & 1 & 1 & 0 & 0 & 0 & 1 & 1 & (99)
\end{array}
$$



# Efficiency

Question: how long does a multiplication of $a$ and $b$ take?

- Measure for efficiency
  - Total number of fundamental operations: double, divide by 2, shift, test for “even”, addition
  - In the recursive and recursive code: maximally 6 operations per call or iteration, respectively
- Essential criterion:
  - Number of recursion calls or
  - Number iterations (in the iterative case)
- $\frac{b}{2^n} \leq 1$ holds for $n \geq \log_2 b$. Consequently not more than $6\lceil \log_2 b \rceil$ fundamental operations.



# 3.2 Fast Integer Multiplication

[Ottman/Widmayer, Kap. 1.2.3]

## Example 2: Multiplication of large Numbers

Primary school:

$$\begin{array}{cccccc|l} & a & b & & c & d & \\ & 6 & 2 & \cdot & 3 & 7 & \\ \hline & & & & 1 & 4 & d\cdot b \\ & & & 4 & 2 & & d\cdot a \\ & & & & 6 & & c\cdot b \\ & & 1 & 8 & & & c\cdot a \\ \hline = & & 2 & 2 & 9 & 4 & \end{array}$$

$2 \cdot 2 = 4$ single-digit multiplications. $\Rightarrow$ Multiplication of two $n$-digit numbers: $n^2$ single-digit multiplications



## Observation

$$\begin{aligned} ab \cdot cd &= (10 \cdot a + b) \cdot (10 \cdot c + d) \\ &= 100 \cdot a \cdot c + 10 \cdot a \cdot c \\ &\quad + 10 \cdot b \cdot d + b \cdot d \\ &\quad + 10 \cdot (a - b) \cdot (d - c) \end{aligned}$$



## Improvement?

$$\begin{array}{cccccc|l} & a & b & & c & d & \\ & 6 & 2 & \cdot & 3 & 7 & \\ \hline & & & & 1 & 4 & d\cdot b \\ & & & 1 & 4 & & d\cdot b \\ & & & 1 & 6 & & (a-b)\cdot(d-c) \\ & & & 1 & 8 & & c\cdot a \\ & & 1 & 8 & & & c\cdot a \\ \hline = & & 2 & 2 & 9 & 4 & \end{array}$$

$\rightarrow$ $3$ single-digit multiplications.



## Large Numbers

$$6237 \cdot 5898 = \underbrace{62}_{a'}\underbrace{37}_{b'} \cdot \underbrace{58}_{c'}\underbrace{98}_{d'}$$

Recursive / inductive application: compute $a' \cdot c'$, $a' \cdot d'$, $b' \cdot c'$ and $c' \cdot d'$ as shown above.

$\rightarrow 3 \cdot 3 = 9$ instead of $16$ single-digit multiplications.

## Generalization

Assumption: two numbers with $n$ digits each, $n = 2^k$ for some $k$.

$$\begin{aligned}(10^{n/2}a + b) \cdot (10^{n/2}c + d) = 10^n \cdot a \cdot c + 10^{n/2} \cdot a \cdot c \\ + 10^{n/2} \cdot b \cdot d + b \cdot d \\ + 10^{n/2} \cdot (a - b) \cdot (d - c)\end{aligned}$$

Recursive application of this formula: algorithm by Karatsuba and Ofman (1962).



## Analysis

$M(n)$: Number of single-digit multiplications.

Recursive application of the algorithm from above $\Rightarrow$ recursion equality:

$$M(2^k) = \begin{cases} 1 & \text{if } k = 0, \\ 3 \cdot M(2^{k-1}) & \text{if } k > 0. \end{cases}$$



## Iterative Substition

Iterative substition of the recursion formula in order to guess a solution of the recursion formula:

$$\begin{aligned}M(2^k) &= 3 \cdot M(2^{k-1}) = 3 \cdot 3 \cdot M(2^{k-2}) = 3^2 \cdot M(2^{k-2}) \\ &= \ldots \\ &\overset{!}{=} 3^k \cdot M(2^0) = 3^k.\end{aligned}$$



## Proof: induction

*Hypothesis H*:

$$M(2^k) = 3^k.$$

*Base clause ($k = 0$)*:

$$M(2^0) = 3^0 = 1. \quad \checkmark$$

*Induction step ($k \to k + 1$)*:

$$M(2^{k+1}) \overset{\text{def}}{=} 3 \cdot M(2^k) \overset{\text{H}}{=} 3 \cdot 3^k = 3^{k+1}.$$

■

## Comparison

Traditionally $n^2$ single-digit multiplications.

Karatsuba/Ofman:

$$M(n) = 3^{\log_2 n} = (2^{\log_2 3})^{\log_2 n} = 2^{\log_2 3 \log_2 n} = n^{\log_2 3} \approx n^{1.58}.$$

Example: number with $1000$ digits: $1000^2/1000^{1.58} \approx 18$.



