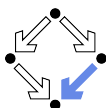# Analysis of Complexity

Wolfgang Schreiner
Wolfgang.Schreiner@risc.jku.at

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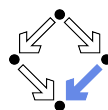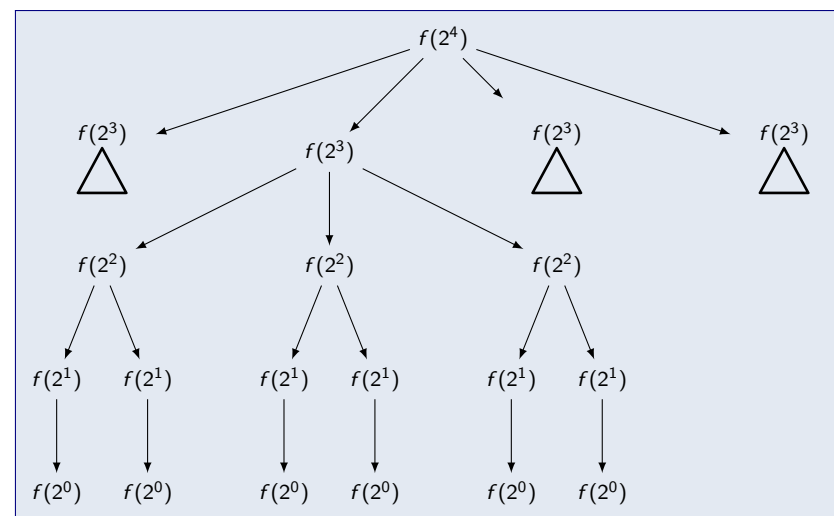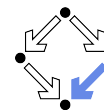

Example

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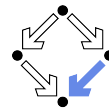
- $f(m)$ calls $f(m/2)$ recursively $\lfloor \log_2 m \rfloor$ times .
  - $f(2^n)$ calls $f(2^{n-1})$ recursively n times.
- How often is f called in total when executing $f(m) = f(2^n)$?
  - Actually, this value is also the result of f.

The analysis of a program involving both loops and recursion.

Recursion Tree
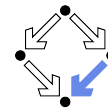
Recursion Tree

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):

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

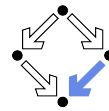What is the number of nodes/function calls $T(n)$ for $f(2^n)$?

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

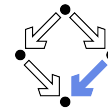Again we add the number of nodes in each level of the tree.

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

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

$$n!/n! = 1$$
$$n!/(n-1)! = n$$
$$n!/(n-2)! = n \cdot (n-1)$$
$$n!/(n-3)! = n \cdot (n-1) \cdot (n-2)$$
$$\cdots$$
$$n!/0! = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1$$

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

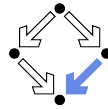$$T(n) = \sum_{i=0}^{n} \frac{n!}{i!}$$

- Induction step: we prove

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

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

$$= 1 + \sum_{i=0}^{n} \frac{(n+1)!}{i!} = \frac{(n+1)!}{(n+1)!} + \sum_{i=0}^{n} \frac{(n+1)!}{i!} = \sum_{i=0}^{n+1} \frac{(n+1)!}{i!} \quad \square$$
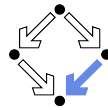
Asymptotic Characterization of the Solution

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 $O(n!) = O((\log_2 m)!)$.

Sums

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

procedure INSERTIONSORT(a)	Cost
$n \leftarrow length(a)$	1
for i **from** 1 **to** $n-1$ **do**	n
$\quad x \leftarrow a[i]$	$n-1$
$\quad j \leftarrow i-1$	$n-1$
$\quad$ **while** $j \geq 0 \wedge a[j] > x$ **do**	$\sum_{i=1}^{n-1} n_i$
$\quad\quad a[j+1] \leftarrow a[j]$	$\sum_{i=1}^{n-1}(n_i - 1)$
$\quad\quad j \leftarrow j-1$	$\sum_{i=1}^{n-1}(n_i - 1)$
$\quad$ **end while**	
$\quad a[j+1] \leftarrow x$	$n-1$
end for	
end procedure	

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

$$T(n) = 1 + n + (n-1) + (n-1) + (n-1) +$$

$$\left(\sum_{i=1}^{n-1} n_i\right) + \left(\sum_{i=1}^{n-1}(n_i - 1)\right) + \left(\sum_{i=1}^{n-1}(n_i - 1)\right) = 4n - 2 + \sum_{i=1}^{n-1}(3n_i - 2)$$

