Fundamental Algorithms

Chapter 1: Introduction

Christian Scheideler

WS 2017

Basic Information

- Lectures: Fr 11:15-12:45 and 13:15-14:00, F0.530
- Tutorials: Mo 9-11 am (F1.110) and We 2-4 pm (F0.530)
- Assistant: Kristian Hinnenthal
- Tutorials start in third week
- Course webpage: http://cs.uni-paderborn.de/ti/lehre/veranstaltungen/ws-20172018/fundamental-algorithms/
- Written exam at the end of the course
- Prerequisite: Data Structures and Algorithms (DuA)

Basic Information

Homework assignments and bonus points:

- New homework assignment: every Friday on the course webpage, starting with today.
- Submission of homework: one week later, by 11:15 am. Homework can be submitted by a team of 1-3 people.
- Discussion of homework: one week later, in the tutorials.
- Presentation of a solution in a tutorial: bonus of 0.3 points for the presenting person. These points can only be used if the exam is passed.

Contents

- Advanced Heaps
  - Binomial Heaps
  - Fibonacci Heaps
  - Radix Heaps
  - Applications
- Advanced Search Structures
  - Splay Trees
  - (a,b)-Trees
- Graph Algorithms
  - Connected Components
  - Shortest Paths
  - Matchings
- Network Flows
  - Ford-Fulkerson Algorithm
  - Preflow-Push Algorithm
  - Applications
- String Matching Algorithms
  - Knuth-Morris-Pratt Algorithm
  - Boyer-Moore Algorithm
  - Aho-Corasick Algorithm

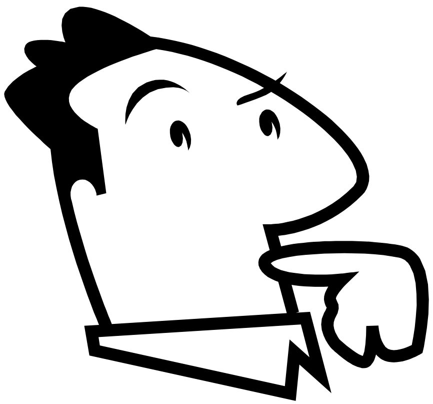
Literature

- T.H. Corman, C.E.Leiserson, R.L. Rivest. Introduction to Algorithms. MIT Press, 1990.
- R.K. Ahuja, T.L. Magnanti, J.B. Orlin. Network flows. Prentice Hall, 1993.
- G.A. Stephen. String searching algorithms. World Scientific Publishing, 1994.
- S.O. Krumke, H. Noltemeier. Graphentheoretische Konzepte und Algorithmen. Teubner, 2005.
- A.V. Aho, J.E. Hopcroft, J.D. Ullman. Data Structures and Algorithms. Addison-Wesley , 1987.
- M.R. Garey, D.S. Johnson. Computers and intractability. Freeman, 2000 (22nd reprint).
- U. Manber. Introduction to algorithms. Addison-Wesley, 1989.
- S.S. Skiena. The algorithm design manual. Springer, 1998.
- T. Ottmann (Editor). Prinzipien des Algorithmenentwurfs. Spektrum Akad. Verlag , 1998.
- U.Schöning. Algorithmik. Spektrum Akad. Verlag , 2001.
- V. Turau. Algorithmische Graphentheorie. Addison-Wesley, 1996.
- D. Jungnickel. Graphs, networks and algorithms. Springer, 1999.

Introduction

Topic: Fundamental Algorithms

Theory?

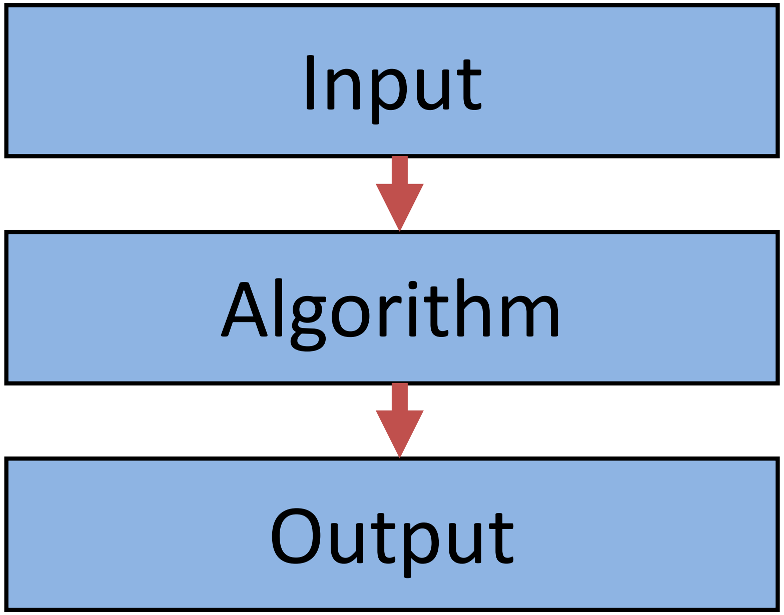


Do I have to write programs?

- What is an algorithm?
- What is a data structure?

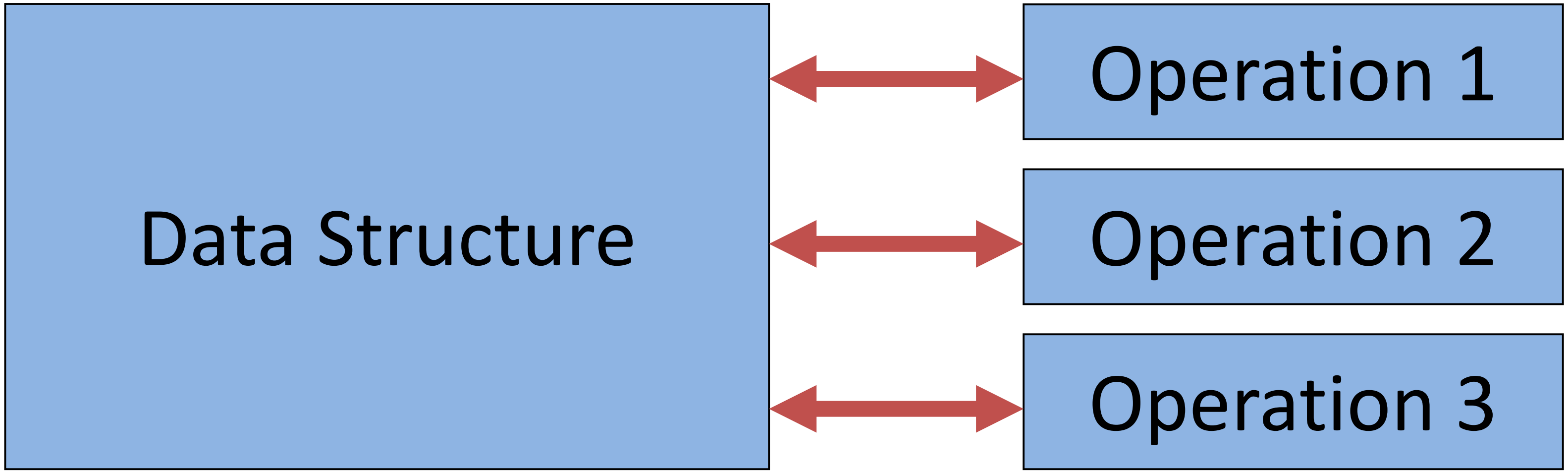
What is an Algorithm?

