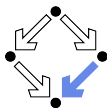


Analysis of Complexity

Wolfgang Schreiner
Wolfgang.Schreiner@risc.jku.at

Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria
<http://www.risc.jku.at>





1. Example

2. Sums

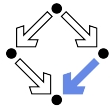
3. Recurrences

4. Divide and Conquer

5. Randomization

6. Amortized Analysis

Example



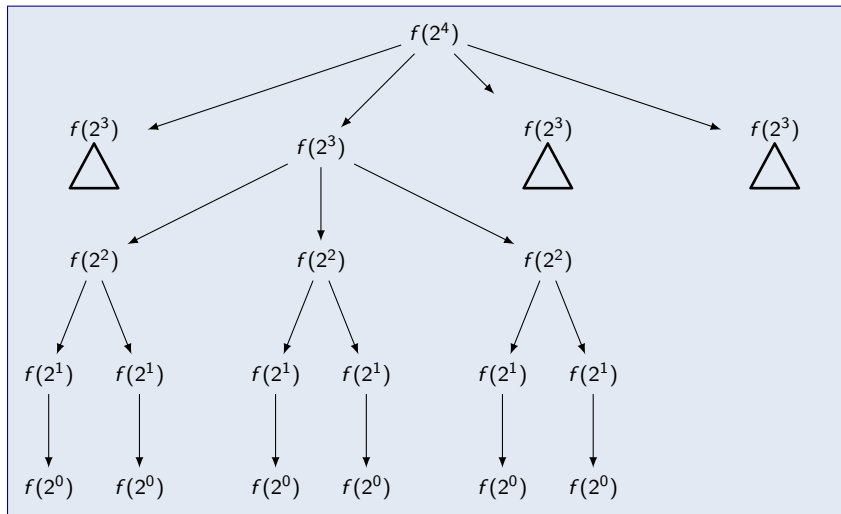
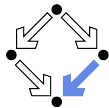
We are going to analyze the following program function:

```
static int f(int m) {
    if (m == 1) return 1;
    int s = 1;
    for (int i=0; i<log2(m); i++)
        s = s+f(m/2);
    return s;
}
```

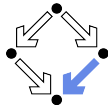
- $f(m)$ calls $f(m/2)$ recursively $\lfloor \log_2 m \rfloor$ times .
 - $f(2^n)$ calls $f(2^{n-1})$ recursively n times.
- How often is f called in total when executing $f(m) = f(2^n)$?
 - Actually, this value is also the result of f .

The analysis of a program involving both loops and recursion.

Recursion Tree



Recursion Tree



Each node in the recursion tree denotes one function call.

- Tree of height 4:

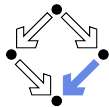
- Level 0: 1 node.
- Level 1: 4 nodes.
- Level 2: $4 \cdot 3$ nodes.
- Level 3: $4 \cdot 3 \cdot 2$ nodes.
- Level 4: $4 \cdot 3 \cdot 2 \cdot 1$ nodes.

- Total number of nodes (function calls):

$$1 + 4 + 4 \cdot 3 + 4 \cdot 3 \cdot 2 + 4 \cdot 3 \cdot 2 \cdot 1 = 65$$

What is the number of nodes/function calls $T(n)$ for $f(2^n)$?

A Recurrence

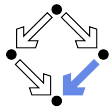


From the code of f , the following recurrence defines $T(n)$.

$$T(n) := \begin{cases} 1 & \text{if } n = 0 \\ 1 + n \cdot T(n-1) & \text{else} \end{cases}$$

- $m = 2^0 = 1$: 1 function call.
- $m = 2^n > 1$: $1 + n \cdot T(n-1)$ function calls.

We need an explicit solution of this recurrence.



Solving the Recurrence

Again we add the number of nodes in each level of the tree.

$$T(n) \stackrel{?}{=} 1 + n + n \cdot (n-1) + n \cdot (n-1) \cdot (n-2) + \dots + n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1$$

$$T(n) \stackrel{?}{=} \sum_{i=0}^n \frac{n!}{i!}$$

$$n!/n! = 1$$

$$n!/(n-1)! = n$$

$$n!/(n-2)! = n \cdot (n-1)$$

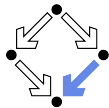
$$n!/(n-3)! = n \cdot (n-1) \cdot (n-2)$$

...

$$n!/0! = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1$$

We need to verify that this is indeed a valid solution of the recurrence.

Verifying the Solution



We prove $(\forall n \in \mathbb{N} : T(n) = \sum_{i=0}^n \frac{n!}{i!})$ by induction on n .

- **Induction base:**

$$T(0) = 1 = 0!/0! = \sum_{i=0}^0 \frac{0!}{i!}$$

- **Induction hypothesis:** we assume

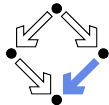
$$T(n) = \sum_{i=0}^n \frac{n!}{i!}$$

- **Induction step:** we prove

$$T(n+1) = \sum_{i=0}^{n+1} \frac{(n+1)!}{i!}$$

$$\begin{aligned} T(n+1) &= 1 + (n+1) \cdot T(n) = 1 + (n+1) \cdot \sum_{i=0}^n \frac{n!}{i!} = 1 + \sum_{i=0}^n \frac{(n+1) \cdot n!}{i!} \\ &= 1 + \sum_{i=0}^n \frac{(n+1)!}{i!} = \frac{(n+1)!}{(n+1)!} + \sum_{i=0}^n \frac{(n+1)!}{i!} = \sum_{i=0}^{n+1} \frac{(n+1)!}{i!} \quad \square \end{aligned}$$

Asymptotic Characterization of the Solution



The explicit solution does not give much intuition about its growth.

$$T(n) = \sum_{i=0}^n \frac{n!}{i!} = n! \cdot \sum_{i=0}^n \frac{1}{i!} < n! \cdot \sum_{i=0}^{\infty} \frac{1}{i!} = n! \cdot e = O(n!)$$

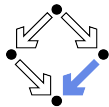
$$\sum_{i=0}^{\infty} 1/i! = e \text{ (Euler's number).}$$

The number of function calls is $O(n!) = O((\log_2 m)!)$.



-
1. Example
 - 2. Sums**
 3. Recurrences
 4. Divide and Conquer
 5. Randomization
 6. Amortized Analysis

Sums



Sort integer array $a[0 \dots n-1]$ of length $n \geq 1$ in ascending order.

```
procedure INSERTIONSORT( $a$ )
```

```
   $n \leftarrow \text{length}(a)$ 
```

```
  for  $i$  from 1 to  $n-1$  do
```

```
     $x \leftarrow a[i]$ 
```

```
     $j \leftarrow i-1$ 
```

```
    while  $j \geq 0 \wedge a[j] > x$  do
```

```
       $a[j+1] \leftarrow a[j]$ 
```

```
       $j \leftarrow j-1$ 
```

```
    end while
```

```
     $a[j+1] \leftarrow x$ 
```

```
  end for
```

```
end procedure
```

Cost

1

n

$n-1$

$n-1$

$\sum_{i=1}^{n-1} n_i$

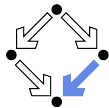
$\sum_{i=1}^{n-1} (n_i - 1)$

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

$n-1$

Sums arise from the analysis of iterative algorithms.

Worst Case Time Complexity



$n_i = i + 1$: maximum number of times **while** test is executed for value i .