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

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

$$\sum_{i=1}^{n-1}(3i+1) = \sum_{i=0}^{n-1}(3i+1) - 1 = n \cdot 1 + 3 \cdot \frac{(n-1) \cdot n}{2} - 1 = \frac{3n^2 - n - 2}{2}$$
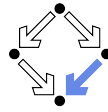
  - Arithmetic series: $\sum_{i=0}^{n}(a + i \cdot d) = (n+1) \cdot a + d \cdot \frac{n \cdot (n+1)}{2}$
  - Geometric series: $\sum_{i=0}^{n}(a \cdot q^i) = a \cdot \frac{q^{n+1} - 1}{q-1}$

    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 $1, \ldots, i+1$ 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: $\overline{T}(n) = 4n - 2 + \frac{1}{2} \cdot \sum_{i=1}^{n-1}(3i+2)$
  $$\overline{T}(n) = 4n - 2 + \sum_{i=1}^{n-1}\left(3 \cdot \frac{i+2}{2} - 2\right) = 4n - 2 + \frac{1}{2} \cdot \sum_{i=1}^{n-1}(3i+2)$$
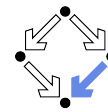
- Closed form: $\overline{T}(n) = \frac{3n^2 + 17n - 12}{4}$
  $$\sum_{i=1}^{n-1}(3i+2) = \sum_{i=0}^{n-1}(3i+2) - 2 = \left(2n + 3 \cdot \frac{(n-1) \cdot n}{2}\right) - 2 = \frac{3n^2 + n - 4}{2}$$
  $$\overline{T}(n) = 4n - 2 + \frac{3n^2 + n - 4}{4} = \frac{16n - 8 + 3n^2 + n - 4}{4} = \frac{3n^2 + 17n - 12}{4}$$

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

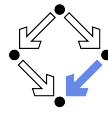# Asymptotic Time Complexity

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

- $\overline{T}(n) \simeq \frac{T(n)}{2}$ (for large n)
  - In the average, the algorithm is twice as fast as in the worst case.
- $\overline{T}(n) = T(n) = \Theta(n^2)$
  - 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:
  $$0, 4, 4+7, 4+7+10, 4+7+10+13, \ldots = 0, 4, 11, 21, 34, \ldots$$
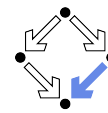- On-line Encyclopedia of Integer Sequences (http://oeis.org)

Solving Sums by Guessing and Verifying

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

```
> sum(3*i+1,i=1..n-1);
                       2
              3/2 n  - 1/2 n - 1

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

                     2
              -2 - n + 3 n
Out[1]= -------------
                2

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

              (-1 + n) (2 + 3 n)
Out[2]= ------------------
                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
$$\sum_{i=1}^{n}(3i+1) = \sum_{i=1}^{n-1}(3i+1) + (3n+1) = \frac{3n^2-n-2}{2} + (3n+1)$$
$$= \frac{3n^2-n-2+6n+2}{2} = \frac{3n^2+5n}{2}$$

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

Recurrences

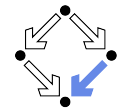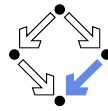Find in sorted array $a[l, r]$ the position of value x (-1, if x does not occur).

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

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 $n = 2^m$.
$$T(2^m) = 5 + T(2^{m-1})$$
$$= \underbrace{5 + \ldots + 5}_{m \text{ times}} + T(1) = \underbrace{5 + \ldots + 5}_{m+1 \text{ times}} + T(0)$$
$$= 5 \cdot (m+1) + 2 = 5m + 7$$

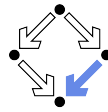- General solution: for all $n \geq 1$.
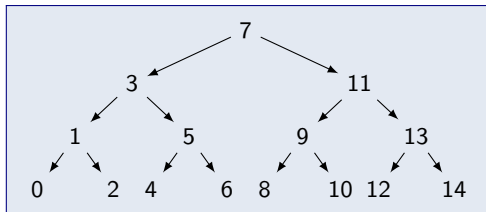$$T(n) = 5 \cdot \lfloor \log_2 n \rfloor + 7$$

  - 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).
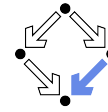
Average Time Complexity

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:
  - $7 \to 3 \to 5$: 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, ..., 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: $0, 2, 10, 34, 98, \ldots$
  ```
  A036799        2+2^(n+1)*(n-1).
            0, 2, 10, 34, 98, ...
  ```
- Computer algebra system:
  ```
  > sum(i*2^i,i=0..m-1);
  ```
  $$m\,2^m - 2\,2^m + 2$$
- Result: $\sum_{i=0}^{m-1} i \cdot 2^i = 2^m \cdot (m-2) + 2$
  - Verification by induction proof.
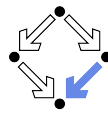- Average number of function calls:
  $$\frac{1}{2^m-1} \cdot \sum_{i=0}^{m-1} i \cdot 2^i = \frac{1}{2^m-1} \cdot (2^m \cdot (m-2)+2) = \frac{2^m \cdot (m-2)}{2^m-1} + \frac{2}{2^m-1} \simeq m-2$$
- Average time complexity:
  $$\overline{T}_{\text{found}}(2^m-1) \simeq 5 \cdot (m-2) + 2 = 5m - 8 = T_{\text{found}}(2^m-1) - 5$$

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

Solving Recurrences by Guessing & Verifying

$$T(1) = 7$$
$$T(n) = 5 + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right), \text{ if } n > 1$$

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

- Simplified recurrence:
  $$U(1) = 0$$
  $$U(n) = 1 + U\left(\left\lfloor \frac{n}{2} \right\rfloor\right), \text{ if } n > 1$$

- Integer sequence: $0, 1, 1, 2, 2, 2, 2, 3, \ldots$
  ```
  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));
                 7 ln(2) + 5 ln(n)
                 -----------------
                       ln(2)

