Basics of Algorithm Theory

Asymptotic Notation

Notation Notation	Definition
Big-O	$O(f) := \{g : \mathbb{N} \to \mathbb{R}^+ \quad \vert \quad \exists c \in \mathbb{R}^+ \quad \exists n_0 \in \mathbb{N} \quad \forall n \geq n_0 \quad g(n) \leq c \cdot f(n)\}$
Big-Omega	$\Omega(f) := \{g : \mathbb{N} \to \mathbb{R}^+ \quad \vert \quad \exists c \in \mathbb{R}^+ \quad \exists n_0 \in \mathbb{N} \quad \forall n \geq n_0 \quad c \cdot f(n) \leq g(n)\}$
Big-Theta	$\theta(f) := \{g : \mathbb{N} \to \mathbb{R}^+ \quad \vert \quad \exists c_1, c_2 \in \mathbb{R}^+ \quad \exists n_0 \in \mathbb{N} \quad \forall n \geq n_0 \quad c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n)\}$
Small-Oh	$\Omega(f) := \{g : \mathbb{N} \to \mathbb{R}^+ \quad \vert \quad \forall c \in \mathbb{R}^+ \quad \exists n_0 \in \mathbb{N} \quad \forall n \geq n_0 \quad g(n) \leq c \cdot f(n)\}$
Small-Omega	$O(f) := \{g : \mathbb{N} \to \mathbb{R}^+ \quad \vert \quad \forall c \in \mathbb{R}^+ \quad \exists n_0 \in \mathbb{N} \quad \forall n \geq n_0 \quad g(n) \geq c \cdot f(n)$

Sample growth rates

Function	Asymptotic complexity	Proof
$\theta(log_\alpha(n))$	$\theta(log_\beta(n))$	$log_\alpha(n) = \frac{1}{\log_\beta(\alpha)} \cdot log_\beta(n)$ for all $\alpha, \beta \in \mathbb{R}^+ \setminus \{1\}$ and all $n \in N$
$\theta(log(n!))$	$\theta(n \cdot log(n))$	Using Stirling's formula: $n! \approx \sqrt{2\pi n}(\frac{n}{e})^n$
$\theta(\sum_{K=1}^{n} \frac{1}{n})$	$\theta(ln(n)) = \theta(log(n))$	$lim_{n \to +\infty}(H_n - ln(n)) = \gamma$

Ackermann Function

The Ackermann function grows extremely rapidly. For example: $A(4,3) = 2^{2^{65536}} - 3$

$$A(m,n) := \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1, 1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m, n-1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

Inverse Ackermann Function

The inverse Ackermann function grows extremely slowly. It is at most four for any input of practical relevance.

$\alpha(n) := \min\{m \in \mathbb{N} : A(m,m) \geq n\}$

Real-world occurence of $O(\alpha)$: Combinatorial complexity of lower envelope of n line segments.

Log-star (Iterated logarithm)

Log-star also grows very slowly. It's less than 6 for any input of practical relevance

$$log^*(n) := \begin{cases} 0 & \text{if } n \leq 1 \\ 1 + log^*(log(n)) & \text{if } n > 1 \end{cases}$$

Example:

$$log^*(2^{16}) = 1 + log^*(2^{2^2}) = 2 + log^*(2^2) = 3 + log^*(2^1) = 4 + log^*(1) = 4$$

However, we have $\alpha \in o(log^*)$

Growth rate: $k + \epsilon$

A term of the form $k + \epsilon$ means that there is some additive positive constant $\epsilon \in \mathbb{R}^+$ that needs to be added to k. The constant ϵ may be regarded as arbitary small but it will never equal zero.

Suppose that some algorithm runs in $2^c n^{1+1/c}$ time, where $c \in \mathbb{R}^+$ is a user-chosen constant:

- For $c := 2$, the complexity term equals $4n^{3/2}$, which is in $O(n^{3/2})$
- For $c := 9$, the complexity term equals $2^9 n^{10/9}$, which is in $O(n^{10/9})$
- It is easy to see that $1 + 1/c$ approaches 1 as c approaches infinity. However, c cannot be set to infinity (or made arbitarily large) since then the 2^c term would dominate the complexity of our algorithm.
- Hence, the best possible way to express this complexity is to write $O(n^{1+\epsilon})$

Master Theorem

Consider constants $n_0 \in \mathbb{N}$ and $a, b \in \mathbb{N}$ with $b \geq 2$ and a function $f : \mathbb{N} \to \mathbb{R}_0^+$.