Definition: An algorithm is a formal step-by-step procedure in order to solve instances of a given problem in finite time.

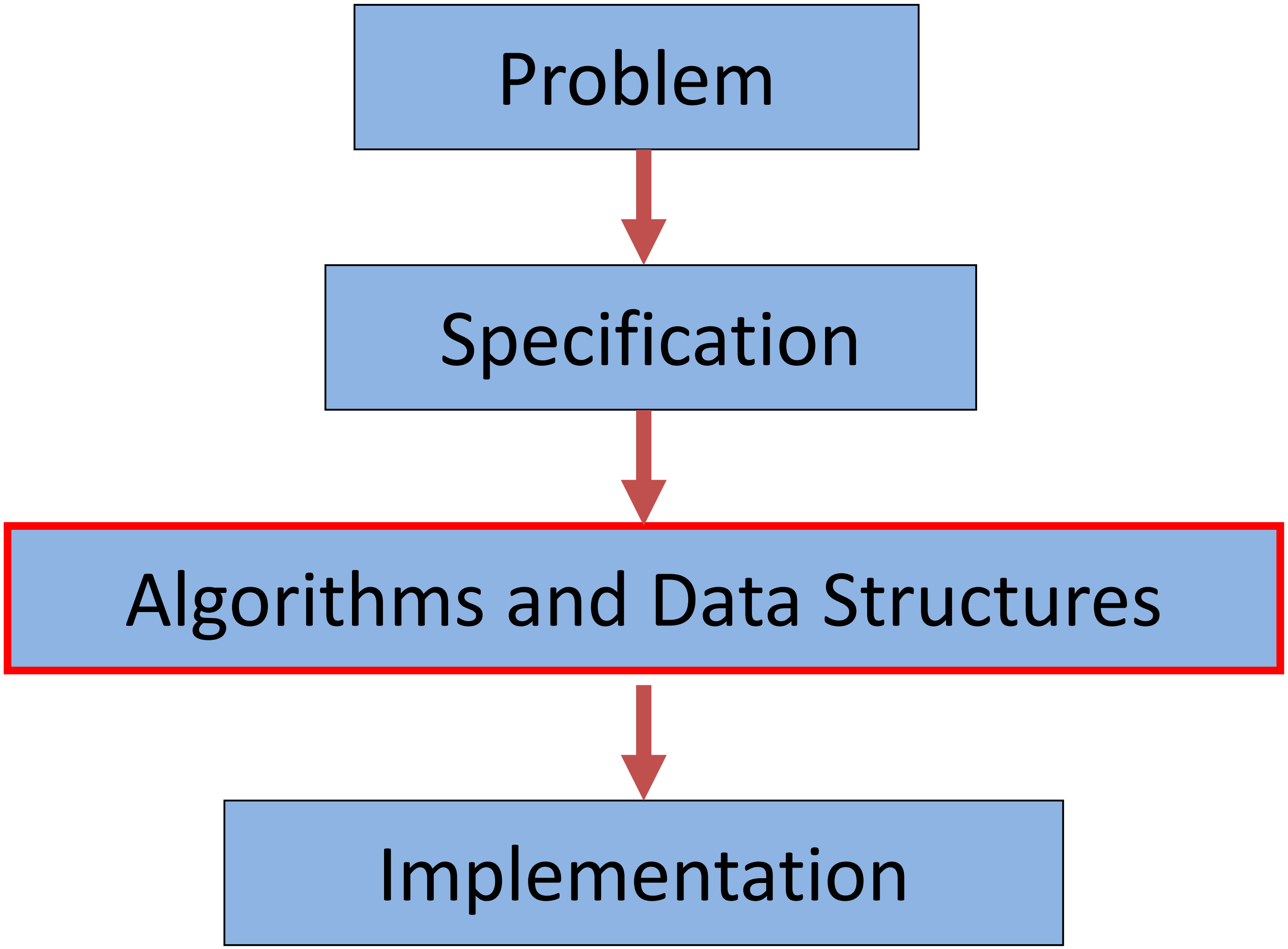


What is a Data Structure?

Definition: A data structure is a particular way of storing and organizing data that allows them to be accessed and maintained.



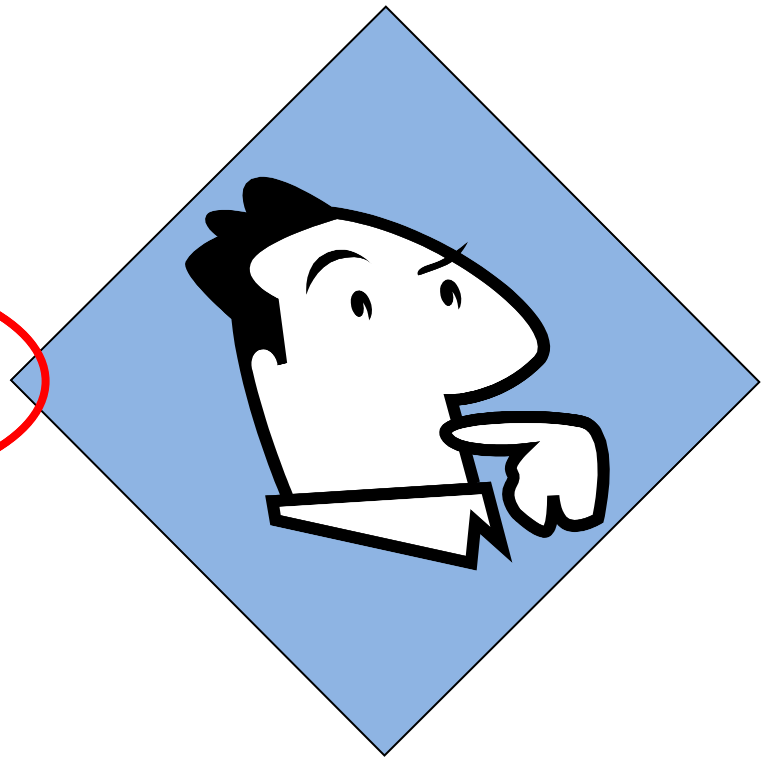
Software Development



Fundamental Problems

Correctness

Efficiency



Complexity

Robustness / Security

Efficiency

Important: Runtime and resource consumption

Why?