- **Worst case time complexity:** $T(n) = 4n - 2 + \sum_{i=1}^{n-1} (3i + 1)$

$$\begin{aligned} T(n) &= 1 + n + (n-1) + (n-1) + (n-1) + \\ &\quad \left(\sum_{i=1}^{n-1} n_i \right) + \left(\sum_{i=1}^{n-1} (n_i - 1) \right) + \left(\sum_{i=1}^{n-1} (n_i - 1) \right) = 4n - 2 + \sum_{i=1}^{n-1} (3n_i - 2) \\ &= 4n - 2 + \sum_{i=1}^{n-1} (3 \cdot (i+1) - 2) = 4n - 2 + \sum_{i=1}^{n-1} (3i + 1) \end{aligned}$$

- **Closed form:** $\sum_{i=1}^{n-1} (3i + 1) = \frac{3n^2 - n - 2}{2}$

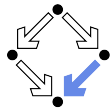
$$\sum_{i=1}^{n-1} (3i + 1) = \sum_{i=0}^{n-1} (3i + 1) - 1 = n \cdot 1 + 3 \cdot \frac{(n-1) \cdot n}{2} - 1 = \frac{3n^2 - n - 2}{2}$$

- **Arithmetic series:** $\sum_{i=0}^n (a + i \cdot d) = (n+1) \cdot a + d \cdot \frac{n \cdot (n+1)}{2}$
- **Geometric series:** $\sum_{i=0}^n (a \cdot q^i) = a \cdot \frac{q^{n+1} - 1}{q - 1}$

High school knowledge.

Worst case time complexity $T(n) = 4n - 2 + \frac{3n^2 - n - 2}{2} = \frac{3n^2 + 7n - 6}{2}$.

Average Time Complexity



Maximum value n_i is replaced by expected value $E[N_i]$.

- Expected value of random variable N_i : $E[N_i] = \frac{i+2}{2}$
 - Assume that all permutations of a have equal probability.
 - Consequently each value $1, \dots, i+1$ of N_i has equal probability.

$$E[N_i] = \frac{1}{i+1} \cdot \sum_{j=1}^{i+1} j = \frac{(i+2) \cdot (i+1)}{2 \cdot (i+1)} = \frac{i+2}{2}$$

- Average time complexity: $\bar{T}(n) = 4n - 2 + \frac{1}{2} \cdot \sum_{i=1}^{n-1} (3i + 2)$

$$\bar{T}(n) = 4n - 2 + \sum_{i=1}^{n-1} (3 \cdot \frac{i+2}{2} - 2) = 4n - 2 + \frac{1}{2} \cdot \sum_{i=1}^{n-1} (3i + 2)$$

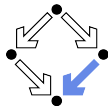
- Closed form: $\bar{T}(n) = \frac{3n^2 + 17n - 12}{4}$

$$\sum_{i=1}^{n-1} (3i + 2) = \sum_{i=0}^{n-1} (3i + 2) - 2 = (2n + 3 \cdot \frac{(n-1) \cdot n}{2}) - 2 = \frac{3n^2 + n - 4}{2}$$

$$\bar{T}(n) = 4n - 2 + \frac{3n^2 + n - 4}{4} = \frac{16n - 8 + 3n^2 + n - 4}{4} = \frac{3n^2 + 17n - 12}{4}$$

Average time complexity $\bar{T}(n) = \frac{3n^2 + 17n - 12}{4}$.

Asymptotic Time Complexity



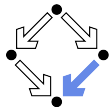
Worst case/average time $T(n) = \frac{3n^2+7n-6}{2}$ and $\bar{T}(n) = \frac{3n^2+17n-12}{4}$.

- $\bar{T}(n) \simeq \frac{T(n)}{2}$ (for large n)
 - In the average, the algorithm is twice as fast as in the worst case.
- $\bar{T}(n) = T(n) = \Theta(n^2)$
 - Asymptotic complexity is the same in the average as in the worst case.
- **Asymptotic estimation:** $\sum_{i=0}^{\Theta(n)} \Theta(i) = \Theta(n^2)$
 - a linear number of times linear complexity gives quadratic complexity.

$$\sum_{i=0}^{\Theta(n)} \Theta(i^k) = \Theta(n^{k+1})$$

Frequently, a quick estimation of asymptotic time complexity is possible.

Solving Sums by Guessing and Verifying



One may consult an (electronic/printed) table of integer sequences.

- Determine summation values for growing number of summands:

$$0, 4, 4 + 7, 4 + 7 + 10, 4 + 7 + 10 + 13, \dots = 0, 4, 11, 21, 34, \dots$$

- On-line Encyclopedia of Integer Sequences (<http://oeis.org>)

[login](#)

This site is supported by donations to [The OEIS Foundation](#).

[Hints](#)

(Greetings from [The On-Line Encyclopedia of Integer Sequences!](#))

Search: **seq:0,4,11,21,34**

Displaying 1-1 of 1 result found.

page 1

Sort: [relevance](#) | [references](#) | [number](#) | [modified](#) | [created](#) Format: [long](#) | [short](#) | [data](#)

[A115067](#)

$(3 \cdot n^2 - n) / 2$.

+20
18

0, 4, 11, 21, 34, 50, 69, 91, 116, 144, 175, 209, 246, 286, 329, 375, 424, 476, 531, 589, 650, 714, 781, 851, 924, 1000, 1079, 1161, 1246, 1334, 1425, 1519, 1616, 1716, 1819, 1925, 2034, 2146, 2261, 2379, 2500, 2624, 2751, 2881, 3014, 3150, 3289, 3431, 3576 ([list](#); [graph](#); [refs](#); [listen](#); [history](#); [text](#); [internal format](#))

OFFSET 1,2

LINKS

[Table of n, a\(n\) for n=1..49.](#)

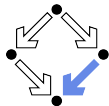
Alfred Hoehn, [Illustration of initial terms of A000326, A005449, A045943, A115067](#) [temporary remark: the case n=4 appears to be incorrect in the illustration]

[Index entries for sequences related to linear recurrences with constant coefficients](#), signature (3,-3,1).

FORMULA

$a(n) = (3 \cdot n + 2) \cdot (n - 1) / 2$.

Solving Sums by Guessing and Verifying



One may consult a computer algebra system (Maple, Mathematica, ...).

```
> sum(3*i+1,i=1..n-1);
```

$$\frac{3}{2} n^2 - \frac{1}{2} n - 1$$

```
In[1]:= Sum[3*i+1,{i,1,n-1}]
```

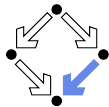
$$\text{Out [1]} = \frac{-2 - n + 3 n^2}{2}$$

```
In[2]:= FindSequenceFunction[{0,4,11,21,34},n]
```

$$\text{Out [2]} = \frac{(-1 + n) (2 + 3 n)}{2}$$

However the solution was initially *guessed*, it must be subsequently *verified*.

Solving Sums by Guessing and Verifying



Prove $\forall n \in \mathbb{N} : n \geq 1 \Rightarrow \sum_{i=1}^{n-1} (3i+1) = \frac{3n^2-n-2}{2}$ by induction on n .

- Base case $n = 1$:

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

- We assume for fixed $n \geq 1$ $\sum_{i=1}^{n-1} (3i+1) = \frac{3n^2-n-2}{2}$ and show

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

This equation holds, because we have

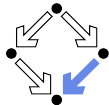
$$\begin{aligned} \sum_{i=1}^n (3i+1) &= \sum_{i=1}^{n-1} (3i+1) + (3n+1) = \frac{3n^2-n-2}{2} + (3n+1) \\ &= \frac{3n^2-n-2+6n+2}{2} = \frac{3n^2+5n}{2} \end{aligned}$$

$$\frac{3 \cdot (n+1)^2 - (n+1) - 2}{2} = \frac{3n^2+6n+3-n-1-2}{2} = \frac{3n^2+5n}{2} \quad \square$$



-
1. Example
 2. Sums
 - 3. Recurrences**
 4. Divide and Conquer
 5. Randomization
 6. Amortized Analysis

Recurrences



Find in sorted array $a[l, r]$ the position of value x (-1 , if x does not occur).

```
function BINARYSEARCH( $a, x, l, r$ )  $\triangleright n = r - l + 1$ 
  if  $l > r$  then
    return  $-1$ 
  end if
   $m \leftarrow \lfloor \frac{l+r}{2} \rfloor$ 
  if  $a[m] = x$  then
    return  $m$ 
  else if  $a[m] < x$  then
    return BINARYSEARCH( $a, x, m + 1, r$ )
  else
    return BINARYSEARCH( $a, x, l, m - 1$ )
  end if
end function
```

Cost

1

1

1

1

1

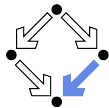
1

$\leq 1 + T(\lfloor \frac{n}{2} \rfloor)$

$\leq 1 + T(\lfloor \frac{n}{2} \rfloor)$

Recurrences arise from the analysis of recursive algorithms.