Let $f : \mathbb{N} \to \mathbb{R}_0^+$ be an eventually non-decreasing function such that

$$T(n) = a \cdot T(\tfrac{n}{b}) + f(n)$$

for all $n \in \mathbb{N}$ with $n \geq n_0$, where we interpret $\frac{n}{b}$ as either $\lceil \frac{n}{b} \rceil$ and $\lfloor \frac{n}{b} \rfloor$.

Then we have

$$T \in \begin{cases} \theta(f) & \text{if } f \in \Omega(n^{log_b(a)+\epsilon}) \\ \theta(n^{log_b(a) \cdot log(n)}) & \text{if } f \in \theta(n^{log_b(a)}) \\ \theta(n^{log_b(a)}) & \text{if } f \in O(n^{log_a(b)-\epsilon}) \end{cases}$$

Complexity of an algorithm

Worst-case complexity

A *worst-case complexity* of an algorithm is a function $f : \mathbb{N} \to \mathbb{R}^+$ that gives an upper bound on the number of elementary operations (memory units, ...) used by an algorithm with respect to the size of its input, for all inputs of the same size.

Average-case complexity

An *average-case complexity* of an algorithm is a function $g : \mathbb{N} \to \mathbb{R}^+$ that models the average number of elementary operations (memory units, ...) used by an algoirhtm with respect to the size of its input.

Input size

The *size of the input* of an algorithm is a quantity that measures the number of input items relevant for elementary operations of the algorithm.

Model of Computation

Elementary Operation

An *elementary operation* is an operation whose running time is assumed not to depend on the specific values of its operands.

For example: The time taken by the comparison of two floating-point numbers is frequently assumed to be constant.

Still, what constitutes an elementary operation depends on the model of computation.

Model of computation

A model of computation specifies the elementary operations that may be executed together with their respective costs.

For example: Turing Machine (TM) model. This model is often used when talking about theoretical issues like NP-completeness. But, the TM model is cumbersome to use for analyzing actual complexities and impractical for most "real" applications

Hence, several alternative models have been proposed:

- Random Access Machine (RAM) model
- Real RAM model
- Algebraic Decision/Computation Tree (ADT/ACT) model
- Blum-Shub-Smale model

Integer Random Access Machine

Every memory location can hold a single integer number.

The following operations are available at unit cost:

- Reading and writing (i.e., file I/O) of a number
- The three standard arithmetic operations: $+, -, \times$
- Comparison between two integers: $<, \leq, =, \neq, \geq, >$
- Indirect addressing of memory

Note: The RAM model is not very very realistic. For instance, it allows solving PRIME in $O(nlog(log(n))$ time using the sieve of Eratosthenes.

Real random Access Machine

Every memory location can hold a single real(!) number.

The following operations are available at unit cost:

- Reading and writing (i.e., file I/O) of a real number
- The four standard arithmetic operations and square roots: $+, -, \times, \div, \sqrt{}$
- Comparisons between two reals: $<, \leq, =, \neq, \geq, >$
- Indirect addressing of memory (for integer addresses only!)

Real RAM is a widely used model of computation, e.g., in geometric modeling and computational geometry. It's still unrealistic, though, since just one real number can encode an infinite amount of information.

The Real RAM is truly more powerful than the RAM. For example, given two finite sets S and T of positive integers, decide whether $\sum_{s \in S} \sqrt{s} > \sum_{t \in T} \sqrt{t}$

Floor function

A real RAM does not allow to convert between integer variables and real variables. In particular, it does not allow the use of the floor function $\lfloor \cdot \rfloor$

MAXGAP

Given: A set S of $n \geq 2$ distinct (unsorted!) real numbers.

Compute: The maximum gap δ among the sorted numbers of S.