- Large amounts of data (bio informatics)
- Real-time applications (games)

Goal of the lecture:

Advanced set of efficient algorithms and data structures for important problems

Efficiency

Measurement of Efficiency:

- Algorithm:
  based on input size
  (e.g., memory needed for a given input)
- Data structure:
  based on size of the data structure
  (e.g., number of elements in data structure)
  resp. the length of the request sequence
  applied to an initially empty data structure

Efficiency

Input size:

- Size of numbers: length of binary encoding
- Size of a set or sequence of numbers: often just the number of elements

Example: Sorting

Input: sequence of numbers $a_1,\dots,a_n \in \mathbb{N}$

Output: sorted sequence of these numbers

Size of input: n

Measurement of Efficiency

Basic Notation:

- $\mathbb{N}=\{1,2,3,\ldots\}$: set of natural numbers
- $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$: set of non-negative integers
- $\mathbb{Z}$: set of integers
- $\mathbb{R}$: set of real numbers
- For all $n \in \mathbb{N}_0$: $[n]=\{1,\ldots,n\}$ and $[n]_0 = \{0,\ldots,n\}$
- Given a set L let $\wp^k(L)$ the set of subsets of L of cardinality k

Measurement of Efficiency

- I: set of instances
- $T: I \rightarrow \mathbb{N}$: runtime $T(i)$ of algorithm for instance $i \in I$
- I_n: set of all instances of size n.

Common measures:

- Worst case: $t(n)=\max\{T(i): i \in I_n\}$
- Best case: $t(n)=\min\{T(i): i \in I_n\}$
- Average case: $t(n) = (1/|I_n|) \sum_{i \in I_n} T(i)$

We will mainly look at the worst case.

Measurement of Efficiency

Why worst case?

- “typical case” hard to grasp, average case is not necessarily a good measure for that
- gives guarantees for the efficiency of an algorithm (important for robustness)

Exact formula for t(n) very hard to compute!

Much easier: asymptotic growth

Asymptotic Notation

Informally: Two functions $f(n)$ and $g(n)$ have the same asymptotic growth if there are constants $c>0$ and $d>0$ so that $c<f(n)/g(n)<d$ resp. $c<g(n)/f(n)<d$ for all sufficiently large n.

Example: n^2, $5n^2-7n$ and $n^2/10+100n$ have the same asymptotic growth since, e.g.,

$$1/5<(5n^2-7n)/n^2<5 \text{ resp. } 1/5<n^2/(5n^2-7n)<5$$

for all $n\geq 2$.

Asymptotic Notation

Why is it sufficient to consider sufficiently large n?

Goal: algorithms that are efficient even for very large instances (i.e., they scale well).

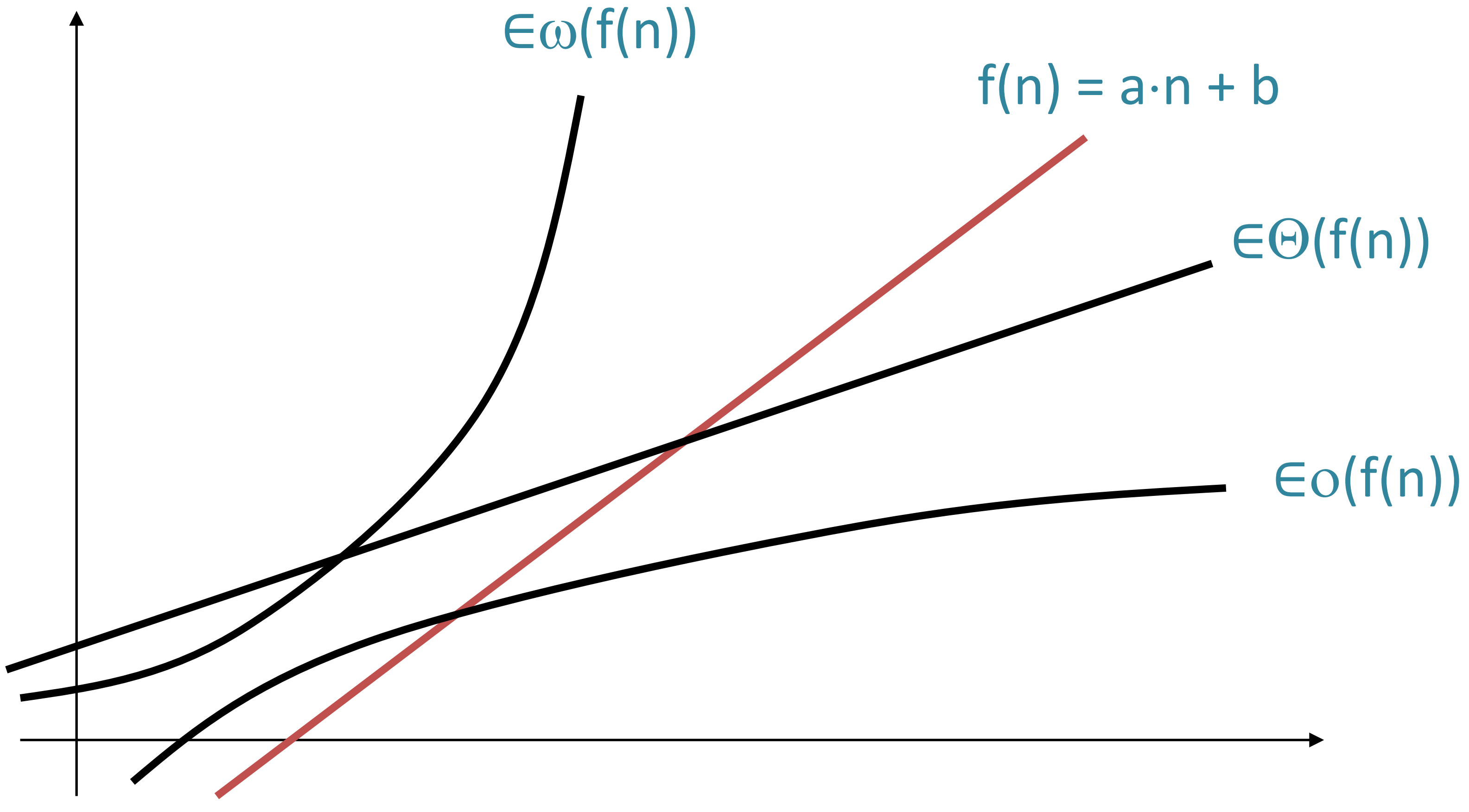
The following sets formalize asymptotic growth:

- $O(f(n))=\{ g(n) \mid \exists c>0\ \exists n_0>0\ \forall n > n_0: g(n) \leq c \cdot f(n) \}$
- $\Omega(f(n))=\{ g(n) \mid \exists c>0\ \exists n_0>0\ \forall n > n_0: g(n) \geq c \cdot f(n) \}$
- $\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$
- $o(f(n))=\{ g(n) \mid \forall c>0\ \exists n_0>0\ \forall n>n_0: g(n) \leq c \cdot f(n) \}$
- $\omega(f(n))=\{ g(n) \mid \forall c>0\ \exists n_0>0\ \forall n>n_0: g(n) \geq c \cdot f(n) \}$

Only functions f(n) (resp. g(n)) with $\exists N>0\ \forall n>N: f(n) > 0$!