Worst Case Time Complexity



- Recurrence Relation:

$$T(0) = 2$$

$$T(n) = 5 + T(\lfloor \frac{n}{2} \rfloor), \text{ if } n \geq 1$$

- Special solution: assume $n = 2^m$.

$$\begin{aligned} T(2^m) &= 5 + T(2^{m-1}) \\ &= \underbrace{5 + \dots + 5}_{m \text{ times}} + T(1) = \underbrace{5 + \dots + 5}_{m+1 \text{ times}} + T(0) \\ &= 5 \cdot (m+1) + 2 = 5m + 7 \end{aligned}$$

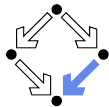
- General solution: for all $n \geq 1$.

$$T(n) = 5 \cdot \lfloor \log_2 n \rfloor + 7$$

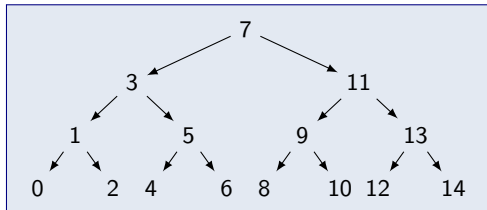
- Verified later.

Worst case time complexity $T(n) = 5 \cdot \lfloor \log_2 n \rfloor + 7$
($T_{\text{found}}(n) = T(n) - 5 = 5 \cdot \lfloor \log_2 n \rfloor + 2$, if x occurs in a).

Average Time Complexity



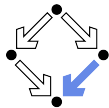
For $n = 2^m - 1$, recursion tree of height $m - 1$ with n nodes:



- Each path describes the function calls to find a particular element:
 - $7 \rightarrow 3 \rightarrow 5$: 3 function calls to find element at position 5.
- In total $n = 2^m - 1$ paths:
 - 1 path of length 0, 2 of length 1, 4 of length 2, \dots , 2^i of length i .
- Assume all paths are equally likely.
 - I.e., assume x occurs in a , at every position with equal probability.

If x is in a , average number of calls is $\frac{1}{2^m - 1} \cdot \sum_{i=0}^{m-1} i \cdot 2^i$.

Average Time Complexity



Determine the closed form of $\sum_{i=0}^{m-1} i \cdot 2^i$.

- **Integer sequence:** 0, 2, 10, 34, 98, ...

$$\begin{array}{l} \text{A036799} \qquad \qquad \qquad 2+2^{(n+1)} \cdot (n-1). \\ 0, 2, 10, 34, 98, \dots \end{array}$$

- **Computer algebra system:**

```
> sum(i*2^i,i=0..m-1);
```

$$m \cdot 2^m - 2^m + 2$$

- **Result:** $\sum_{i=0}^{m-1} i \cdot 2^i = 2^m \cdot (m-2) + 2$

- Verification by induction proof.

- **Average number of function calls:**

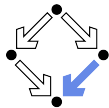
$$\frac{1}{2^m - 1} \cdot \sum_{i=0}^{m-1} i \cdot 2^i = \frac{1}{2^m - 1} \cdot (2^m \cdot (m-2) + 2) = \frac{2^m \cdot (m-2)}{2^m - 1} + \frac{2}{2^m - 1} \simeq m-2$$

- **Average time complexity:**

$$\bar{T}_{\text{found}}(2^m - 1) \simeq 5 \cdot (m-2) + 2 = 5m - 8 = T_{\text{found}}(2^m - 1) - 5$$

One recursive call less in the average case (similar, if x is not in a).

Solving Recurrences by Guessing & Verifying



$$T(1) = 7$$

$$T(n) = 5 + T(\lfloor \frac{n}{2} \rfloor), \text{ if } n > 1$$

One may consult an (electronic/printed) table of integer sequences.

- Simplified recurrence:

$$U(1) = 0$$

$$U(n) = 1 + U(\lfloor \frac{n}{2} \rfloor), \text{ if } n > 1$$

- Integer sequence: 0, 1, 1, 2, 2, 2, 2, 3, ...

A000523 `Log_2(n)` rounded down.

0, 1, 1, 2, 2, 2, 2, 3, ...

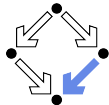
...

FORMULA ... `a(n)=floor(lb(n))`.

- $U(n) = \lfloor \log_2 n \rfloor$, $T(n) = a \cdot \lfloor \log_2 n \rfloor + b$

Closed form $T(n) = 5 \cdot \lfloor \log_2 n \rfloor + 7$.

Solving Recurrences by Guessing & Verifying



One may consult a computer algebra system.

```
> rsolve({T(1)=7,T(n)=5+T(n/2)},T(n));
```

$$\frac{7 \ln(2) + 5 \ln(n)}{\ln(2)}$$

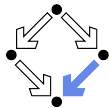
```
In[3] := RSolve[{T[1]==7,T[n]==5+T[n/2]},T[n],n]
```

```
Out[3]= {{T[n] -> 7 +  $\frac{5 \text{Log}[n]}{\text{Log}[2]}$ }}
```

- Real solution $T(n) = 5 \cdot \log_2(n) + 7$.
- Integer solution $T(n) = 5 \cdot \lfloor \log_2(n) \rfloor + 7$.

However the solution was initially *guessed*, it must be subsequently *verified*.

Solving Recurrences by Guessing & Verifying



$$T(1) = 7$$

$$T(n) = 5 + T(\lfloor \frac{n}{2} \rfloor), \text{ if } n > 1$$

We show for all $n \in \mathbb{N}$ with $n \geq 1$, $T(n) = 5 \cdot \lfloor \log_2 n \rfloor + 7$.

- Induction base $n = 1$:

$$5 \cdot \lfloor \log_2 1 \rfloor + 7 = 5 \cdot 0 + 7 = 7 = T(1)$$

- Ind. hypothesis: for $n > 1$ and $1 \leq m < n$, assume $T(m) = 5 \cdot \lfloor \log_2 m \rfloor + 2$.

- Case $n = 2m$:

$$\begin{aligned} T(n) &= 5 + T(\lfloor \frac{n}{2} \rfloor) = 5 + T(m) = 5 + (5 \cdot \lfloor \log_2 m \rfloor + 2) = 5 \cdot (1 + \lfloor \log_2 m \rfloor) + 2 \\ &= 5 \cdot \lfloor 1 + \log_2 m \rfloor + 2 = 5 \cdot \lfloor \log_2 2 + \log_2 m \rfloor + 2 = 5 \cdot \lfloor \log_2 2m \rfloor + 2 \\ &= 5 \cdot \lfloor \log_2 n \rfloor + 2 \end{aligned}$$

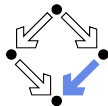
- Case $n = 2m + 1$:

$$\begin{aligned} T(n) &= 5 + T(\lfloor \frac{n}{2} \rfloor) = 5 + T(m) = 5 + (5 \cdot \lfloor \log_2 m \rfloor + 2) = 5 \cdot (1 + \lfloor \log_2 m \rfloor) + 2 \\ &= 5 \cdot \lfloor 1 + \log_2 m \rfloor + 2 = 5 \cdot \lfloor \log_2 2 + \log_2 m \rfloor + 2 = 5 \cdot \lfloor \log_2 2m \rfloor + 2 \\ &= 5 \cdot \lfloor \log_2(n-1) \rfloor + 2 = 5 \cdot \lfloor \log_2 n \rfloor + 2 \end{aligned}$$



-
1. Example
 2. Sums
 3. Recurrences
 - 4. Divide and Conquer**
 5. Randomization
 6. Amortized Analysis

Divide and Conquer



We are going to analyze the Mergesort algorithm.

```
procedure MERGESORT( $a, l, r$ )  ▷  $n = r - l + 1$ 
  if  $l < r$  then
     $m \leftarrow \lfloor \frac{l+r}{2} \rfloor$ 
    MERGESORT( $a, l, m$ )
    MERGESORT( $a, m+1, r$ )
    MERGE( $a, l, m, r$ )
  end if
end procedure
```

Cost

1

1

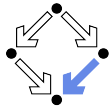
$1 + T(\lfloor \frac{n}{2} \rfloor)$

$1 + T(\lfloor \frac{n}{2} \rfloor)$

$O(n)$

We will investigate the asymptotic time complexity only.