$\delta := max\{y - x : x, y \in S \wedge x \leq y \wedge \neg(\exists z \in S : x < z < y)\}$

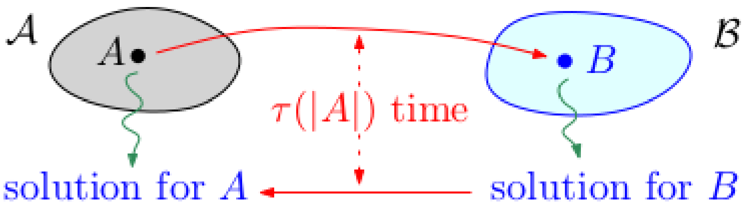
MAXGAP has an $\Omega(nlogn)$ lower bound in the ACT model. Adding the floor function to the Real RAM allows to solve MAXGAP in $\theta(n)$ time.

Reduction of a problem

Reduction

A problem $\mathcal{A}$ can be reduced (or transformed) to a problem $\mathcal{B}$ if

- every instance A of $\mathcal{A}$ can be converted to an instance B of $\mathcal{B}$
- a solution S for $\mathcal{B}$ can be computed, and
- S can be transformed back into a correct solution for $\mathcal{A}$.



τ-Reducibility

A problem A is τ-reproducible to B, denoted by $A \leq_\tau B$, if

- $\mathcal{A}$ can be reduced to B
- for any instance A of $\mathcal{A}$, steps 1 and 3 of the reduction can be carried out in at most $\tau(|A|)$ time, where $|A|$ denotes the input size of A.

Upper bound via reduction

Suppose that A is reducible to B such that the order of the input size is preserved. If problem B can be solved in $O(T)$ time, then A can be solved in at most $O(T + \tau)$ time.

Lower bound via reduction

Suppose that A is τ-reducible to B such that the order of the input size is preserved. If problem A is known to require $\Omega(\tau)$ time, then B requires at least $\Omega(T - \tau)$ time.

Intuitive explanation

Let's assume that we are given an problem A and we know that its complexity has a lower bound of $\Omega(T_A)$. It's important to emphasize that we can rely on this bound. On other words, we know that this bound is correct due to some proof we or somebody else has done previously.

Next, we reduce A to a problem B. In other words, we find a transformation that can map any instance of A to an instance of B. Now, let's assume that problem B can be solved faster than T_A and the transformation can done efficently (faster than $O(T_A)$). However, this would mean that we can solve A by reducing it to B faster than $\Omega(T_A)$ time.

Contradiction: We know that such a solution for A does not exist, hence, we also know that such a solution for B cannot exist. Therefore, B needs to have a lower bound of at least $\Omega(T - \tau)$.

Example: ELEMENTUNIQUENESS and SORTING

PROBLEM: ELEMENTUNIQUNESS

Given: A set S of n real numbers $x_1, x_2, \ldots, x_n$

Decide: Are any two numbers of S equal?

LEMMA:

- ELEMENTUNIQUENESS can be solved in $O(f) + O(n)$ time if we can sort n numbers in $O(f)$ time.
- SORTING requires $\Omega(f)$ time if ELEMENTUNIQUENESS requires $\Omega(f)$ time.

Proof:

Given:

- We know that SORTING can be solved in $O(f)$ time
- We know that ELEMENTUNIQUENESS requires at least $\Omega(f)$ time

Obviously, after sorting $x_1, x_2, \ldots, x_n$ elements, we can solve ELEMENTUNIQUENESS in $O(n)$ time. Rest follows from definition of reduction.

Example: CLOSESTPAIR and ELEMENTUNIQUENESS

PROBLEM: CLOSESTPAIR

Given: A set of S of n points in the Euclidean plane

Compute: Those two points of S whose mutal distance is minimum among all all pairs of points of S.

LEMMA:

- ELEMENTUNIQUENESS can be solved in $O(f) + O(n)$ time if CLOSESTPAIR can be solved in $O(f)$ time.
- CLOSESTPAIR requires $\Omega(f)$ time if ELEMENTUNIQUENESS requires $\Omega(f)$ time.

Proof:

Given:

- We know that CLOSESTPAIR can be solved in $O(f)$ time.
- We know that ELEMENTUNIQUENESS requires at least $\Omega(f)$ time.