They are supposed to be measures for time and space.

Asymptotic Notation



Asymptotic Notation

- $\limsup_{n\to\infty} x_n$: $\lim_{n\to\infty} (\sup_{m\geq n} x_m)$

  sup: supremum (example: $\sup\{ x\in\mathbb{R} \mid x<2 \} = 2$)
- $\liminf_{n\to\infty} x_n$: $\lim_{n\to\infty} (\inf_{m\geq n} x_m)$

  inf: infimum (example: $\inf \{ x\in\mathbb{R} \mid x>3 \} = 3$)

Alternative way of defining O-notation:

- $O(g(n))=\{ f(n) \mid \exists\, c>0 \ \limsup_{n\to\infty} f(n)/g(n) \leq c \}$
- $\Omega(g(n))=\{ f(n) \mid \exists\, c>0 \ \liminf_{n\to\infty} f(n)/g(n) \geq c \}$
- $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$
- $o(g(n))=\{ f(n) \mid \limsup_{n\to\infty} f(n)/g(n) = 0 \}$
- $\omega(g(n))=\{ f(n) \mid \liminf_{n\to\infty} g(n)/f(n) = 0 \}$

Crash Course on Limits

Let $f:\mathbb{N}\rightarrow\mathbb{R}$ be a function and $a,b\in\mathbb{R}$.

- f has in ∞ limit b if for every $\varepsilon>0$ there is a $k>0$ with $|f(z)-b|<\varepsilon$ for all $z\in\mathbb{R}$ with $z>k$. In this case we write

$$\lim_{z\rightarrow\infty} f(z) = b$$

- f has in ∞ limit ∞ if for every $c>0$ there is a $k>0$ with $f(z)>c$ for all $z\in\mathbb{R}$ with $z>k$. In this case we write

$$\lim_{z\rightarrow\infty} f(z) = \infty$$

For every sequence $(x_n)_{n\in\mathbb{N}}$ of real numbers, $\liminf\limits_{n\rightarrow\infty}$ and $\limsup\limits_{n\rightarrow\infty}$ are well-defined and in $\mathbb{R}\cup\{-\infty, \infty\}$, which is not the case for $\lim\limits_{n\rightarrow\infty}$ since there are sequences that do not converge to some limit according to the definition above. The existence of the infimum and supremum is the result of a theorem by Bolzano and Weierstraß.

Asymptotic Notation

Examples:

- n^2, $5n^2$-$7n$, $n^2/10 + 100n \in O(n^2)$
- $n \log n \in \Omega(n)$, $n^3 \in \Omega(n^2)$
- $\log n \in o(n)$, $n^3 \in o(2^n)$
- $n^5 \in \omega(n^3)$, $2^{2n} \in \omega(2^n)$

Asymptotic Notation

O-Notation is also used as a place holder for a function:

- Instead of $g(n) \in O(f(n))$ we may also write $g(n) = O(f(n))$
- Instead of $f(n)+g(n)$ with $g(n) \in o(h(n))$ we may also write $f(n)+g(n) = f(n)+o(h(n))$
- Instead of $O(f(n)) \subseteq O(g(n))$ we may also write $O(f(n)) = O(g(n))$

Example: $n^3 + n = n^3 + o(n^3) = (1+o(1))n^3 = O(n^3)$

Equations with O-Notation should only be read from left to right!

Asymptotic Notation

O- and Ω- resp. o- and ω-notation are complementary to each other, i.e.:

- $f(n) = O(g(n)) \;\Rightarrow\; g(n) = \Omega(f(n))$
- $f(n) = \Omega(g(n)) \;\Rightarrow\; g(n) = O(f(n))$
- $f(n) = o(g(n)) \;\Rightarrow\; g(n) = \omega(f(n))$
- $f(n) = \omega(g(n)) \;\Rightarrow\; g(n) = o(f(n))$

Proof: follows from definition of notation

Asymptotic Notation

Proof of $f(n) = O(g(n)) \Rightarrow g(n) = \Omega(f(n))$:

- $f(n) = O(g(n))$: there are $c, n_0 > 0$ so that

  $$f(n) \le c \cdot g(n) \text{ for all } n > n_0.$$

- Hence, there are $c'(=1/c), n_0 > 0$ so that

  $$g(n) \ge c' \cdot f(n) \text{ for all } n > n_0.$$

- Therefore, $g(n) = \Omega(f(n))$.

Asymptotic Notation

O-, Ω- and Θ-notation is reflexive, i.e.:

- $f(n) = O(f(n))$
- $f(n) = \Omega(f(n))$
- $f(n) = \Theta(f(n))$

Θ-notation is symmetric, i.e.

- $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$.

Proof: via definition of notation

Asymptotic Notation

O-, Ω- und Θ-notation is transitive, i.e.:

$f(n) = O(g(n))$ and $g(n) = O(h(n))$ implies $f(n) = O(h(n))$.

$f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n))$ implies $f(n) = \Omega(h(n))$.

$f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$ implies $f(n) = \Theta(h(n))$.

Proof: via definition of notation.

Transitivity also holds for o- and ω-notation.

Asymptotic Notation

Proof for transitivity of O-notation:

- $f(n) = O(g(n)) \Leftrightarrow$ there are $c', n'_0 > 0$ so that

  $f(n) \leq c' g(n)$ for all $n > n'_0$.
- $g(n) = O(h(n)) \Leftrightarrow$ there are $c'', n''_0 > 0$ so that

  $g(n) \leq c'' h(n)$ for all $n > n''_0$.

Let $n_0 = \max\{n'_0, n''_0\}$ and $c = c' \cdot c''$. Then for all $n > n_0$:

$$f(n) \leq c' \cdot g(n) \leq c' \cdot c'' \cdot h(n) = c \cdot h(n).$$

Asymptotic Notation

Theorem 1.1:
Let $f_1(n) = O(g_1(n))$ and $f_2(n)=O(g_2(n))$.
Then it holds:
(a) $f_1(n) + f_2(n) = O(g_1(n) + g_2(n))$
(b) $f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$
Expressions also correct for Ω , o, ω and Θ.

