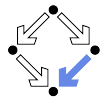


Analysis of Complexity

Wolfgang Schreiner
Wolfgang.Schreiner@risc.jku.at

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



1. Sums

2. Recurrences

3. Divide and Conquer

4. Randomization

5. Amortized Analysis

Sums



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

```

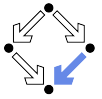
procedure INSERTSORT( $a$ )
   $n \leftarrow \text{length}(a)$ 
  for  $i$  from 1 to  $n-1$  do
     $x \leftarrow a[i]$ 
     $j \leftarrow i-1$ 
    while  $j \geq 0 \wedge a[j] > x$  do
       $a[j+1] \leftarrow a[j]$ 
       $j \leftarrow j-1$ 
    end while
     $a[j+1] \leftarrow x$ 
  end for
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: $\sum_{i=1}^{n-1} (3i + 1) = \frac{3n^2 - n - 2}{2}$

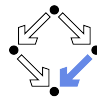
$$\sum_{i=1}^{n-1} (3i + 1) = \sum_{i=0}^{n-1} (3i + 1) - 1 = n \cdot 1 + 3 \cdot \frac{(n-1) \cdot n}{2} - 1 = \frac{3n^2 - n - 2}{2}$$

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

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

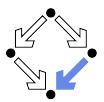
- Closed form: $\bar{T}(n) = \frac{3n^2+17n-12}{4}$

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

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

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

Asymptotic Time Complexity



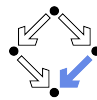
Worst case/average time $T(n) = \frac{3n^2+7n-6}{2}$ and $\bar{T}(n) = \frac{3n^2+17n-12}{4}$.

- $\bar{T}(n) \simeq \frac{T(n)}{2}$ (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.

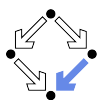
- Determine summation values for growing number of summands:

$$0, 4, 4+7, 4+7+10, 4+7+10+13, \dots = 0, 4, 11, 21, 34, \dots$$
- On-line Encyclopedia of Integer Sequences (<http://oeis.org>)

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

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

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}]

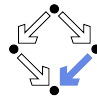
          2
      -2 - n + 3 n
Out[1]= -----
          2

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

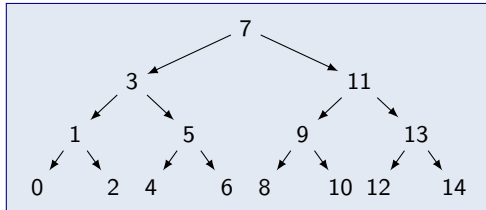
      (-1 + n) (2 + 3 n)
Out[2]= -----
          2
```

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

Average Time Complexity



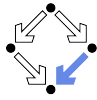
For $n = 2^m - 1$, recursion tree of height $m - 1$ with n nodes:



- 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, ..., 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, ...

$$\begin{aligned} &A036799 && 2+2^{-(n+1)}*(n-1). \\ &&& 0, 2, 10, 34, 98, \dots \end{aligned}$$

- Computer algebra system:

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

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

- 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 and 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 Log₂(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 and 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]
```

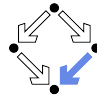
```
Out[3] = {{T[n] -> 7 + -----}}
          5 Log[n]
          -----
          Log[2]
```

- 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 and 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 $n = 2m + 1$:

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

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.

1. Sums

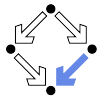
2. Recurrences

3. Divide and Conquer

4. Randomization

5. Amortized Analysis

Recurrence



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

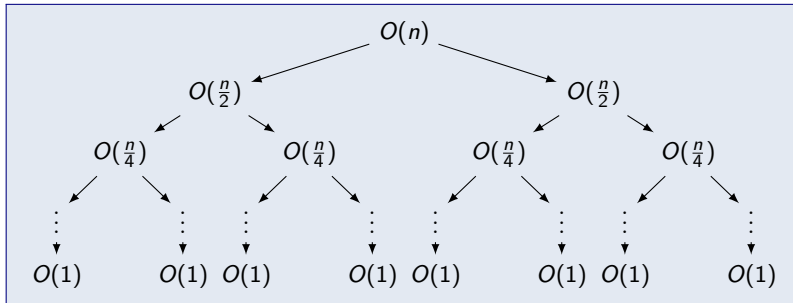
$$\begin{aligned} T(n) &= 1 + 1 + 1 + T(\lceil \frac{n}{2} \rceil) + 1 + T(\lfloor \frac{n}{2} \rfloor) + O(n) \\ &= T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + O(n) + 4 \\ &= T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + O(n) \end{aligned}$$

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

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

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

Guess of Asymptotic Time Complexity

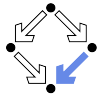


- Execution is the depth-first left-to-right traversal of a binary tree.
- Total time is the sum of the times spent in all tree nodes.

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

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

Verification of 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' .

Verification of Asymptotic Time Complexity (Contd)

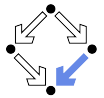


How to choose N' ?

- 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 $T(n) = \dots T\left(\frac{n}{2}\right) \dots$
 - 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 $N' := \max\{2, 3, N\}$.

Verification of Asymptotic Time 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 $c' := \max\{1, T(N')\}$.

Verification of Asymptotic Time 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

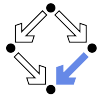
$$T(m) \leq c' \cdot m \cdot \log_2 m$$

- **Induction step:** we show

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

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

Verification of Asymptotic Time Complexity (Contd)



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

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

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

Verification of Asymptotic Time Complexity (Contd)

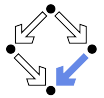


- **Case $\frac{n}{2}$ means $\lceil \frac{n}{2} \rceil$:** since $n > N' \geq N$,

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

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

Verification of Asymptotic Time Complexity (Contd)

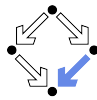


We continue the derivation.

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

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

Verification of Asymptotic Time Complexity (Contd)



We continue the derivation.

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

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.

Example



Analysis of MERGESORT.

■ **Recurrence:**

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

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

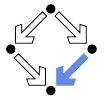
$$\begin{aligned} \log_b a &= 1 \\ \Theta(n) &= \Theta(n^1) = \Theta(n^{\log_b a}) \end{aligned}$$

■ **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.

The Master Theorem



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

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

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

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

■ $f(n) = \Theta(n^{\log_b a})$:

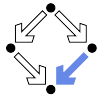
$$T(n) = \Theta(n^{\log_b a} \cdot \log n)$$

■ $f(n) = \Omega(n^{(\log_b a) + \epsilon})$ for some $\epsilon > 0$ 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.

Arbitrary Precision Multiplication



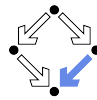
Multiply two natural numbers a and b with n digits each.

function MULTIPLY(a, b)	Cost
$n \leftarrow \text{digits}(a) \triangleright \text{digits}(a) = \text{digits}(b)$	1
$c \leftarrow 0$	1
for i from $n - 1$ to 0 do	$n + 1$
$p \leftarrow \text{MULTIPLYDIGIT}(a, b_i)$	$n \cdot \Theta(n)$
$c \leftarrow \text{SHIFT}(c, 1)$	$\sum_{i=0}^{n-1} \Theta(2n - i - 1)$
$c \leftarrow \text{ADD}(p, c)$	$n \cdot \Theta(n)$
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.

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

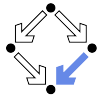
$$b = b' \cdot d^{\frac{n}{2}} + b''$$

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

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

Basis of a recursive multiplication algorithm.

Arbitrary Precision Multiplication (Recursive)



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	1
return c	
end function	

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

Arbitrary Precision Multiplication (Recursive)



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** ($a = 4, b = 2$):

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

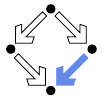
$$f(n) = O(n) = O(n^1) = O(n^{2-1}) = O(n^{(\log_b a)-1})$$

■ **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.

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

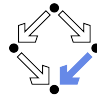
$$s \cdot t = (s_{\frac{n}{2}} \cdot d^{\frac{n}{2}} + s') \cdot (t_{\frac{n}{2}} \cdot d^{\frac{n}{2}} + t')$$

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

- 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