Reduce ELEMENTUNIQUENESS to CLOSESTPAIR:

- Let $S' := \{x_1, x_2, \ldots, x_n\}$ be an element of ELEMENTUNIQUENESS. Transform S' into an element of CLOSESTPAIR by mapping every x_i to an element $(x_i, 0)$. Obviously, the elements are unique if and only if the closest pair has non-zero distance. This transformation can be conducted in $O(n)$ time.

Example: SORTING and CONVEXHULL

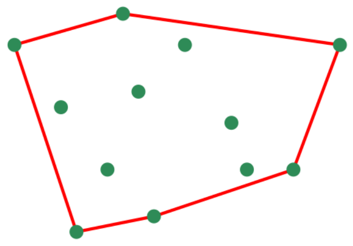
PROBLEM: CONVEXHULL

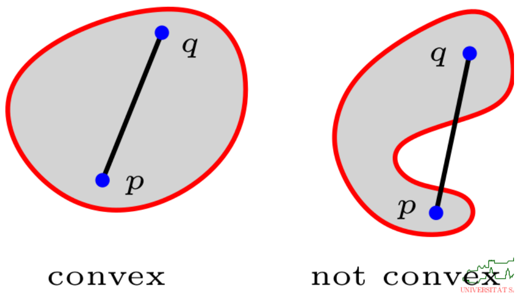
Given: A set of S of n points in the Euclidean plan $\mathbb{R}^2$

Compute: The convex hull $CH(S)$ (the smallest convex super set of S)

Convex set

A set $X \subset \mathbb{R}^2$ is *convex* if for every pair of points $p, q \in X$ also the line segment $\bar{pq}$ is contained in X.





LEMMA:

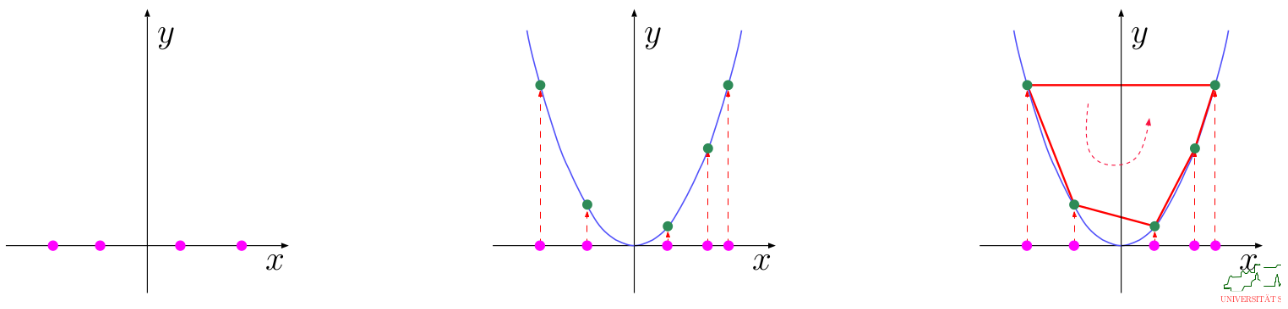
- CONVEXHULL requires $\Omega(f)$ time if SORTING requires $\Omega(f)$ time.

Proof:

Given:

- We know that SORTING requires at least $\Omega(f)$ time

Suppose that $S' := \{x_1, x_2, \ldots, x_n\}$ is an instance of SORTING. We transform it into an instance of CONVEXHULL by mapping every point x_i to (x_i, x_i^2). The convex hull of S contains a list of vertices sorted by x-coordinates. One pass through this list will find the smallest element. The sorted numbers can be obtained by a second pass through this list, at a total extra cost of $O(n)$ time.



Decision Problems

Decision Problem

A problem is a *decision problem* if the output sought for a particular instance of the problem always is the answer YES / NO.

Example: Boolean satisfiability (SAT), "Are the numbers sorted?"

Problem: SAT

Given: A propositional formula A.

Decide: Is A satisfiable? For instance, does there exist an assighnment of truth values to the Boolean variables of A such that A evaluates to true?