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

                        5 Log[n]
Out[3]= {{T[n] -> 7 + --------}}
                        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

$$T(1) = 7$$
$$T(n) = 5 + T(\lfloor \tfrac{n}{2} \rfloor), \text{ if } n > 1$$
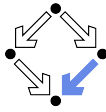
We show for all $n \in \mathbb{N}$ with $n \geq 1$, $T(n) = 5 \cdot \lfloor \log_2 n \rfloor + 7$.

- 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 $T(m) = 5 \cdot \lfloor \log_2 m \rfloor + 7$.
  - Case $n = 2m$:
$$T(n) = 5 + T(\lfloor \tfrac{n}{2} \rfloor) = 5 + T(m) = 5 + (5 \cdot \lfloor \log_2 m \rfloor + 7) = 5 \cdot (1 + \lfloor \log_2 m \rfloor) + 7$$
$$= 5 \cdot \lfloor 1 + \log_2 m \rfloor + 7 = 5 \cdot \lfloor \log_2 2 + \log_2 m \rfloor + 7 = 5 \cdot \lfloor \log_2 2m \rfloor + 7$$
$$= 5 \cdot \lfloor \log_2 n \rfloor + 7$$
  - Case $n = 2m + 1$:
$$T(n) = 5 + T(\lfloor \tfrac{n}{2} \rfloor) = 5 + T(m) = 5 + (5 \cdot \lfloor \log_2 m \rfloor + 7) = 5 \cdot (1 + \lfloor \log_2 m \rfloor) + 7$$
$$= 5 \cdot \lfloor 1 + \log_2 m \rfloor + 7 = 5 \cdot \lfloor \log_2 2 + \log_2 m \rfloor + 7 = 5 \cdot \lfloor \log_2 2m \rfloor + 7$$
$$= 5 \cdot \lfloor \log_2(n-1) \rfloor + 7 = 5 \cdot \lfloor \log_2 n \rfloor + 7$$

Divide and Conquer

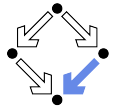We are going to analyze the Mergesort algorithm.

```
procedure MergeSort(a, l, r)     ▷ n = r − l + 1          Cost
    if l < r then                                          1
        m ← ⌊(l+r)/2⌋                                       1
        MergeSort(a, l, m)                                 1 + T(⌈n/2⌉)
        MergeSort(a, m+1, r)                               1 + T(⌊n/2⌋)
        Merge(a, l, m, r)                                  O(n)
    end if
end procedure
```

We will investigate the asymptotic time complexity only.