Recurrence



- $T(1)$ can be solved in $O(1)$.

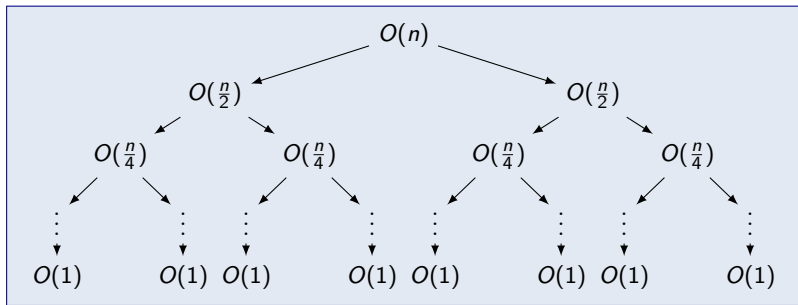
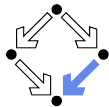
$$\begin{aligned}T(n) &= 1 + 1 + 1 + T(\lceil \frac{n}{2} \rceil) + 1 + T(\lfloor \frac{n}{2} \rfloor) + O(n) \\ &= T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + O(n) + 4 \\ &= T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + O(n)\end{aligned}$$

- For asymptotic analysis, it suffices to consider $\frac{n}{2} \in \{\lfloor \frac{n}{2} \rfloor, \lceil \frac{n}{2} \rceil\}$.

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

We are going to guess an asymptotic solution of this recurrence.

Guessing the Asymptotic Time Complexity

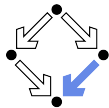


- Execution is the depth-first left-to-right traversal of a binary tree.
- Total time is the sum of the times spent in all tree nodes.

$$T(n) = \sum_{i=0}^{O(\log n)} 2^i \cdot O\left(\frac{n}{2^i}\right) = \sum_{i=0}^{O(\log n)} O(n) = O(n \cdot \log n)$$

We are going to verify our guess of the asymptotic solution.

Verifying the Asymptotic Time Complexity



We prove that every solution $T(n)$ of the recurrence

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

is asymptotically bound by $T(n) = O(n \cdot \log n)$.

- From recurrence, there exist $c > 0$ and $N \geq 1$ such that, for all $n \geq N$

$$T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + c \cdot n$$

- From the definition of $O(n \cdot \log n) = O(n \cdot \log_2 n)$, it suffices to show

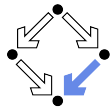
$$\exists c \in \mathbb{R}_{\geq 0}, N \in \mathbb{N} : \forall n \geq N : T(n) \leq c \cdot n \cdot \log_2 n$$

- Thus our goal is to find c' and N' for which we are able to prove

$$\forall n \geq N' : T(n) \leq c' \cdot n \cdot \log_2 n$$

Because of the characterization of T by a recurrence, this naturally leads to an induction proof with base N' .

Verifying Asymptotic Complexity (Contd)

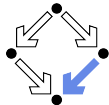


How to choose N' ?

- Clearly $N' \geq \max\{2, N\}$.
 - $n = 1 \Rightarrow c' \cdot n \cdot \log_2 n = 0$
 - $N' < N$: induction hypothesis cannot be applied.
- But $N' \geq 2$ is problematic for recurrence $T(n) = \dots T(\frac{n}{2}) \dots$.
 - Induction step proves goal for n based on assumption it holds for $\frac{n}{2}$.
 - $N' = 1$: every sequence of divisions by 2 which starts with $n \geq 1$ eventually leads to the base case $n = 1$.
 - $N' \geq 2$: sequence may bypass base case $n = N'$, e.g., $(N' + 1)/2 < N'$.
- Solution: show induction base for all n with $2 \leq n \leq N'$.
 - Also ensure that every sequence of divisions starting with $n > N'$ eventually reaches some base case, i.e., $\frac{n}{2} \geq 2$.

We choose $N' := \max\{3, N\}$.

Verifying Asymptotic Complexity (Contd)

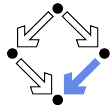


How to choose c' ?

- Have to prove $T(n) \leq c' \cdot n \cdot \log_2 n$.
 - for all $2 \leq n \leq N'$.
- It suffices to prove $T(n) \leq c'$.
 - $n \cdot \log_2 n \geq 1$, for all $2 \leq n \leq N'$.
- It suffices to prove $T(N') \leq c'$
 - $T(2) \leq T(3) \leq \dots \leq T(N')$

We choose $c' := \max\{\lfloor \cdot \rfloor, T(N')\}$.

Verifying Asymptotic Complexity (Contd)



- **Induction base:** by choice of N' and c' , we have for all $2 \leq n \leq N'$

$$T(n) \leq c' \cdot n \cdot \log_2 n$$

- **Induction assumption:** for $n > N'$ and $2 \leq m < n$, we assume

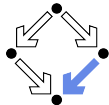
$$T(m) \leq c' \cdot m \cdot \log_2 m$$

- **Induction step:** we show

$$T(n) \leq c' \cdot n \cdot \log_2 n$$

We continue by case distinction on the interpretation of $\frac{n}{2}$.

Verifying Asymptotic Complexity (Contd)

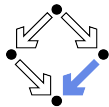


- Case $\frac{n}{2}$ means $\lfloor \frac{n}{2} \rfloor$: since $n > N' \geq N$,

$$\begin{aligned} T(n) &\leq 2 \cdot T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + c \cdot n \\ &\leq 2 \cdot c' \cdot \left\lfloor \frac{n}{2} \right\rfloor \cdot \log_2 \left\lfloor \frac{n}{2} \right\rfloor + c \cdot n \\ &\leq 2 \cdot c' \cdot \frac{n}{2} \cdot \log_2 \left(\frac{n}{2}\right) + c \cdot n \\ &= c' \cdot n \cdot \log_2 \left(\frac{n}{2}\right) + c \cdot n \\ &= c' \cdot n \cdot ((\log_2 n) - 1) + c \cdot n \\ &= c' \cdot n \cdot (\log_2 n) + (c - c') \cdot n \leq c' \cdot n \cdot \log_2 n \end{aligned}$$

Last inequality holds if $c' \geq c$, because then $c - c' \leq 0$.

Verifying Asymptotic Complexity (Contd)

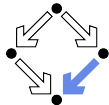


- Case $\frac{n}{2}$ means $\lceil \frac{n}{2} \rceil$: since $n > N' \geq N$,

$$\begin{aligned} T(n) &\leq 2 \cdot T\left(\lceil \frac{n}{2} \rceil\right) + c \cdot n \\ &\leq 2 \cdot c' \cdot \lceil \frac{n}{2} \rceil \cdot \log_2 \lceil \frac{n}{2} \rceil + c \cdot n \\ &\leq c' \cdot (n+1) \cdot \log_2 \left(\frac{n+1}{2}\right) + c \cdot n \\ &= c' \cdot (n+1) \cdot (\log_2(n+1) - 1) + c \cdot n \\ &\leq c' \cdot (n+1) \cdot \left(\log_2 n + \frac{1}{2} - 1\right) + c \cdot n \end{aligned}$$

$\log_2(n+1) \leq (\log_2 n) + \frac{1}{2}$ holds for all $n > 2$ and thus for all $n \geq N'$.