Algebraic Decision Trees

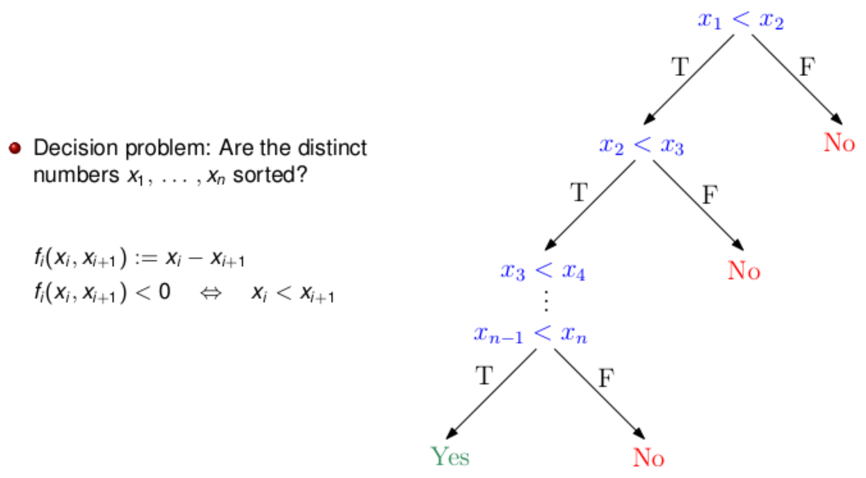
An *algebraic decision tree (ADT)* with input vector $(x_1, x_2, \ldots, x_n) \in \mathbb{R}^n$ is a finite rooted binary tree where

- every internal node v is given by a predicate

  $f_v(x_1, x_2, \ldots, x_n) \quad ? \quad 0$
- such that
  - $f_v : \mathbb{R}^n \rightarrow \mathbb{R}$ is a polynomial in $x_1, \ldots, x_n$ of degree deg(f_v), and
  - "?" is a comparison operator ($<, \leq, \geq, >, \neq$)
- every leaf node is associated with Yes and No

It solves a decision problem P if its leaves are associated correctly with Yes and No relative to P for every input $(x_1, x_2, \ldots, x_n)$. The algebraic degree of a decision tree with k internal nodes $v_1, \ldots, v_k$ is given by the maximum degree of a polynomial at a node: max$_{1 \leq i \leq k}$ deg(f_{v_i}). An ADT is a *linear decision tree* if its algebraic degree is 1.

Example: Linear Decision Tree



- Decision problem: Are the distinct numbers $x_1, \ldots, x_n$ sorted?

$f_i(x_i, x_{i+1}) := x_i - x_{i+1}$

$f_i(x_i, x_{i+1}) < 0 \quad \Leftrightarrow \quad x_i < x_{i+1}$

Note that every comparison $x_1 < x_2$ splits $\mathbb{R}^2$ into two half-planes. It yields true if and only if $(x_1, x_2) \in \mathbb{R}^2$ is a member of $W := \{(u_1, u_2) \in \mathbb{R}^2 : u_1 < u_2\}$.

Membership set

For a decision problem P with input variables $x_1, x_2, \ldots, x_n \in \mathbb{R}^2$ we define W_P as the set of points in $\mathbb{R}^n$ for which the answer to the decision problem is Yes.

$$W_P := \{(u_1, \ldots, u_n) \in \mathbb{R}^n : u_1, u_2, \ldots, u_n \text{ yield } Yes \text{ for } P\}$$

The set W_P is called the *membership set* of P. Furthermore, we denote $\bar{W_P}$ as the set of points in $\mathbb{R}^n$ for which the answer is No. Formally, we can define $\bar{W_P}$ as $\bar{W_P} := \mathbb{R}^n \setminus W_P$

Definition (45)

For a decision problem P with input $x_1, x_2, \ldots, x_n \in \mathbb{R}$ and membership set W_P we denote the number of disjoint connected components of W_P by $\#(W_P)$ and the number of disjoint connected components of $\bar{W_P}$ by $\#(\bar{W}_P)$.