Recurrence

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

$$T(n) = 1 + 1 + 1 + T(\lceil \tfrac{n}{2} \rceil) + 1 + T(\lfloor \tfrac{n}{2} \rfloor) + O(n)$$
$$= T(\lceil \tfrac{n}{2} \rceil) + T(\lfloor \tfrac{n}{2} \rfloor) + O(n) + 4$$
$$= T(\lceil \tfrac{n}{2} \rceil) + T(\lfloor \tfrac{n}{2} \rfloor) + O(n)$$

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

$$T(n) = 2 \cdot T(\tfrac{n}{2}) + O(n)$$

We are going to guess an asymptotic solution of this recurrence.
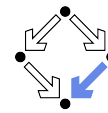
Guessing the Asymptotic Time Complexity

- 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 only verify a special form of the recurrence (see the lecture notes for the general case).

A Special Form of the Recurrence

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

$T(1) = 1, T(2) = 2 \cdot 1 + 2 = 4 = 2 \cdot 2, T(4) = 2 \cdot 4 + 4 = 12 = 4 \cdot 3, T(8) = 2 \cdot 12 + 8 = 42 = 8 \cdot 4, T(16) = 2 \cdot 32 + 16 = 80 = 16 \cdot 5, \ldots$

We prove by induction on i:
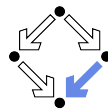
$$\forall i \in \mathbb{N} : T(2^i) = 2^i \cdot (i+1)$$

- Base case: we have $T(2^0) = T(1) = 1 = 2^0 \cdot 1$.
- Induction assumption: we assume $T(2^i) = 2^i \cdot (i+1)$.
- Induction step: we have

$$T(2^{i+1}) = 2 \cdot T(2^i) + 2^{i+1} = 2 \cdot 2^i \cdot (i+1) + 2^{i+1}$$
$$= 2^{i+1} \cdot (i+1) + 2^{i+1} = 2^{i+1} \cdot (i+2).$$

For $n = 2^i$, we thus have $T(n) = n \cdot (\log_2 n + 1) = O(n \cdot \log n)$.

The Master Theorem

Let $a \geq 1$, $b > 1$, $f : \mathbb{N} \to \mathbb{N}$, $T : \mathbb{N} \to \mathbb{N}$ 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)                              Cost
    n ← digits(a)    ▷ digits(a) = digits(b)         1
    c ← 0                                            1
    for i from n−1 to 0 do                           n+1
        p ← MULTIPLYDIGIT(a, b_i)                    n · Θ(n)
        c ← SHIFT(c, 1)                              Σ_{i=0}^{n−1} Θ(2n−i−1)
        c ← ADD(p, c)                                n · Θ(n)
    end for
    return c                                         1
end function
```
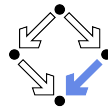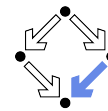
$$T(n) = 3 + (n+1) + 2n \cdot \Theta(n) + \sum_{i=0}^{n-1} \Theta(2n-i-1) = \Theta(n^2)$$

Classical ("school") algorithm has quadratic time complexity.

Arbitrary Precision Multiplication

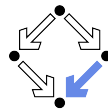Split a and b into halves (a', a'') and (b', b'') of $\frac{n}{2}$ digits.

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

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

Basis of a recursive multiplication algorithm.
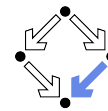

Arbitrary Precision Multiplication (Rec.)

```
function MULTIPLY(a, b)                                        Cost
    n ← digits(a)      ▷ digits(a) = digits(b) = 2^m          1
    if n = 1 then                                             1
        c ← MULTIPLYDIGIT(a_0, b_0)                           Θ(1)
    else
        a' ← a_{n/2...n−1}; a'' ← a_{0... n/2 −1}             Θ(n)
        b' ← b_{n/2...n−1}; b'' ← b_{0... n/2 −1}             Θ(n)
        u ← MULTIPLY(a', b')                                  1 + T(n/2)
        v ← MULTIPLY(a', b'')                                 1 + T(n/2)
        w ← MULTIPLY(a'', b')                                 1 + T(n/2)
        x ← MULTIPLY(a'', b'')                                1 + T(n/2)
        y ← ADD(v, w)                                         Θ(n)
        y ← SHIFT(y, n/2)                                     Θ(n)
        c ← SHIFT(u, n)                                       Θ(n)
        c ← ADD(c, y)                                         Θ(n)
        c ← ADD(c, x)                                         Θ(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:

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

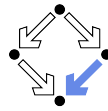- Case 1 of the master theorem ($a = 4, b = 2$):

$$(\log_b a) = (\log_2 4) = 2$$
$$f(n) = O(n) = O(n^1) = O(n^{2-1}) = O(n^{(\log_b a)-1})$$

- 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.