Verifying Asymptotic Complexity (Contd)

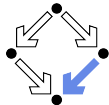


We continue the derivation.

$$\begin{aligned}T(n) &\leq c' \cdot (n+1) \cdot \left((\log_2 n) + \frac{1}{2} - 1 \right) + c \cdot n \\&= c' \cdot (n+1) \cdot \left((\log_2 n) - \frac{1}{2} \right) + c \cdot n \\&= c' \cdot n \cdot (\log_2 n) + \frac{c'}{2} \cdot (2 \cdot (\log_2 n) - n - 1) + c \cdot n \\&\leq c' \cdot n \cdot (\log_2 n) + \frac{c'}{2} \cdot \left(2 \cdot \frac{n}{4} - n - 1 \right) + c \cdot n\end{aligned}$$

Last inequality holds if $\log_2 n \leq \frac{n}{4}$, which holds for all $n \geq 16$.

Verifying Asymptotic Complexity (Contd)



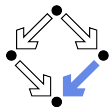
We continue the derivation.

$$\begin{aligned}T(n) &\leq c' \cdot n \cdot (\log_2 n) + \frac{c'}{2} \cdot \left(2 \cdot \frac{n}{4} - n - 1\right) + c \cdot n \\&\leq c' \cdot n \cdot (\log_2 n) - \frac{c' \cdot n}{4} + c \cdot n \\&= c' \cdot n \cdot (\log_2 n) + \frac{4c - c'}{4} \cdot n \leq c' \cdot n \cdot \log_2 n\end{aligned}$$

Last inequality holds if $c' \geq 4c$, because then $4c - c' \leq 0$.

In both cases $N' := \max\{16, N\}$, $c' := \max\{4c, T(N')\}$ lets proof succeed.

The Master Theorem



Let $a \geq 1$, $b > 1$, $f : \mathbb{N} \rightarrow \mathbb{N}$, $T : \mathbb{N} \rightarrow \mathbb{N}$ satisfying the recurrence:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$

- $f(n) = O(n^{(\log_b a) - \varepsilon})$ for some $\varepsilon > 0$:

$$T(n) = \Theta(n^{\log_b a})$$

- $f(n) = \Theta(n^{\log_b a})$:

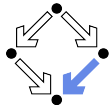
$$T(n) = \Theta(n^{\log_b a} \cdot \log n)$$

- $f(n) = \Omega(n^{(\log_b a) + \varepsilon})$ for some $\varepsilon > 0$ and there exist some c with $0 < c < 1$ and some $N \in \mathbb{N}$ such that $\forall n \geq N : a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n)$:

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

Easy analysis of a large class of divide and conquer algorithms.

Example



Analysis of MERGESORT.

■ Recurrence:

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

■ Case 2 of Master Theorem ($a = b = 2$):

$$\log_b a = 1$$

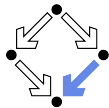
$$\Theta(n) = \Theta(n^1) = \Theta(n^{\log_b a})$$

■ Solution:

$$T(n) = \Theta(n^{\log_b a} \cdot \log n) = \Theta(n^1 \cdot \log n) = \Theta(n \cdot \log n)$$

No tedious proof required any more.

Arbitrary Precision Multiplication



Multiply two natural numbers a and b with n digits each.

```
function MULTIPLY( $a, b$ )
```

```
   $n \leftarrow \text{digits}(a) \triangleright \text{digits}(a) = \text{digits}(b)$ 
```

```
   $c \leftarrow 0$ 
```

```
  for  $i$  from  $n - 1$  to  $0$  do
```

```
     $p \leftarrow \text{MULTIPLYDIGIT}(a, b_i)$ 
```

```
     $c \leftarrow \text{SHIFT}(c, 1)$ 
```

```
     $c \leftarrow \text{ADD}(p, c)$ 
```

```
  end for
```

```
  return  $c$ 
```

```
end function
```

Cost

1

1

$n + 1$

$n \cdot \Theta(n)$

$\sum_{i=0}^{n-1} \Theta(2n - i - 1)$

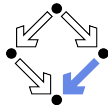
$n \cdot \Theta(n)$

1

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

Classical (“school”) algorithm has quadratic time complexity.

Arbitrary Precision Multiplication



Split a and b into halves (a', a'') and (b', b'') of $\frac{n}{2}$ digits.

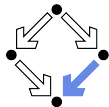
$$a = a' \cdot d^{\frac{n}{2}} + a''$$

$$b = b' \cdot d^{\frac{n}{2}} + b''$$

$$\begin{aligned} a \cdot b &= (a' \cdot d^{\frac{n}{2}} + a'') \cdot (b' \cdot d^{\frac{n}{2}} + b'') \\ &= a' \cdot b' \cdot d^n + (a' \cdot b'' + a'' \cdot b') \cdot d^{\frac{n}{2}} + a'' \cdot b'' \end{aligned}$$

Basis of a recursive multiplication algorithm.

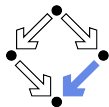
Arbitrary Precision Multiplication (Rec.)



function MULTIPLY(a, b)	Cost
$n \leftarrow \text{digits}(a) \quad \triangleright \text{digits}(a) = \text{digits}(b) = 2^m$	1
if $n = 1$ then	1
$c \leftarrow \text{MULTIPLYDIGIT}(a_0, b_0)$	$\Theta(1)$
else	$\Theta(n)$
$a' \leftarrow a_{\frac{n}{2} \dots n-1}; a'' \leftarrow a_{0 \dots \frac{n}{2}-1}$	$\Theta(n)$
$b' \leftarrow b_{\frac{n}{2} \dots n-1}; b'' \leftarrow b_{0 \dots \frac{n}{2}-1}$	$1 + T(\frac{n}{2})$
$u \leftarrow \text{MULTIPLY}(a', b')$	$1 + T(\frac{n}{2})$
$v \leftarrow \text{MULTIPLY}(a', b'')$	$1 + T(\frac{n}{2})$
$w \leftarrow \text{MULTIPLY}(a'', b')$	$1 + T(\frac{n}{2})$
$x \leftarrow \text{MULTIPLY}(a'', b'')$	$\Theta(n)$
$y \leftarrow \text{ADD}(v, w)$	$\Theta(n)$
$y \leftarrow \text{SHIFT}(y, \frac{n}{2})$	$\Theta(n)$
$c \leftarrow \text{SHIFT}(u, n)$	$\Theta(n)$
$c \leftarrow \text{ADD}(c, y)$	$\Theta(n)$
$c \leftarrow \text{ADD}(c, x)$	$\Theta(n)$
end if	
return c	1
end function	

Four recursive calls of the algorithm with half the input size.

Arbitrary Precision Multiplication (Rec.)



Analysis of recursive algorithm by the Master Theorem.

■ Recurrence:

$$\begin{aligned}T(n) &= 3 + 4 \cdot \left(1 + T\left(\frac{n}{2}\right)\right) + 7 \cdot \Theta(n) \\ &= 4 \cdot T\left(\frac{n}{2}\right) + \Theta(n)\end{aligned}$$

■ Case 1 of the master theorem ($a = 4, b = 2$):

$$(\log_b a) = (\log_2 4) = 2$$

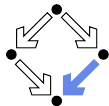
$$f(n) = O(n) = O(n^1) = O(n^{2-1}) = O(n^{(\log_b a)-1})$$

■ Solution:

$$T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$$

Also the recursive algorithm has quadratic time complexity.