Theorem 1.2:
(a) $c \cdot f(n) = \Theta(f(n))$ for any constant $c>0$
(b) $O(f(n))+O(g(n)) = O(f(n)+g(n))$
(c) $O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$
(d) $O(f(n)+g(n)) = O(f(n))$ if $g(n)=O(f(n))$
Expressions with O are also correct for Ω , o, ω and Θ.

Be careful with inductive use of (d)!!!

Asymptotic Notation

Proof of Theorem 1.1 (a):

- Let $f_1(n) = O(g_1(n))$ and $f_2(n)=O(g_2(n))$.
- Then it holds:

  $\exists\, c_1>0\ \exists\, n_1>0\ \forall\, n\geq n_1: f_1(n) \leq c_1\cdot g_1(n)$

  $\exists\, c_2>0\ \exists\, n_2>0\ \forall\, n\geq n_2: f_2(n) \leq c_2\cdot g_2(n)$

- Hence, with $c_0=\max\{c_1,c_2\}$ and $n_0=\max\{n_1,n_2\}$ we get:

  $\exists\, c_0>0\ \exists\, n_0>0\ \forall\, n\geq n_0:$

  $$f_1(n)+f_2(n) \leq c_0\cdot(g_1(n)+g_2(n))$$

- Therefore, $f_1(n)+f_2(n) = O(g_1(n)+g_2(n))$.

Proof of (b): exercise

Asymptotic Notation

Proof of Theorem 1.2 (b):

- Consider arbitrary functions $h_1(n)$ and $h_2(n)$ with $h_1(n)=O(f(n))$ and $h_2(n)=O(g(n))$.
- From Theorem 1.1 (a) we know that

$$h_1(n)+h_2(n) = O(f(n)+g(n))$$

- Hence,

$$O(f(n))+O(g(n)) = O(f(n)+g(n))$$

Proof of (c) and (d): exercise

Asymptotic Notation

Theorem 1.3: Let $p(n) = \sum_{i=0}^{k} a_i \cdot n^i$ with $a_k>0$. Then we have $p(n) = \Theta(n^k)$.

Proof:

To show: $p(n) = O(n^k)$ and $p(n) = \Omega(n^k)$.

- $p(n) = O(n^k)$: For all $n \geq 1$,

  $p(n) \leq \sum_{i=0}^{k} |a_i|\, n^i \leq n^k \sum_{i=0}^{k} |a_i|$
- Hence, definition of $O()$ is satisfied with $c=\sum_{i=0}^{k} |a_i|$ and $n_0=1$.

- $p(n) = \Omega(n^k)$: For all $n \geq 2k \cdot A/a_k$ and $A=\max_i |a_i|$,
  $p(n) \geq a_k \cdot n^k - \sum_{i=0}^{k-1} A \cdot n^i \geq a_k\, n^k - k \cdot A\, n^{k-1} \geq a_k\, n^k/2$
- Hence, definition of $\Omega()$ is satisfied with $c=a_k/2$ and $n_0=2kA/a_k$.

Pseudo Code

We will use pseudo code in order to formally specify an algorithm.

Declaration of variables:

v: T : Variable v of type T

v=x: T : is initialized with the value x

Types of variables:

- integer, boolean, char
- Pointer to T: pointer to an element of type T
- Array[i..j] of T: array of elements with index i to j of type T

Pseudo-Code

Allocation and de-allocation of space:

- v:= allocate Array[1..n] of T
- dispose v

Important commands: (C: condition, I,J: commands)

- v:=A   // v receives the result of expression A
- if C then I else J
- repeat I until C,   while C do I
- for v:=a to e do I
- foreach e∈S do I
- return v

Runtime Analysis

What do we know?

- O-notation ($O(f(n))$, $\Omega(f(n))$, $\Theta(f(n)$, …)
- Pseudo code
  (if then else, while do, allocate/dispose,…)

How do we use this to analyze algorithms?

Runtime Analysis

Worst-case runtime:

- T(I): worst-case runtime of instruction I
- T(elementary command) = O(1)
- T(return x) = O(1)
- T(I;I´) = T(I) + T(I´)
- T(if C then I else I´) = T(C) + max{T(I),T(I´)}
- T(for i:=a to b do I) = $\sum_{i=a}^{b}$ T(I)
- T(repeat I until C) = $\sum_{i=1}^{k}$ (T(C)+T(I))
  (k: number of iterations)
- T(while C do I) = $\sum_{i=1}^{k}$ (T(C)+T(I))

Runtime analysis difficult for while- und repeat-loops since we need to determine k, which is sometimes not so easy!

Example: Computation of Sign

Input: number $x \in \mathbb{R}$

Algorithm Signum(x):

if $x<0$ then return -1	$O(1)$
if $x>0$ then return 1	$O(1)$
return 0	$O(1)$

Th 1.2: total runtime: $O(1+1+1)=O(1)$

Example: Minimum

Input: array of numbers A[1],…,A[n]

Minimum Algorithm:

min := 1	$O(1)$
for i:=1 to n do	$O(\sum_{i=1}^{n} T(I))$
if A[i]<min then min:=A[i]	$O(1)$
return min	$O(1)$

runtime: $O(1 + (\sum_{i=1}^{n} 1) + 1) = O(n)$

Example: Sorting

Input: array of numbers A[1],…,A[n]

Bubblesort Algorithm:

```
for i:=1 to n-1 do
   for j:= n-1 downto i do
      if A[j]>A[j+1] then
         x:=A[j]
         A[j]:=A[j+1]
         A[j+1]:=x
```

$O(\sum_{i=1}^{n-1} T(I))$

$O(\sum_{j=i}^{n-1} T(I))$

$O(1 + T(I))$

O(1)

O(1)

O(1)

runtime: $O(\sum_{i=1}^{n-1} \sum_{j=i}^{n-1} 1)$

Example: Sorting

Input: array of numbers A[1],…,A[n]

Bubblesort Algorithm:

```
for i:=1 to n-1 do
   for j:= n-1 downto i do
      if A[j]>A[j+1] then
         x:=A[j]
         A[j]:=A[j+1]
         A[j+1]:=x
```

$$\sum_{i=1}^{n-1} \sum_{j=i}^{n-1} 1$$
$$= \sum_{i=1}^{n-1} (n-i)$$
$$= \sum_{i=1}^{n-1} i$$
$$= n(n-1)/2$$
$$= O(n^2)$$

Example: Binary Search

Input: number x and sorted array A[1],…,A[n]

Binary Search Algorithm:

```
l:=1; r:=n
while l < r do
    m:=(r+l) div 2
    if A[m] = x then return m
    if A[m] < x then l:=m+1
                 else r:=m-1
return l
```

$O(1)$

$O(\sum_{i=1}^{k} T(I))$

$O(1)$

$O(1)$

$O(1)$

$O(1)$

$O(1)$

$O(\sum_{i=1}^{k} 1) = O(k)$

Example: Binary Search

Input: number x and sorted array $A[1],\ldots,A[n]$

Binary Search Algorithm:

```
l:=1; r:=n
while l < r do
    m:=(r+l) div 2
    if A[m] = x then return m
    if A[m] < x then l:=m+1
                else r:=m-1
return l
```

$O(\sum_{i=1}^{k} 1) = O(k)$

What is k ?? →witness!