## Best possible algorithm?

We only know the upper bound $n^{\log_2 3}$.

There are (for large $n$) practically relevant algorithms that are faster. Example: Schönhage-Strassen algorithm (1971) based on fast Fouriertransformation with running time $\mathcal{O}(n \log n \cdot \log \log n)$. The best upper bound is not known.

Lower bound: $n$. Each digit has to be considered at least once.



## Appendix: Asymptotics with Addition and Shifts

For each multiplication of two $n$-digit numbers we also should take into account a constant number of additions, subtractions and shifts

Additions, subtractions and shifts of $n$-digit numbers cost $\mathcal{O}(n)$

Therefore the asymptotic running time is determined (with some $c > 1$) by the following recurrence

$$T(n) = \begin{cases} 3 \cdot T\left(\frac{1}{2}n\right) + c \cdot n & \text{if } n > 1 \\ 1 & \text{otherwise} \end{cases}$$



## Appendix: Asymptotics with Addition and Shifts

Assumption: $n = 2^k$, $k > 0$

$$\begin{aligned} T(2^k) &= 3 \cdot T\left(2^{k-1}\right) + c \cdot 2^k \\ &= 3 \cdot (3 \cdot T(2^{k-2}) + c \cdot 2^{k-1}) + c \cdot 2^k \\ &= 3 \cdot (3 \cdot (3 \cdot T(2^{k-3}) + c \cdot 2^{k-2}) + c \cdot 2^{k-1}) + c \cdot 2^k \\ &= 3 \cdot (3 \cdot (...(3 \cdot T(2^{k-k}) + c \cdot 2^1)...) + c \cdot 2^{k-1}) + c \cdot 2^k \\ &= 3^k \cdot T(1) + c \cdot 3^{k-1} 2^1 + c \cdot 3^{k-2} 2^2 + ... + c \cdot 3^0 2^k \\ &\leq c \cdot 3^k \cdot (1 + 2/3 + (2/3)^2 + ... + (2/3)^k) \end{aligned}$$

Die geometrische Reihe $\sum_{i=0}^{k} \varrho^i$ mit $\varrho = 2/3$ konvergiert für $k \to \infty$ gegen $\frac{1}{1-\varrho} = 3$.

Somit $T(2^k) \leq c \cdot 3^k \cdot 3 \in \Theta(3^k) = \Theta(3^{\log_2 n}) = \Theta(n^{\log_2 3})$.

## 3.3 Maximum Subarray Problem

Algorithm Design – 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.

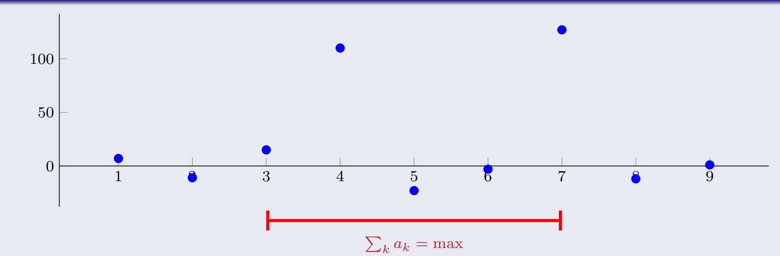


# Maximum Subarray Problem

*Given:* an array of $n$ real 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:

$$\sum_{i=1}^{n}\sum_{j=i}^{n}(j-i+1) = \sum_{i=1}^{n}\sum_{j=0}^{n-i}(j+1) = \sum_{i=1}^{n}\sum_{j=1}^{n-i+1} j = \sum_{i=1}^{n}\frac{(n-i+1)(n-i+2)}{2}$$
$$= \sum_{i=0}^{n}\frac{i\cdot(i+1)}{2} = \frac{1}{2}\left(\sum_{i=1}^{n} i^2 + \sum_{i=1}^{n} i\right)$$
$$= \frac{1}{2}\left(\frac{n(2n+1)(n+1)}{6} + \frac{n(n+1)}{2}\right) = \frac{n^3+3n^2+2n}{6} = \Theta(n^3).$$

■



## 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 $(a_1, a_2, \ldots, a_n)$
**Output**: $I$, $J$ such that $\sum_{k=J}^{J} a_k$ maximal.