Arbitrary Precision Multiplication



Anatolii Karatsuba and Yuri Ofman, 1962.

$$\begin{aligned}a \cdot b &= (a' \cdot d^{\frac{n}{2}} + a'') \cdot (b' \cdot d^{\frac{n}{2}} + b'') \\&= a' \cdot b' \cdot d^n + (a' \cdot b'' + a'' \cdot b') \cdot d^{\frac{n}{2}} + a'' \cdot b'' \\&= a' \cdot b' \cdot d^n + ((a' + a'') \cdot (b' + b'') - a' \cdot b' - a'' \cdot b'') \cdot d^{\frac{n}{2}} + a'' \cdot b''\end{aligned}$$

- Two multiplications $a' \cdot b'$ and $a'' \cdot b''$ of numbers with $\frac{n}{2}$ digits.
- Product $s \cdot t$ where $s = a' + a''$ and $t = b' + b''$ may have $\frac{n}{2} + 1$ digits.

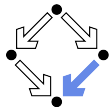
$$s = s_{\frac{n}{2}} \cdot d^{\frac{n}{2}} + s', t = t_{\frac{n}{2}} \cdot d^{\frac{n}{2}} + t'$$

$$\begin{aligned}s \cdot t &= (s_{\frac{n}{2}} \cdot d^{\frac{n}{2}} + s') \cdot (t_{\frac{n}{2}} \cdot d^{\frac{n}{2}} + t') \\&= s_{\frac{n}{2}} \cdot t_{\frac{n}{2}} \cdot d^n + (s_{\frac{n}{2}} \cdot t' + t_{\frac{n}{2}} \cdot s') \cdot d^{\frac{n}{2}} + s' \cdot t'\end{aligned}$$

- Can compute $s \cdot t$ from product $s' \cdot t'$ of numbers of length $\frac{n}{2}$.

We only need three multiplications of numbers with $\frac{n}{2}$ digits.

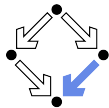
Karatsuba Algorithm



function MULTIPLY(<i>a</i> , <i>b</i>)	Cost
<i>n</i> ← <i>digits</i> (<i>a</i>) ▷ <i>digits</i> (<i>a</i>) = <i>digits</i> (<i>b</i>) = 2^m	1
if <i>n</i> = 1 then	1
<i>c</i> ← MULTIPLYDIGIT(<i>a</i> ₀ , <i>b</i> ₀)	Θ(1)
else	
<i>a'</i> ← <i>a</i> _{$\frac{n}{2} \dots n-1$} ; <i>a''</i> ← <i>a</i> _{0...$\frac{n}{2}-1$}	Θ(<i>n</i>)
<i>b'</i> ← <i>b</i> _{$\frac{n}{2} \dots n-1$} ; <i>b''</i> ← <i>b</i> _{0...$\frac{n}{2}-1$}	Θ(<i>n</i>)
<i>s</i> ← ADD(<i>a'</i> , <i>a''</i>)	Θ(<i>n</i>)
<i>t</i> ← ADD(<i>b'</i> , <i>b''</i>)	Θ(<i>n</i>)
<i>u</i> ← MULTIPLY(<i>a'</i> , <i>b'</i>)	$1 + T(\frac{n}{2})$
<i>v</i> ← MULTIPLY(<i>s</i> , <i>t</i>)	Θ(<i>n</i>) + $T(\frac{n}{2})$
<i>x</i> ← MULTIPLY(<i>a''</i> , <i>b''</i>)	$1 + T(\frac{n}{2})$
<i>y</i> ← SUBTRACT(<i>v</i> , <i>u</i>)	Θ(<i>n</i>)
<i>y</i> ← SUBTRACT(<i>v</i> , <i>x</i>)	Θ(<i>n</i>)
<i>y</i> ← SHIFT(<i>y</i> , $\frac{n}{2}$)	Θ(<i>n</i>)
<i>c</i> ← SHIFT(<i>u</i> , <i>n</i>)	Θ(<i>n</i>)
<i>c</i> ← ADD(<i>c</i> , <i>y</i>)	Θ(<i>n</i>)
<i>c</i> ← ADD(<i>c</i> , <i>x</i>)	Θ(<i>n</i>)
end if	
return <i>c</i>	1
end function	

Three recursive calls of the algorithm with half the input size.

Karatsuba Algorithm



Analysis of Karatsuba algorithm by the Master Theorem.

■ **Recurrence:**

$$\begin{aligned}T(n) &= 3 + 2 \cdot \left(1 + T\left(\frac{n}{2}\right)\right) + \left(\Theta(n) + T\left(\frac{n}{2}\right)\right) + 10 \cdot \Theta(n) \\ &= 3 \cdot T\left(\frac{n}{2}\right) + \Theta(n)\end{aligned}$$

■ **Case 1 of Master Theorem ($a = 3, b = 2$):**

$$\log_b a = \log_2 3$$

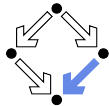
$$f(n) = O(n) = O(n^1) = O(n^{(\log_b a) - \epsilon})$$

$$1.58 < \log_2 3 < 1.59, \epsilon = (\log_2 3) - 1 > 0.58$$

■ **Solution:**

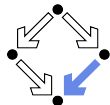
$$T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_2 3}) = o(n^2)$$

The Karatsuba algorithm has a better asymptotic time complexity than the classical algorithm and is thus implemented in computer algebra systems.



-
1. Example
 2. Sums
 3. Recurrences
 4. Divide and Conquer
 - 5. Randomization**
 6. Amortized Analysis

The Quicksort Algorithm

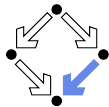


Sort array a in range $[l, r]$ of size $n = r - l + 1$ in ascending order.

procedure QUICKSORT(a, l, r) $\triangleright n = r - l + 1$	Cost
if $l < r$ then	1
choose $p \in [l, r]$	$O(n)$
$m \leftarrow$ PARTITION(a, l, r, p) $\triangleright i = m - l$	$\Theta(n)$
QUICKSORT($a, l, m - 1$)	$1 + T(i)$
QUICKSORT($a, m + 1, r$)	$1 + T(n - i - 1)$
end if	
end procedure	

Two recursive calls with input sizes i and $n - i - 1$ (for some $0 \leq i \leq n - 1$).

Time Complexity of Quicksort



- Recurrence:

$$T(n) = T(i) + T(n - i - 1) + \Theta(n)$$

- One interval is empty ($i = 0$ or $i = n - 1$):

$$T(n) = T(0) + T(n - 1) + \Theta(n) = \Theta(1) + T(n - 1) + \Theta(n) = T(n - 1) + \Theta(n)$$

$$= \sum_{i=0}^{n-1} \Theta(i) = \Theta(n^2)$$

- Unbalanced binary recursion tree where every left child is a leaf and the path from root along every right child to rightmost leaf has length n .

- Both intervals have same size ($i = \frac{n}{2}$):

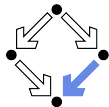
$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + \Theta(n) = 2 \cdot T\left(\frac{n}{2}\right) + \Theta(n)$$

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

- Case 2 of the Master Theorem ($a = b = 2$).
- Balanced binary recursion tree of depth $\log_2 n$.

Worst case time complexity is quadratic; best case is linear-logarithmic.