$\Phi(i):=(r-l+1)$ in iteration i

$\Phi(1)=n$, $\Phi(i+1)\leq\Phi(i)/2$

$\Phi(i)\leq 1$: done

Thus, $k \leq \log n + 1$

Runtime via Potential Function

Find a potential function Φ and $\delta,\Delta>0$ so that

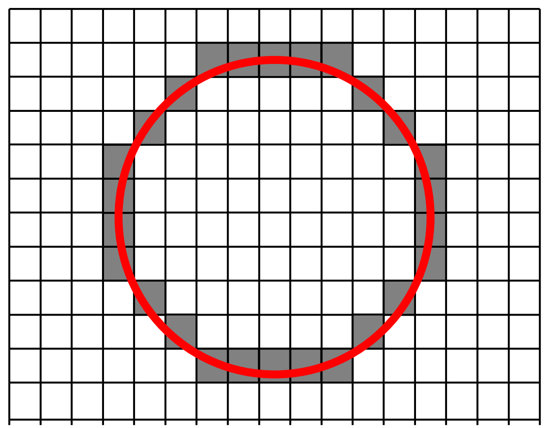
- Φ decreases (resp. increases) by at least δ in each iteration of the while-/repeat-loop and
- Φ is bounded from below (resp. above) by Δ.

Then the while-/repeat-loop is executed at most $1+|\Phi_0-\Delta|/\delta$ times, where Φ_0 is the initial value of Φ.

Better bounds on the number of iterations are possible if, e.g., δ depends on Φ like in Binary Search.

Example: Bresenham Algorithm

```
(x,y):=(0,R)
F:=1-R
plot(0,R); plot(R,0); plot(0,-R); plot(-R,0)
while x<y do
    x:=x+1
    if F<0 then
        F:=F+2·x - 1
    else
        F:=F+2·(x-y)
        y:=y-1
    plot(x,y); plot(y,x); plot(-x,y); plot(y,-x)
    plot(x,-y); plot(-y,x); plot(-y,x); plot(-x,-y)
```



O(1)

O(1)

O(1)

$O(\sum_{i=1}^{k} T(I))$

everything

O(1)

$O(\sum_{i=1}^{k} 1) = O(k)$

Example: Bresenham Algorithm

```
(x,y):=(0,R)
F:=1-R
plot(0,R); plot(R,0); plot(0,-R); plot(-R,0)
while x<y do
    x:=x+1
    if F<0 then
        F:=F+2·x - 1
    else
        F:=F+2·(x-y)
        y:=y-1
    plot(x,y); plot(y,x); plot(-x,y); plot(y,-x)
    plot(x,-y); plot(-y,x); plot(-y,x); plot(-x,-y)
```

Potential function:

$\phi(x,y) = y-x$

monotonic: reduces by ≥ 1 per round of while-loop

bounded: while condition

Example: Bresenham Algorithm

```
(x,y):=(0,R)
F:=1-R
plot(0,R); plot(R,0); plot(0,-R); plot(-R,0)
while x<y do
    x:=x+1
    if F<0 then
        F:=F+2·x - 1
    else
        F:=F+2·(x-y)
        y:=y-1
    plot(x,y); plot(y,x); plot(-x,y); plot(y,-x)
    plot(x,-y); plot(-y,x); plot(-y,x); plot(-x,-y)
```

Potential function:

$\phi(x,y) = y-x$

Number of rounds:

$\phi_0(x,y) = R$, $\phi(x,y)>0$

$\rightarrow$ at most R rounds

Example: Factorial

Input: natural number n

Algorithm Factorial(n):

if $n=1$ then return 1 $\quad O(1)$

else return $n \cdot$ Factorial($n-1$) $\quad O(1 + ??)$

Runtime:

- $T(n)$: runtime of Factorial(n)
- $T(n) = T(n-1) + O(1), \; T(1) = O(1)$

Master-Theorem

Theorem 1.4: For some positive constants a,b,c and d with $n=b^k$ for some natural number k let

$$t(n) = a \quad \text{if } n \le 1$$

$$t(n) = c \cdot n + d \cdot t(n/b) \quad \text{if } n > 1$$

Then it holds that

$$t(n) = \Theta(n) \quad \text{if } d<b$$

$$t(n) = \Theta(n \log n) \quad \text{if } d=b$$

$$t(n) = \Theta(n^{\log_b d}) \quad \text{if } d>b$$

Amortized Analysis

- S: state space of data structure
- F: sequence of operations $Op_1, Op_2, Op_3, \ldots, Op_n$
- s_0: initial state of data structure

$$s_0 \xrightarrow{Op_1} s_1 \xrightarrow{Op_2} s_2 \xrightarrow{Op_3} \ldots \xrightarrow{Op_n} s_n$$

- Total runtime $T(F) = \sum_{i=1}^{n} T_{Op_i}(s_{i-1})$

Amortized Analysis

- Total runtime $T(F) = \sum_{i=1}^{n} T_{Op_i}(s_{i-1})$
- A family of functions $A_X(s)$, one per operation X, is called a family of amortized time bounds if for every sequence F of operations,

  $$T(F) \leq A(F) := c + \sum_{i=1}^{n} A_{Op_i}(s_{i-1})$$

  for some constant c independent of F
- I.e., the amortized runtime of an operation Op denotes the average runtime of Op in the worst case.

Amortized Analysis

- Trivial choice of $A_X(s)$:
  $A_X(s) := T_X(s)$

- Alternative choice of $A_X(s)$:
  via potential function $\phi : S \to \mathbb{R}_{\geq 0}$
  $\to$simplifies proofs

Amortized Analysis

Theorem 1.5: Let S be the state space of a data structure, s_0 be the initial state, and $\phi: S \rightarrow \mathbb{R}_{\geq 0}$ be an arbitrary non-negative function. For some operation X and a state s with $s \rightarrow s'$ define

$$A_X(s) := T_X(s) + (\phi(s') - \phi(s)).$$

Then the functions $A_X(s)$ form a family of amortized time bounds.