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

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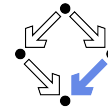- Product $s \cdot t$ where $s = a' + a''$ and $t = b' + b''$ may have $\frac{n}{2} + 1$ digits.

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

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

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

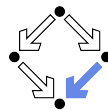We only need three multiplications of numbers with $\frac{n}{2}$ digits.

Karatsuba Algorithm

	Cost
function MULTIPLY(a, b)	
$n \leftarrow digits(a)$ ▷ $digits(a) = digits(b) = 2^m$	1
if $n = 1$ **then**	1
$c \leftarrow$ MULTIPLYDIGIT(a_0, b_0)	$\Theta(1)$
else	
$a' \leftarrow a_{\frac{n}{2}\ldots n-1}; a'' \leftarrow a_{0\ldots\frac{n}{2}-1}$	$\Theta(n)$
$b' \leftarrow b_{\frac{n}{2}\ldots n-1}; b'' \leftarrow b_{0\ldots\frac{n}{2}-1}$	$\Theta(n)$
$s \leftarrow$ ADD(a', a'')	$\Theta(n)$
$t \leftarrow$ ADD(b', b'')	$\Theta(n)$
$u \leftarrow$ MULTIPLY(a', b')	$1 + T(\frac{n}{2})$
$v \leftarrow$ MULTIPLY'(s, t)	$\Theta(n) + T(\frac{n}{2})$
$x \leftarrow$ MULTIPLY(a'', b'')	$1 + T(\frac{n}{2})$
$y \leftarrow$ SUBTRACT(v, u)	$\Theta(n)$
$y \leftarrow$ SUBTRACT(v, x)	$\Theta(n)$
$y \leftarrow$ SHIFT$(y, \frac{n}{2})$	$\Theta(n)$
$c \leftarrow$ SHIFT(u, n)	$\Theta(n)$
$c \leftarrow$ ADD(c, y)	$\Theta(n)$
$c \leftarrow$ ADD(c, x)	$\Theta(n)$
end if	
return c	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:

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

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

$$\log_b a = \log_2 3$$
$$f(n) = O(n) = O(n^1) = O(n^{(\log_b a) - \varepsilon})$$
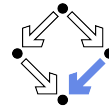$$1.58 < \log_2 3 < 1.59, \varepsilon = (\log_2 3) - 1 > 0.58$$

- 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.

The Quicksort Algorithm

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

	Cost
procedure QUICKSORT(a, l, r) $\quad \triangleright n = r - l + 1$	1
if $l < r$ **then**	
choose $p \in [l, r]$	$O(n)$
$m \leftarrow$ PARTITION(a, l, r, p) $\quad \triangleright i = m - l$	$\Theta(n)$
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 $0 \le i \le n - 1$).

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.

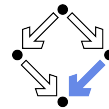- Both intervals have same size ($i = \frac{n}{2}$):
$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + \Theta(n) = 2 \cdot T\left(\frac{n}{2}\right) + \Theta(n)$$
$$= \Theta(n \cdot \log n)$$

  - 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.
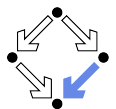

Average Time Complexity of Quicksort

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

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

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

Average Time Complexity of Quicksort
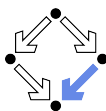
Consider special form of recurrence (see lecture notes for the general case).

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

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

Average Time Complexity of Quicksort

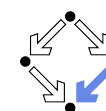We solve the special recurrence.

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

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

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

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

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

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

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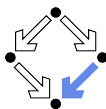
- `Psi(n)`: $\Psi(n) = H_{n-1} - \gamma$,

```
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 ∈ [l, r]`

  determines a pivot element $a[p]$ that is equally likely to be the element at any of the positions $l, \ldots, r$.
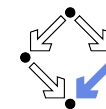- We might choose a fixed index in interval $[l, r]$, e.g.,

  `p ← 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
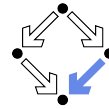
We might shuffle the input array randomly before sorting it.