```

function MULTIPLY(a, b)
  n ← digits(a) ▷ digits(a) = digits(b) = 2m
  if n = 1 then
    c ← MULTIPLYDIGIT(a0, b0)
  else
    a' ← a2m-1...n-1}; a'' ← a0...2m-1-1}
    b' ← b2m-1...n-1}; b'' ← b0...2m-1-1}
    s ← ADD(a', a'')
    t ← ADD(b', b'')
    u ← MULTIPLY(a', b')
    v ← MULTIPLY(s, t)
    x ← MULTIPLY(a'', b'')
    y ← SUBTRACT(v, u)
    y ← SUBTRACT(y, x)
    y ← SHIFT(y, 2m)
    c ← SHIFT(u, n)
    c ← ADD(c, y)
    c ← ADD(c, x)
  end if
  return c
end function
    
```

	Cost
1	1
if n = 1 then	1
c ← MULTIPLYDIGIT(a ₀ , b ₀)	Θ(1)
else	Θ(n)
a' ← a _{2^{m-1}...n-1}; a'' ← a_{0...2^{m-1}-1}}}	Θ(n)
b' ← b _{2^{m-1}...n-1}; b'' ← b_{0...2^{m-1}-1}}}	Θ(n)
s ← ADD(a', a'')	Θ(n)
t ← ADD(b', b'')	Θ(n)
u ← MULTIPLY(a', b')	1 + T($\frac{n}{2}$)
v ← MULTIPLY(s, t)	Θ(n) + T($\frac{n}{2}$)
x ← MULTIPLY(a'', b'')	1 + T($\frac{n}{2}$)
y ← SUBTRACT(v, u)	Θ(n)
y ← SUBTRACT(y, x)	Θ(n)
y ← SHIFT(y, 2 ^m)	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

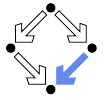
Three recursive calls of the algorithm with half the input size.

Wolfgang Schreiner

<http://www.risc.jku.at>

37/63

Karatsuba Algorithm



Analysis of Karatsuba algorithm by the Master Theorem.

■ **Recurrence:**

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

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

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

$$\log_b a = \log_2 3$$

$$f(n) = O(n) = O(n^1) = O(n^{(\log_2 3) - \epsilon})$$

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

■ **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.

Wolfgang Schreiner

<http://www.risc.jku.at>

38/63

1. Sums

2. Recurrences

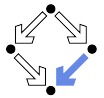
3. Divide and Conquer

4. Randomization

5. Amortized Analysis



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) ▷ n = r - l + 1	1
if l < r then	O(n)
choose p ∈ [l, r]	Θ(n)
m ← PARTITION(a, l, r, p) ▷ i = m - l	1 + T(i)
QUICKSORT(a, l, m - 1)	1 + T(n - i - 1)
QUICKSORT(a, m + 1, r)	
end if	
end procedure	

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

Wolfgang Schreiner

<http://www.risc.jku.at>

39/63

Wolfgang Schreiner

<http://www.risc.jku.at>

40/63

Time Complexity of Quicksort



Recurrence:

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

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

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

- 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 .

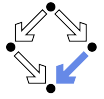
Both intervals have same size ($i = \frac{n}{2}$):

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

- 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

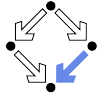


Consider a more special form of the recurrence.

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

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

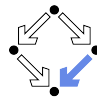
Average Time Complexity of Quicksort



We solve the special recurrence.

$$\begin{aligned} \sum_{i=1}^n \frac{T'(i)}{i+1} &= \sum_{i=1}^n \left(\frac{T'(i-1)}{i} + \frac{2i-1}{i \cdot (i+1)} \right) \\ \sum_{i=1}^n \frac{T'(i)}{i+1} &= \sum_{i=1}^n \frac{T'(i-1)}{i} + \sum_{i=1}^n \frac{2i-1}{i \cdot (i+1)} \\ \sum_{i=1}^n \frac{T'(i)}{i+1} &= \sum_{i=0}^{n-1} \frac{T'(i)}{i+1} + \sum_{i=1}^n \frac{2i-1}{i \cdot (i+1)} \\ \frac{T'(n)}{n+1} &= \frac{T'(0)}{1} + \sum_{i=1}^n \frac{2i-1}{i \cdot (i+1)} \\ T'(n) &= (n+1) \cdot \left(T'(0) + \sum_{i=1}^n \frac{2i-1}{i \cdot (i+1)} \right) \\ T'(n) &= O\left(n \cdot \sum_{i=1}^n \frac{1}{i}\right) = O(n \cdot H_n) = O(n \cdot \log n) \end{aligned}$$

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.

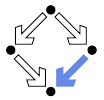
```
> rsolve({T(0)=1,T(n)=(n+1)/n*T(n-1)+1},T(n));
      (n + 1) (Psi(n + 2) + gamma)
```

- 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 $O(n \cdot H_n)$.



Average Time Complexity of Quicksort

We thus guess that the solution of the recurrence

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

is in $O(n \cdot \log n)$. We are now going to verify our guess.