Amortized Analysis

To show: $T(F) \leq c + \sum_{i=1}^{n} A_{Op_i}(s_{i-1})$

Proof:

$$\sum_{i=1}^{n} A_{Op_i}(s_{i-1}) = \sum_{i=1}^{n} [T_{Op_i}(s_{i-1}) + \phi(s_i) - \phi(s_{i-1})]$$
$$= T(F) + \sum_{i=1}^{n} [\phi(s_i) - \phi(s_{i-1})]$$
$$= T(F) + \phi(s_n) - \phi(s_0)$$
$$\Rightarrow T(F) = \sum_{i=1}^{n} A_{Op_i}(s_{i-1}) + \phi(s_0) - \phi(s_n)$$
$$\leq \sum_{i=1}^{n} A_{Op_i}(s_{i-1}) + \phi(s_0)$$

constant

Amortized Analysis

The potential method is universal!

Theorem 1.6: Let $B_X(s)$ be an arbitrary family of amortized time bounds. Then there is a potential function ϕ so that $A_X(s) \leq B_X(s)$ for all states s and all operations X, where $A_X(s)$ is defined as in Theorem 1.5.

Problem: find suitable potential function!

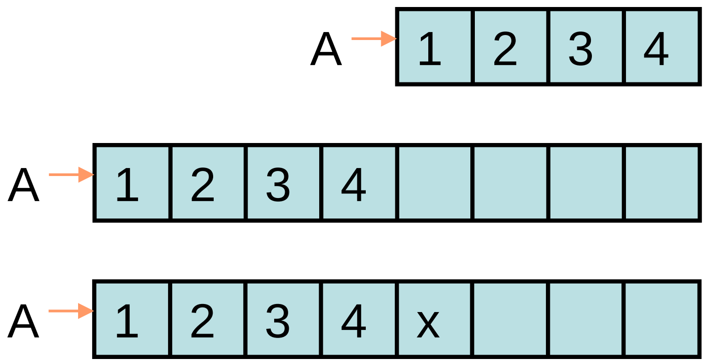
Once found, the rest is often easy.

Example: Dynamic Array

w: current size of array A

Insert(x):
  if n=w then
    A:=reallocate(A,2w)
  n:=n+1
  A[n]:=x

n=w=4:



reallocate(A,w): allocate array B of size w, copy contents of A to B, and return B

Example: Dynamic Array

Remove(i): remove i-th element

Remove(i):

A[i]:=A[n]

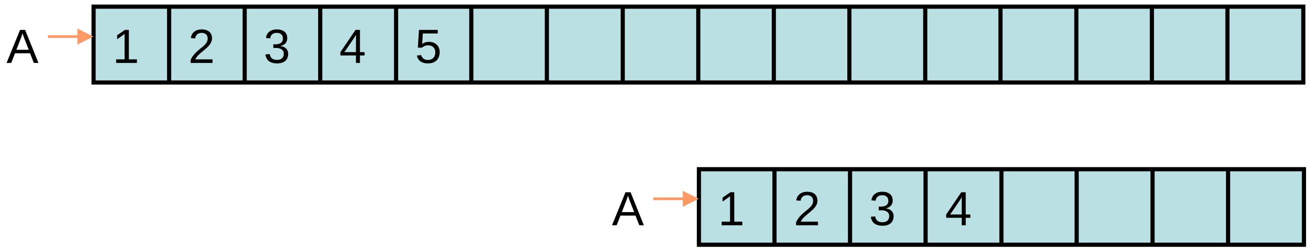
A[n]:=nil

n:=n-1

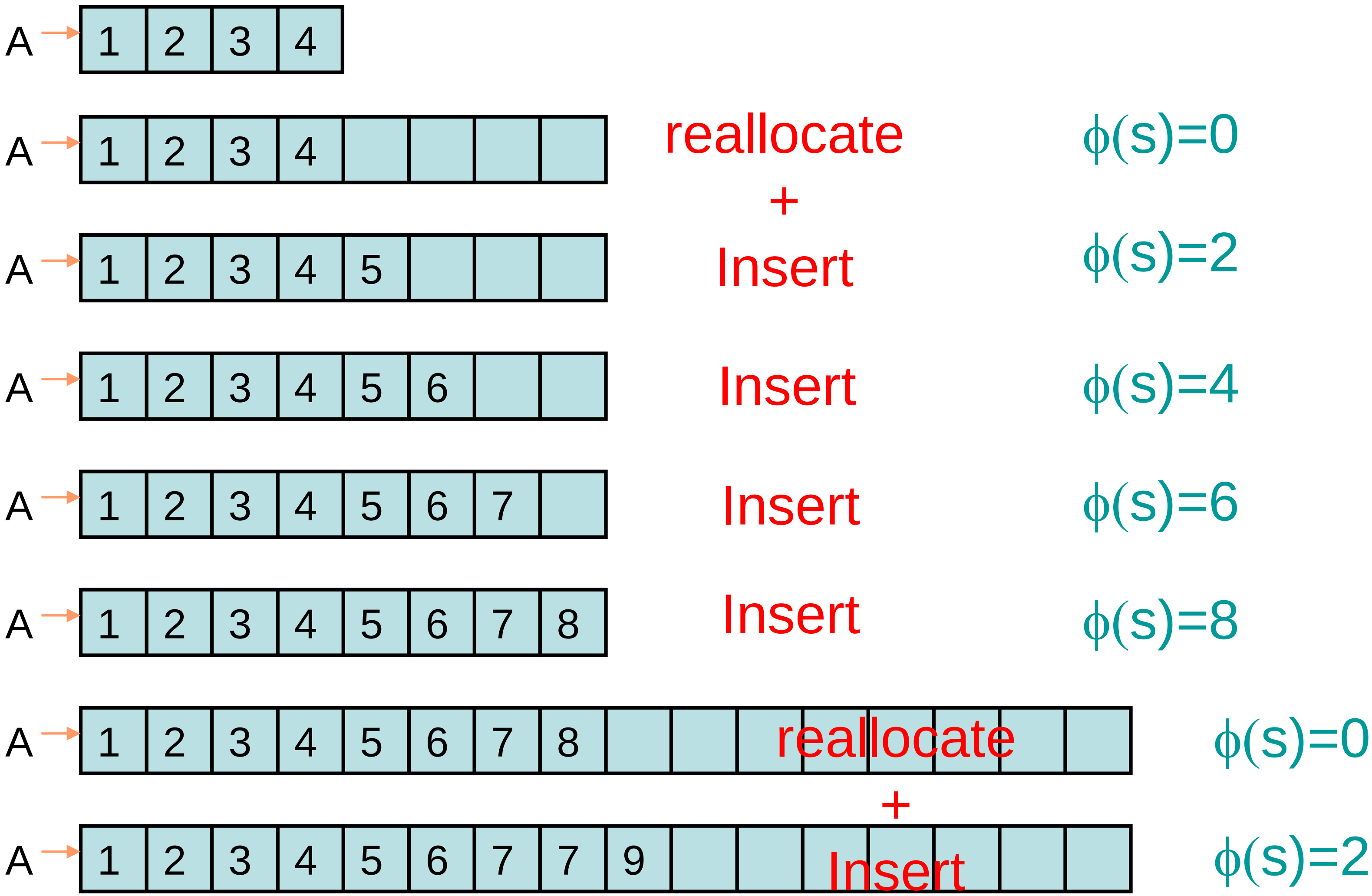
if $n \leq w/4$ and $n>0$ then

A:=reallocate(A,n/2)