```
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
```

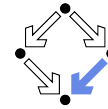Indeed ensures that all permutations are equally likely but is costly.

Randomization

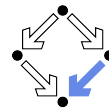We may simply choose the pivot element randomly.

$$p \leftarrow \textsc{Random}(l, r)$$

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.
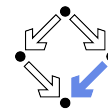
Amortized Analysis

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).
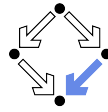
Example

- Consider a stack with the following operations:
  - $\textsc{Push}(s, x)$: push an element x on stack s.
    Time $O(1)$.
  - $\textsc{Pop}(s)$: pop an element from the top of the non-empty stack s.
    Time $O(1)$.
  - $\textsc{MultiPop}(s, k)$: 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 $\textsc{MultiPop}$ 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.

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

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

- The total cost of the sequence is

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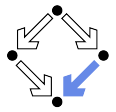
$$\stackrel{(1)}{=} O\left(\sum_{i=0}^{k-1} n_i\right) + O(n) \stackrel{(2)}{=} O(n) + O(n) = O(n)$$

  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 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).

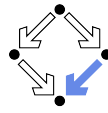$$\hat{c}_i - c_i = C \cdot (\Phi(s_i) - \Phi(s_{i-1}))$$

  - $\hat{c}_i - c_i$: 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:

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

  - 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).

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

  - 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 $O(n)$.
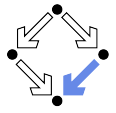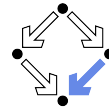

Dynamic Tables

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

- Operation 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.

$$T(n) = 1 + 2 + 3 + \ldots (n-1) = \sum_{i=1}^{n-1} i = O(n^2)$$
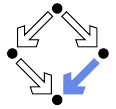
- $m+c$: every c-th call triggers an expansion.

$$T(n) = \sum_{i=1}^{\lceil \frac{n}{c} \rceil} O((i-1) \cdot c) + (n - \lceil \frac{n}{c} \rceil) \cdot O(1)$$

$$= O(\sum_{i=1}^{\lceil \frac{n}{c} \rceil} i) + O(n) = O(n^2) + O(n) = O(n^2)$$

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

$$T(n) = \sum_{i=0}^{\lceil \log_2 n \rceil + 1} O(2^i) + (n - \lceil \log_2 n \rceil - 1) \cdot O(1)$$

$$= O(\sum_{i=0}^{\lceil \log_2 n \rceil + 1} 2^i) + O(n) = O(\sum_{i=0}^{\lfloor \log_2 n \rfloor} 2^i) + O(n) = O(2n) + O(n) = O(n)$$

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

Analysis with the Potential Method

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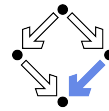 - $num(t)$: number of elements in t.
  - $size(t)$: the number of slots in t.
  - Since $num(t) \geq \frac{size(t)}{2}$, we have $\Phi(t) \geq 0$.
- Change of Potential:
  - After expansion of t_{i-1}: $\Phi(t_i) = 0$.
  - After insertion into t_{i-1}: $\Phi(t_i) = 2 + \Phi(t_{i-1})$.
  - Before expansion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

Cost of expansion can be covered by accumulated credit.

Analysis with the Potential Method

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

  $\hat{c}_i \leq C \cdot 1 + C \cdot \big(2 \cdot num(t_i) - size(t_i) - (2 \cdot num(t_{i-1}) - size(t_{i-1}))\big)$

  $= C \cdot 1 + C \cdot \big(2 \cdot (num(t_{i-1}) + 1) - size(t_{i-1}) - (2 \cdot num(t_{i-1}) - size(t_{i-1}))\big)$

  $= C \cdot 1 + C \cdot 2 = 3C$

- INSERT triggers an expansion:
  - $num(t_i) = num(t_{i-1}) + 1$, $size(t_{i-1}) = num(t_{i-1})$,
  - $size(t_i) = 2 \cdot size(t_{i-1}) = 2 \cdot num(t_{i-1})$

  $\hat{c}_i \leq C \cdot num(t_{i-1}) + C \cdot \big(2 \cdot num(t_i) - size(t_i) - (2 \cdot num(t_{i-1}) - size(t_{i-1}))\big)$

  $= C \cdot num(t_{i-1}) + C \cdot \big(2 \cdot (num(t_{i-1}) + 1) - 2 \cdot num(t_{i-1}) - (2 \cdot num(t_{i-1}) - num(t_{i-1}))\big)$

  $= C \cdot num(t_{i-1}) + C \cdot (2 - num(t_{i-1})) = 2C$

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