- We know there exist some $N \in \mathbb{N}$ and some $c \in \mathbb{R}_{>0}$ such that for all $n \geq N'$

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

- We have to show, for some $N' \in \mathbb{N}$ and $c' \in \mathbb{R}_{>0}$, that for all $n \geq N'$

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



Average Time Complexity of Quicksort

- To ensure that $N' \geq N$ and $\log_2 N' \geq 1$, we choose

$$N' := \max\{2, N\}.$$

- Since we will apply the induction hypothesis to values $2, \dots, n-1$, we have to show the induction base for all values $n \in \{2, \dots, N'\}$.
- Since $n \geq 2$ and $\log_2 n \geq 1$, to ensure

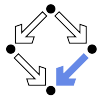
$$T(n) \leq c' \cdot n \cdot \log_2 n$$

it suffices to choose c' such that $c' \geq T(n)$.

- Since $T(2) \leq \dots \leq T(N')$, it thus suffices to choose

$$c' := \max\{_, T(N')\}$$

For $N' := \max\{2, N\}$ and $c' := \max\{_, T(N')\}$ the induction base holds.



Average Time Complexity of Quicksort

- We take arbitrary $n > N'$ and assume that for all i with $2 \leq i < n$

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

- Our goal is to show that

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

- We know

$$\begin{aligned} T(n) &\leq \frac{2}{n} \cdot \sum_{i=0}^{n-1} T(i) + c \cdot n = \frac{2}{n} \cdot \sum_{i=2}^{n-1} T(i) + \frac{2}{n} \cdot (T(0) + T(1)) + c \cdot n \\ &\leq \frac{2}{n} \cdot \sum_{i=2}^{n-1} T(i) + T(0) + T(1) + c \cdot n \leq \frac{2}{n} \cdot \sum_{i=2}^{n-1} T(i) + (T(0) + T(1) + c) \cdot n \\ &= \frac{2}{n} \cdot \sum_{i=2}^{n-1} T(i) + c'' \cdot n \end{aligned}$$

where $c'' := T(0) + T(1) + c$ and the last two inequalities hold because $n > N' \geq 2$.

Average Time Complexity of Quicksort

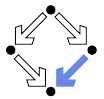


Since $N' \geq 2$, we can apply induction hypothesis to all summands of $\sum_{i=2}^{n-1} T(i)$ and have

$$\begin{aligned} T(n) &\leq \frac{2}{n} \cdot \left(\sum_{i=2}^{n-1} c' \cdot i \cdot \log_2 i \right) + c'' \cdot n = \frac{2 \cdot c'}{n \cdot \ln 2} \cdot \left(\sum_{i=2}^{n-1} i \cdot \ln i \right) + c'' \cdot n \\ &\stackrel{(1)}{\leq} \frac{2 \cdot c'}{n \cdot \ln 2} \cdot \left(\int_2^n i \cdot \ln i \, di \right) + c'' \cdot n \stackrel{(2)}{=} \frac{2 \cdot c'}{n \cdot \ln 2} \cdot \left(\frac{i^2 \cdot \ln i}{2} - \frac{i^2}{4} \right) \Big|_2^n + c'' \cdot n \\ &= \frac{c'}{n \cdot 2 \cdot \ln 2} \cdot \left(2 \cdot i^2 \cdot \ln(i) - i^2 \right) \Big|_2^n + c'' \cdot n \\ &= \frac{c'}{n \cdot 2 \cdot \ln 2} \cdot (2 \cdot n^2 \cdot \ln(n) - n^2 - 8 \cdot \ln 2 + 4) + c'' \cdot n \\ &\leq \frac{c'}{n \cdot 2 \cdot \ln 2} \cdot (2 \cdot n^2 \cdot \ln(n) - n^2) + c'' \cdot n \\ &= \frac{c' \cdot n}{2 \cdot \ln 2} \cdot (2 \cdot \ln(n) - 1) + c'' \cdot n \end{aligned}$$

We use in (1) the fact that the area under the continuous curve $i \cdot \ln i$ in range $2 \leq i \leq n$ overapproximates the corresponding discrete sum in range $2 \leq i \leq n-1$; in (2) we use $\int i \cdot \ln i \, di = \frac{i^2}{4} \cdot (2 \cdot \ln(i) - 1)$.

Average Time Complexity of Quicksort



We now have to ensure that

$$\begin{aligned} \frac{c' \cdot n}{2 \cdot \ln 2} \cdot (2 \cdot \ln(n) - 1) + c'' \cdot n &\leq c' \cdot n \cdot \log_2 n \Leftrightarrow \\ \frac{c'}{2 \cdot \ln 2} \cdot (2 \cdot \ln(n) - 1) + c'' &\leq c' \cdot \frac{\ln(n)}{\ln 2} \Leftrightarrow \\ c' \cdot (2 \cdot \ln(n) - 1) + c'' \cdot 2 \cdot \ln 2 &\leq c' \cdot 2 \cdot \ln(n) \Leftrightarrow \\ c'' \cdot 2 \cdot \ln 2 &\leq c' \cdot (2 \cdot \ln(n) - 2 \cdot \ln(n) + 1) \Leftrightarrow \\ &c'' \cdot 2 \cdot \ln 2 \leq c' \end{aligned}$$

We thus take $c' := \max\{c'' \cdot 2 \cdot \ln 2, T(N')\}$ and are done.

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 .

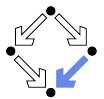
- We might choose a fixed index in interval $[l, r]$, e.g.,

$p \leftarrow r$

- 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



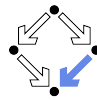
We might shuffle the input array randomly before sorting it.