Linear Tree

In a linear decision tree at every internal node v the predicate

- $f_v(x_1, x_2, \ldots, x_n) \quad ? \quad 0$

corresponds to

- $a_0 + a_1x_1 + a_2x_2 + \ldots + a_nx_n \quad ? \quad 0$

Hence, every internal nodes defines a hyper-plane and we branch at that node depending on whether the input point is above, below, or on this hyper-plane

Putting a lower bound on the height of the decision tree

To investigate how complicated a problem is we need to identify how many comparisons are required to "solve" a problem. Since we branch at every node, we need to look into the height of the decision tree.

Now, let's consider the set $R(v) \subseteq \mathbb{R}^n$ of points of $\mathbb{R}^2$ that reach some particular node v of a linear decision tree. The set $R(v)$ contains all the points that satisfy a set of linear equalities and inequalities. Hence, $R(v)$ forms a convex polyhedron.

Next, we can conclude that every leaf v of a linear decision tree corresponds to a convex set and connected set $R(v) \subseteq \mathbb{R}^n$. (By definition: A convex set is connected. And the intersection of two convex sets is convex.)

Hence, every disjoint connected component of W_p or $\bar{W_P}$ corresponds to one unique leaf of the decision tree. This implies a lower bound of $\#W_P + \#\bar{W}_P$ on the height of the decision tree and, therefore, also on the minimum number of comparisons that any decision algorithm which uses linear decisions has to make in the worst case.

Theorem (46): Dobkin & Lipton

The height h of a linear decision tree is $h \geq log(\#W_P + \#\bar{W}_P)$.

Theorem (47): Petrovskii & Oleinik, Thom, Milnor

Given a decision problem P with n variables. We get for the height h of an algebraic decision tree of degree $d \geq 2$ that solves P: $\qquad h = \Omega(log_d(\#W_P + \#\bar{W}_P) - n)$

Algebraic Computation Tree

An *algebraic computation tree* with input $(x_1, x_2, \ldots, x_n) \in \mathbb{R}^n$ solves a decision problem P if it is a finite rooted tree with at most two children per node and two types of internal nodes.

Computation:

A computation node v has a value f_v determined by one of the following instructions:

- $f_v = f_u \circ f_w$ or $f_v = \sqrt{f_u}$

where $\circ \in \{+, -, \cdot, \backslash\}$ and f_u, f_w are values associated with ancestors of v, input variables or arbitrary real constants.

Branch:

A branch node v has two children and contains one of the predicates

- $f_u > 0$ $f_u \geq 0$ $f_u = 0$

where f_u is a value associated with an ancestor of v or an input variable.

Every leaf node is associated with Yes and No, depending on the correct answer for every $(x_1, x_2, \ldots, x_n)$ relative to P.

Theorem (49)

Given a decision problem P on n variables. If we exclude all intermediate nodes which correspond to additions,, substractions and multiplications by constants then we get for the height h of an algebraic computation tree that solves P:

$h = \Omega(log(\#W_P + \#\bar{W}_P) - n)$\$

Proofing a lower bound for ELEMENTUNIQUENESS

The following algorithm can solve ELEMENTUNIQUENESS without sorting the input first:

$\prod_{1 \leq i \leq j \leq n}(x_i - x_j)\ ?= 0$

Case study if $n = 3$:

First, let's investigate our problem for $n = 3$:

- $x_1 < x_2 < x_3$
- $x_1 < x_3 < x_2$
- $x_2 < x_1 < x_3$
- $x_2 < x_3 < x_1$

- $x_3 < x_1 < x_2$
- $x_3 < x_2 < x_1$

If any of these inequalities is true, then all numbers are different and the answer to our decision problem is No.

In other words, the subset $\bar{W_P}$ of $\mathbb{R}^3$ for which the answer is No:

$$\bar{W_P} := \cup_{\pi \in S_3} \bar{W_\pi}$$

with $\bar{W_\pi} := \{(x_1, x_2, x_3) \in \mathbb{R}^3 : x_{\pi(1)} < x_{\pi(2)} < x_{\pi(3)}\}$

Hence, for $n = 3$ the number of disjoint connected components $\#\bar{W_P} = 6$ because each permutation π results in its own connected component.

The general case:

We have know shown that $n = 3$ leads to 6 disjoint connected components. However, to make sure that it suits for our problem, we need to generalize it. In fact, we need to show that every subset in $\bar{W_P}$ forms its own disjoint connected component.