Average Time Complexity of Quicksort

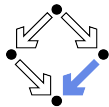


Assume that all n values $0, \dots, n-1$ of i are equally likely.

$$\begin{aligned}T(n) &= \frac{1}{n} \cdot \sum_{i=0}^{n-1} (T(i) + T(n-i-1) + \Theta(n)) \\&= \frac{1}{n} \cdot \left(\sum_{i=0}^{n-1} T(i) + T(n-i-1) \right) + \Theta(n) \\&= \frac{1}{n} \cdot \left(\sum_{i=0}^{n-1} T(i) + \sum_{i=0}^{n-1} T(n-i-1) \right) + \Theta(n) \\&= \frac{1}{n} \cdot \left(\sum_{i=0}^{n-1} T(i) + \sum_{i=0}^{n-1} T(i) \right) + \Theta(n) \\&= \frac{2}{n} \cdot \sum_{i=0}^{n-1} T(i) + \Theta(n)\end{aligned}$$

Is the average time complexity closer to the worst or to the best case?

Average Time Complexity of Quicksort



Consider special form of recurrence (see lecture notes for the general case).

$$T'(n) = \frac{2}{n} \cdot \sum_{i=1}^{n-1} T'(i) + n$$

$$n \cdot T'(n) = 2 \cdot \sum_{i=1}^{n-1} T'(i) + n^2$$

$$(n-1) \cdot T'(n-1) = 2 \cdot \sum_{i=1}^{n-2} T'(i) + (n-1)^2$$

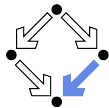
$$n \cdot T'(n) - (n-1) \cdot T'(n-1) = 2 \cdot T'(n-1) + 2n - 1$$

$$n \cdot T'(n) = (n+1) \cdot T'(n-1) + 2n - 1$$

$$\frac{T'(n)}{n+1} = \frac{T'(n-1)}{n} + \frac{2n-1}{n \cdot (n+1)}$$

Terms involving T' have now same shape on left and right side.

Average Time Complexity of Quicksort



We solve the special recurrence.

$$\sum_{i=1}^n \frac{T'(i)}{i+1} = \sum_{i=1}^n \left(\frac{T'(i-1)}{i} + \frac{2i-1}{i \cdot (i+1)} \right)$$

$$\sum_{i=1}^n \frac{T'(i)}{i+1} = \sum_{i=1}^n \frac{T'(i-1)}{i} + \sum_{i=1}^n \frac{2i-1}{i \cdot (i+1)}$$

$$\sum_{i=1}^n \frac{T'(i)}{i+1} = \sum_{i=0}^{n-1} \frac{T'(i)}{i+1} + \sum_{i=1}^n \frac{2i-1}{i \cdot (i+1)}$$

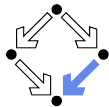
$$\frac{T'(n)}{n+1} = \frac{T'(0)}{1} + \sum_{i=1}^n \frac{2i-1}{i \cdot (i+1)}$$

$$T'(n) = (n+1) \cdot \left(T'(0) + \sum_{i=1}^n \frac{2i-1}{i \cdot (i+1)} \right)$$

$$T'(n) = O\left(n \cdot \sum_{i=1}^n \frac{1}{i}\right) = O(n \cdot H_n) = O(n \cdot \log n)$$

The average time complexity is the same as that of the best case.

Average Time Complexity of Quicksort



We could have also applied a computer algebra system.

```
> rsolve({T(0)=1,T(n)=(n+1)/n*T(n-1)+1},T(n));  
          (n + 1) (Psi(n + 2) + gamma)
```

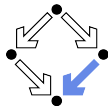
■ Psi(n): $\Psi(n) = H_{n-1} - \gamma$,

```
In[1] := RSolve[{T[0]==1,T[n]==(n+1)/n*T[n-1]+1},T[n],n]  
Out[1]= {{T[n] -> EulerGamma + EulerGamma n +  
          PolyGamma[0, 2 + n] +  
          n PolyGamma[0, 2 + n]}}
```

■ PolyGamma[0,n]: $\Psi(n) = H_{n-1} - \gamma$.

Result is in $O(n \cdot H_n)$.

Ensuring the Average Time Complexity



We have assumed that all values of $i = m - l$ are equally likely but how realistic is this assumption?

- Assumption is satisfied if and only if the choice

`choose $p \in [l, r]$`

determines a pivot element $a[p]$ that is equally likely to be the element at any of the positions l, \dots, r .

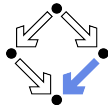
- We might choose a fixed index in interval $[l, r]$, e.g.,

`$p \leftarrow r$`

- But then we get evenly distributed pivot elements only if all $n!$ permutations of a occur with equal probability as inputs.

It is in practice typically hard to estimate how inputs are distributed; worst case situations (input array already sorted) might appear frequently.

Randomization

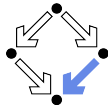


We might shuffle the input array randomly before sorting it.

```
procedure RANDOMIZE(a)  
  n ← length(a)  
  for i from 0 to n − 1 do  
    r ← RANDOM(i, n − 1)  
    b ← a[i]; a[i] ← a[r]; a[r] ← b  
  end for  
end procedure
```

Indeed ensures that all permutations are equally likely but is costly.

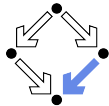
Randomization



We may simply choose the pivot element randomly.

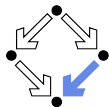
$$p \leftarrow \text{RANDOM}(l, r)$$

In many algorithms, making a random choice (rather than making an arbitrary fixed choice) may yield an average case complexity that is independent of the input distribution.



-
1. Example
 2. Sums
 3. Recurrences
 4. Divide and Conquer
 5. Randomization
 - 6. Amortized Analysis**

Amortized Analysis

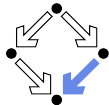


Determine worst-case time complexity $T(n)$ of a sequence of n operations.

- Assume that operations are performed on the same data structure.
 - Then it may be that the worst-case complexity $T_{\text{op}}(i)$ of operation i can be only exhibited for some *few* elements of the sequence.
- $T(n) \leq \sum_{i=1}^n T_{\text{op}}(i)$
 - The worst case for the whole sequence may be smaller than the sum of the worst-cases of each operation.
- $\frac{T(n)}{n} \leq \frac{1}{n} \cdot \sum_{i=1}^n T_{\text{op}}(i)$
 - The contribution of an individual operation to the worst-case complexity of the sequence may be smaller than the average of the individual worst case complexities.
- **Amortized cost** $\frac{T(n)}{n}$: some operations with high costs may be outweighed by many operations with low costs.

Amortized analysis is typically applied to operations that manipulate a certain data structure (e.g., a sequence of method calls on an object).

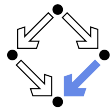
Example



- Consider a **stack** with the following operations:
 - **PUSH**(s, x): push an element x on stack s .
Time $O(1)$.
 - **POP**(s): pop an element from the top of the non-empty stack s .
Time $O(1)$.
 - **MULTIPOP**(s, k): pop k elements from the stack s (if s has $l < k$ elements, then only l elements are popped).
 $O(\min\{l, k\})$ where l is the number of elements on s .
- **Sequence of n operations** can be performed in time $O(n^2)$.
 - Each **MULTIPOP** operation has complexity $O(n)$.
 - The bound is correct but not tight.

We are interested in a much tighter bound.

Aggregate Analysis



- Assume among the n operations, there are k MULTIPOP operations.

$$n = k + \sum_{i=0}^k n_i$$

n_0, n_1, \dots, n_k : number of operations before/after a MULTIPOP.

- The total cost of the sequence is

$$T(n) = \sum_{i=0}^{k-1} O(p_i) + (n - k) \cdot O(1) = O\left(\sum_{i=0}^{k-1} p_i\right) + O(n)$$

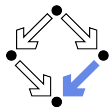
$$\stackrel{(1)}{=} O\left(\sum_{i=0}^{k-1} n_i\right) + O(n) \stackrel{(2)}{=} O(n) + O(n) = O(n)$$

p_i : number of elements popped in call i of MULTIPOP.

- (1) follows from $\sum_{i=0}^{k-1} p_i \leq \sum_{i=0}^{k-1} n_i$.
 - The total number of elements popped from the stack is bound by the total number of previously occurring (PUSH) operations.
- (2) follows from $\sum_{i=0}^{k-1} n_i \leq n$.
 - Number of PUSH operations is bound by number of all operations.
- Sequence of n operations can be performed in time $O(n)$.