```

procedure RANDOMIZE( $a$ )
   $n \leftarrow \text{length}(a)$ 
  for  $i$  from 0 to  $n - 1$  do
     $r \leftarrow \text{RANDOM}(i, n - 1)$ 
     $b \leftarrow a[i]; a[i] \leftarrow a[r]; a[r] \leftarrow b$ 
  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.

Amortized Analysis



We want to determine the worst-case time complexity $T(n)$ of a sequence of n operations.

- Average time complexity of every **single operation**:

$$\frac{T(n)}{n}$$

- “**Amortized cost**”: some operations with high costs may be outweighed by many operations with low costs.
 - The average cost of all operations may be still low.
- We consider the **worst case** for the whole sequence.
 - The average performance of each operation in the worst case is computed.

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

1. Sums

2. Recurrences

3. Divide and Conquer

4. Randomization

5. Amortized Analysis

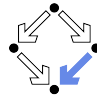
Example



- 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.

$$n = k + \sum_{i=0}^{k-1} n_i$$

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

- The total cost of the sequence is

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

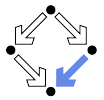
$$\stackrel{(1)}{=} O\left(\sum_{i=0}^{k-1} n_i\right) + O(n) \stackrel{(2)}{=} O(n) + O(n) = O(n)$$

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

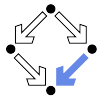
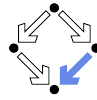


Table t for which a certain amount of space is allocated.

- 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.

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

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

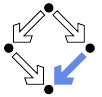
$$\begin{aligned} T(n) &= \sum_{i=1}^{\lceil \frac{n}{c} \rceil} O((i-1) \cdot c) + (n - \lceil \frac{n}{c} \rceil) \cdot O(1) \\ &= O\left(\sum_{i=1}^{\lceil \frac{n}{c} \rceil} i\right) + O(n) = O(n^2) + O(n) = O(n^2) \end{aligned}$$

- $m \cdot c$: 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 $O(1)$.