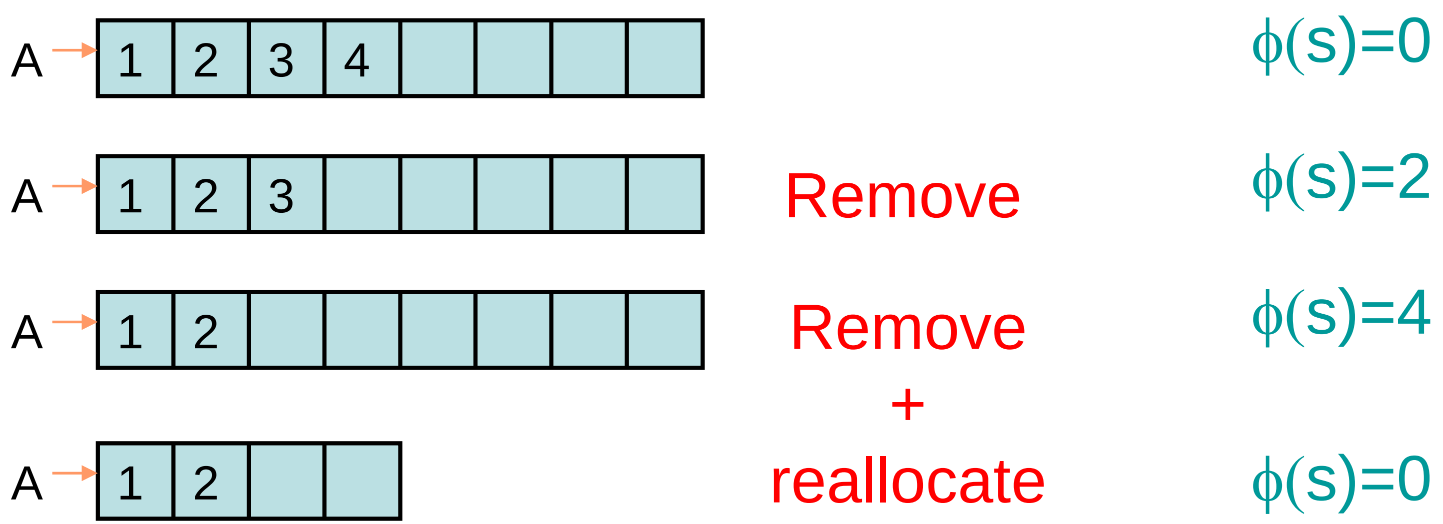
n=5, w=16:



Example: Dynamic Array



Example: Dynamic Array

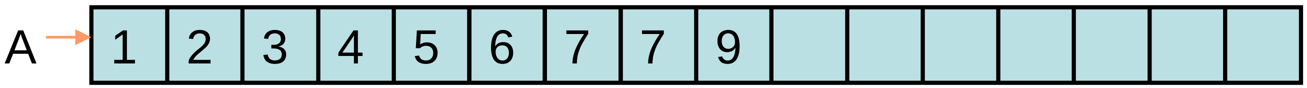


General formula for $\phi(s)$:

(w_s: size of A, n_s: number of entries)

$$\phi(s) = 2|w_s/2 - n_s|$$

Example: Dynamic Array



- formula for $\phi(s)$: $\phi(s) = 2|w_s/2 - n_s|$
- $T_{Insert}(s)$, $T_{Remove}(s)$: runtime of Insert and Remove without reallocate
- set time units so that $T_{Insert}(s) \leq 1$, $T_{Remove}(s) \leq 1$, and $T_{realloc}(s) \leq n_s$

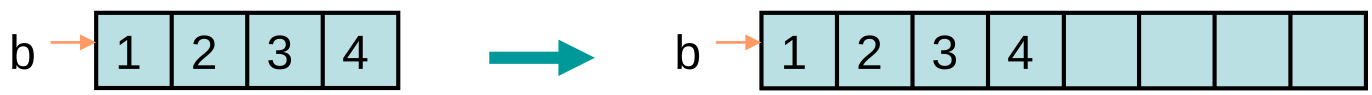
Theorem 1.6:
Let $\Delta\phi = \phi(s´)-\phi(s)$ for $s \rightarrow s´$

- ϕ non-negative, $\phi(s_0)=1$ (w=1, n=0)
- $A_{Insert}(s) = T_{Insert}(s) + \Delta\phi \leq 1+2 = 3$
- $A_{Remove}(s) = T_{Remove}(s) + \Delta\phi \leq 1+2 = 3$
- $A_{realloc}(s) = T_{realloc}(s) + \Delta\phi \leq n_s+(0-n_s) = 0$

Example: Dynamic Array

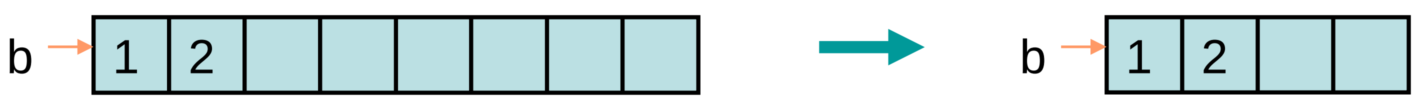
Proof of $A_{realloc}(s) \leq 0$:

- Case 1:



$\phi(s)=n_s$ $\phi(s')=0$

- Case 2:



$\phi(s)=2n_s$ $\phi(s')=0$

Example: Dynamic Array

Recall:

- S: state space of data structure
- F: sequence of operations Op_1, Op_2, Op_3,…,Op_n
- Total runtime $T(F) = \sum_{i=1}^{n} T_{Op_i}(s_{i-1})$
- For a family of amortized time bounds $A_{Op}(s)$,

$$T(F) \leq A(F) := c + \sum_{i=1}^{n} A_{Op_i}(s_{i-1})$$

  for some constant c independent of F

Hence, for any sequence F of n Insert and Remove operations on a dynamic array, $T(F)=O(n)$.