The amortized cost of a single operation is $O(1)$, i.e., constant.

The Potential Method



- Assign to operation i its **actual cost** c_i and **amortized cost** \hat{c}_i :
 - If $\hat{c}_i > c_i$, operation i saves $\hat{c}_i - c_i$ “credit”.
 - If $\hat{c}_i < c_i$, operation i uses up $c_i - \hat{c}_i$ credit.
- The **potential function** $\Phi(s)$ maps data structure s to a real number.
 - $\Phi(s)$: the credit accumulated so far (the “potential” of s).

$$\hat{c}_i - c_i = C \cdot (\Phi(s_i) - \Phi(s_{i-1}))$$

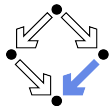
- $\hat{c}_i - c_i$: the credit saved/used by operation i .
 - s_0 : the initial value of s ; s_i : its value after operation i .
 - Constant factor $C \geq 0$.
- Sum of amortized costs:**

$$\sum_{i=1}^n \hat{c}_i = \sum_{i=1}^n (c_i + C \cdot (\Phi(s_i) - \Phi(s_{i-1}))) = \sum_{i=1}^n c_i + C \cdot (\Phi(s_n) - \Phi(s_0))$$

- We can ensure $\sum_{i=1}^n \hat{c}_i \geq \sum_{i=1}^n c_i$ by ensuring $\Phi(s_n) \geq \Phi(s_0)$.

We have $T(n) \leq \sum_{i=1}^n \hat{c}_i$, i.e., can use amortized costs in the analysis.

Example



Stack s with n PUSH, POP, MULTIPOP operations.

- **Potential $\Phi(s)$** : the number of elements in stack s .

$$\Phi(s_n) \geq 0 = \Phi(s_0)$$

- **Amortized cost** $\hat{c}_i = c_i + C \cdot (\Phi(s_i) - \Phi(s_{i-1}))$

$C \geq 0$ upper bound for execution time of PUSH and POP; $C \cdot k'$ upper bound for the execution time of MULTIPOP(s, k).

$$k' = \min\{k, m\}, \quad m = |s|.$$

- PUSH(s, x): $\hat{c}_i \leq C + C \cdot ((m+1) - m) = C + C = 2 \cdot C$.
- POP(s): $\hat{c}_i \leq C + C \cdot ((m-1) - m) = C - C = 0$.
- MULTIPOP(s, k): $\hat{c}_i \leq C \cdot k' + C \cdot ((m - k') - m) = C \cdot k' - C \cdot k' = 0$.

Each operation has amortized cost $O(1)$, a sequence of n operations has worst case time complexity $O(n)$.

Dynamic Tables

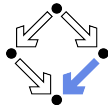
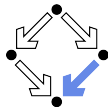


Table t for which a certain amount of space is allocated.

- Operation $\text{INSERT}(t, x)$ inserts value x into t .
 - Sequence of n such operations starting with an empty table.
- If space gets exhausted, t is expanded:
 - More space is allocated and elements are copied from the old space to the new one.
- If n elements are to be copied, time complexity of INSERT is $O(n)$.
 - However, most of the time, there is space available and INSERT can be performed in time $O(1)$.
- Time complexity $O(n^2)$ of a sequence of n INSERT operations.
 - However, this bound is not tight.

We are interested in a much tighter bound.



Aggregate Analysis

How much to expand table of size m ?

- $m+1$: every call of INSERT triggers an expansion.

$$T(n) = 1 + 2 + 3 + \dots + (n-1) = \sum_{i=1}^{n-1} i = O(n^2)$$

- $m+c$: every c -th call triggers an expansion.

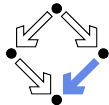
$$\begin{aligned} T(n) &= \sum_{i=1}^{\lceil \frac{n}{c} \rceil} O((i-1) \cdot c) + (n - \lceil \frac{n}{c} \rceil) \cdot O(1) \\ &= O\left(\sum_{i=1}^{\lceil \frac{n}{c} \rceil} i\right) + O(n) = O(n^2) + O(n) = O(n^2) \end{aligned}$$

- $m \cdot c$: every call $c^i + 1$ triggers an expansion (typically $c := 2$).

$$\begin{aligned} T(n) &= \sum_{i=0}^{\lceil \log_2 n \rceil + 1} O(2^i) + (n - \lceil \log_2 n \rceil - 1) \cdot O(1) \\ &= O\left(\sum_{i=0}^{\lceil \log_2 n \rceil + 1} 2^i\right) + O(n) = O\left(\sum_{i=0}^{\lceil \log_2 n \rceil} 2^i\right) + O(n) = O(2n) + O(n) = O(n) \end{aligned}$$

With the last strategy, amortized cost of INSERT is $O(1)$.

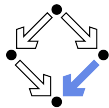
Analysis with the Potential Method



- **Potential:** $\Phi(t) := 2 \cdot \text{num}(t) - \text{size}(t)$
 - $\text{num}(t)$: number of elements in t .
 - $\text{size}(t)$: the number of slots in t .
 - Since $\text{num}(t) \geq \frac{\text{size}(t)}{2}$, we have $\Phi(t) \geq 0$.
- **Change of Potential:**
 - After expansion of t_{i-1} : $\Phi(t_i) = 0$.
 - After insertion into t_{i-1} : $\Phi(t_i) = 2 + \Phi(t_{i-1})$.
 - Before expansion of t_{i-1} : $\Phi(t_{i-1}) = \text{size}(t_{i-1})$

Cost of expansion can be covered by accumulated credit.

Analysis with the Potential Method



- **Amortized cost:** $\hat{c}_i = c_i + C \cdot (\Phi(t_i) - \Phi(t_{i-1}))$.

C : upper bound for the cost of insertion, $C \cdot k$: upper bound for the cost of copying k elements.

- **INSERT does not trigger an expansion:**

$$\text{num}(t_i) = \text{num}(t_{i-1}) + 1, \text{size}(t_i) = \text{size}(t_{i-1}).$$

$$\begin{aligned}\hat{c}_i &\leq C \cdot 1 + C \cdot (2 \cdot \text{num}(t_i) - \text{size}(t_i) - (2 \cdot \text{num}(t_{i-1}) - \text{size}(t_{i-1}))) \\ &= C \cdot 1 + C \cdot (2 \cdot (\text{num}(t_{i-1}) + 1) - \text{size}(t_{i-1}) - (2 \cdot \text{num}(t_{i-1}) - \text{size}(t_{i-1}))) \\ &= C \cdot 1 + C \cdot 2 = 3C\end{aligned}$$

- **INSERT triggers an expansion:**

$$\begin{aligned}\text{num}(t_i) &= \text{num}(t_{i-1}) + 1, \text{size}(t_{i-1}) = \text{num}(t_{i-1}), \\ \text{size}(t_i) &= 2 \cdot \text{size}(t_{i-1}) = 2 \cdot \text{num}(t_{i-1})\end{aligned}$$

$$\begin{aligned}\hat{c}_i &\leq C \cdot \text{num}(t_{i-1}) + C \cdot (2 \cdot \text{num}(t_i) - \text{size}(t_i) - (2 \cdot \text{num}(t_{i-1}) - \text{size}(t_{i-1}))) \\ &= C \cdot \text{num}(t_{i-1}) + C \cdot (2 \cdot (\text{num}(t_{i-1}) + 1) - 2 \cdot \text{num}(t_{i-1}) - (2 \cdot \text{num}(t_{i-1}) - \text{num}(t_{i-1}))) \\ &= C \cdot \text{num}(t_{i-1}) + C \cdot (2 - \text{num}(t_{i-1})) = 2C\end{aligned}$$

In average, we can perform INSERT in time $O(1)$.