$\mathcal{S}_0 \leftarrow 0$
**for** $i \in \{1, \ldots, n\}$ **do** // prefix sum
  $\mathcal{S}_i \leftarrow \mathcal{S}_{i-1} + a_i$
$M \leftarrow 0$; $I \leftarrow 1$; $J \leftarrow 0$
**for** $i \in \{1, \ldots, n\}$ **do**
  **for** $j \in \{i, \ldots, n\}$ **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:

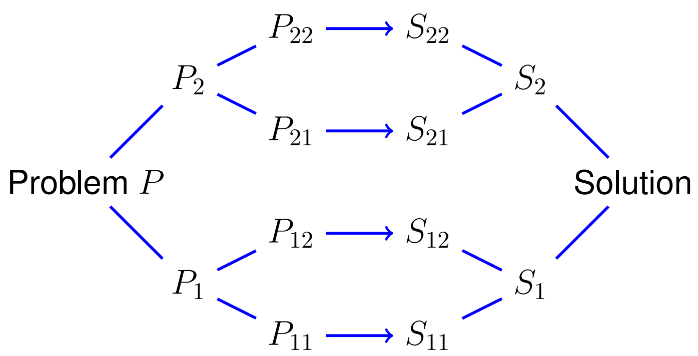
$$\sum_{i=1}^{n} 1 + \sum_{i=1}^{n}\sum_{j=i}^{n} 1 = n + \sum_{i=1}^{n}(n-i+1) = n + \sum_{i=1}^{n} i = \Theta(n^2)$$

■

## divide et impera

**Divide and Conquer**

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






## Maximum Subarray – Divide

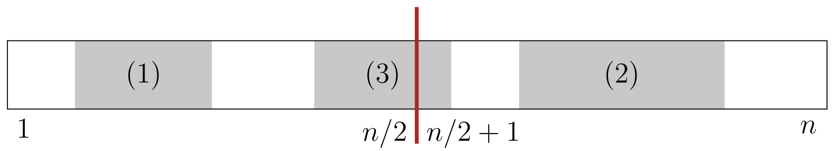
- Divide: Divide the problem into two (roughly) equally sized halves: $(a_1, \ldots, a_n) = (a_1, \ldots, a_{\lfloor n/2 \rfloor}, \quad a_{\lfloor n/2 \rfloor + 1}, \ldots, a_1)$
- Simplifying assumption: $n = 2^k$ for some $k \in \mathbb{N}$.



## 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 $(a_1, a_2, \dots, a_n)$
**Output**: Maximal $\sum_{k=i'}^{j'} a_k$.

**if** $n = 1$ **then**
$\Theta(1)$ **return** $\max\{a_1, 0\}$
**else**
$\Theta(1)$ Divide $a = (a_1, \dots, a_n)$ in $A_1 = (a_1, \dots, a_{n/2})$ und $A_2 = (a_{n/2+1}, \dots, a_n)$
$T(n/2)$ 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

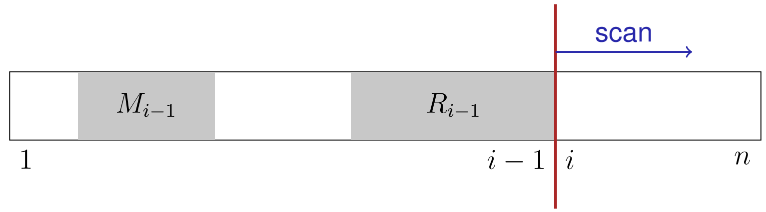
$$T(n) = \Theta(n \log n)$$

■



## Maximum Subarray Sum Problem – Inductively

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




$a_i$: 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 $\Omega(n) \cap \mathcal{O}(n) = \Theta(n)$. ■



# 3.4 Appendix

Derivation of some mathemmatical formulas



## Sums

$$\sum_{i=0}^{n} i^2 = \frac{n \cdot (n+1) \cdot (2n+1)}{6}$$

Trick:

$$\sum_{i=1}^{n} i^3 - (i-1)^3 = \sum_{i=0}^{n} i^3 - \sum_{i=0}^{n-1} i^3 = n^3$$

$$\sum_{i=1}^{n} i^3 - (i-1)^3 = \sum_{i=1}^{n} i^3 - i^3 + 3i^2 - 3i + 1 = n - \frac{3}{2} n \cdot (n+1) + 3 \sum_{i=0}^{n} i^2$$

$$\Rightarrow \sum_{i=0}^{n} i^2 = \frac{1}{6}(2n^3 + 3n^2 + n) \in \Theta(n^3)$$

Can easily be generalized: $\sum_{i=1}^{n} i^k \in \Theta(n^{k+1})$.

# Geometric Series

$$\sum_{i=0}^{n} \rho^i \stackrel{!}{=} \frac{1-\rho^{n+1}}{1-\rho}$$

$$\begin{aligned}\sum_{i=0}^{n} \rho^i \cdot (1-\varrho) &= \sum_{i=0}^{n} \rho^i - \sum_{i=0}^{n} \rho^{i+1} = \sum_{i=0}^{n} \rho^i - \sum_{i=1}^{n+1} \rho^i \\ &= \rho^0 - \rho^{n+1} = 1 - \rho^{n+1}.\end{aligned}$$

For $0 \leq \rho < 1$:

$$\sum_{i=0}^{\infty} \rho^i = \frac{1}{1-\rho}$$