- Let $\pi, \sigma \in S_n$ with $\pi \neq \sigma$. For $1 \leq i, j \leq n$ we define $f_{ij}(x_1, x_2, \ldots, x_n) := x_i - x_j$
- We can conclude that since $\pi \neq \sigma$, there exists $i \neq j$ such that

  $f_{ij} > 0$ for all $p \in \bar{W_\pi}$ but $f_{ij}(p) < 0$ for all $p \in \bar{W_\sigma}$
- Are these points part of the same connect set? Let's recall what being in the same connected set actually means: It means that every point on a line connecting two points in a set is part of the connected set as well.

  In our case, any path from a point in $\bar{W_\pi}$ to a point in W_π must go through a point q where $f_{ij}(q) = 0$.

  But we know that $q \in \bar{W_P}$. Hence, $\bar{W_\pi}$ and $\bar{W_\sigma}$ lie in two different connected components if $\pi \neq \sigma$.
- Since $|S_n| = n!$, we know that $\#\bar{W_P} \geq n!$

Finally, we can now conclude that the height h of an ADT/ACT is $h = \Omega(log(n!) - n)$

This also means that $\Omega(log(n)!) = \Omega(n \cdot log(n))$ comparisons are necessary to solve ELEMENTUNIQUENESS in any ADT/ACT of fixed maximum degree for n input numbers.

Theorem (50)

A comparison-based solution of ELEMENTUNIQUENESS for n real numbers requires $\Omega(n \cdot log(n))$ comparisons.

Corollary (51)

A comparison-based SORTING of n real numbers requires $\Omega(n \cdot log(n))$ comparisons.

Corollary (52)

A solution to CONVEXHULL computation requires $\Omega(n \cdot log(n))$ time in the ACT model in the worst case for n points.

Corollary (53)

A solution to CLOSESTPAIR requires $\Omega(n \cdot log(n))$ time in the ACT model in the worst case for n points.

Proofing a lower bound for MAXGAP and UNIFORMGAP

Problem: MAXGAP

Given: A set of S of $n > 2$ distinct (unsorted!) real numbers.

Compute: The maximum gap δ among the sorted numbers of S.

$\delta := max\{y - x : x, y \in S \wedge x < y \wedge \neg(\exists z \in S : x < z < y)\}$

Problem: UNIFORMGAP

Given: A set $s := \{x_1, x_2, \ldots, x_n\}$ of $n > 2$ distinct (unsorted!) real numbers and $\delta \in \mathbb{R}^+$.

Decide: Is the gap between the sorted numbers of S uniformly δ?

$\exists \pi \in S_n \; x_{\pi(i+1)} = x_{\pi(i)} + \delta$ for all $i \in \{1, 2, \ldots, n-1\}$

Proofing the lower bound for UNIFORMGAP

We first start by defining the membership set W_P which includes all points for which the result of the ACT would be Yes.

The membership set is given by $W_P := \cup_{\pi \in S_n} W_\pi$

with $W_\pi := \{(x_1, x_2, \ldots, x_n\}) \in \mathbb{R}^n : x_{\pi(i+1) = x_{\pi(i)} + \delta}$ for all $i \in \{1, 2, \ldots, n-1\}$

We now consider the function $f_{ij} := x_i - x_j$ for $1 \leq i, j \leq n$.

For $\pi, \delta \in S_n$: If $\pi \neq \sigma$ then at least one f_{ij} must take different signs over W_π and W_δ.

Hence, any path leading from a point W_π to a point in W_δ mast pass through a point p of $\mathbb{R}^n$ with $f_{ij}(p) = 0$. However, $p \notin W$ since δ is a real positive number. Therefore, the distance between two elements x_i and x_j can never be 0.

Therefore, we can conclude that W_π and W_σ form different connected components of W if $w \neq \delta$.

Since $|S_n| = n!$, we get $\#W_P \geq n!$ which establishes the lower $\Omega(n \cdot log(n))$ lower bound.

Theorem (54)

A solution to UNIFORMGAP for n numbers requires $\Omega(n \cdot log(n))$ time in the ACT model.

Proofing the lower bound for MAXGAP

We can show a lower bound for MAXGAP by reducing UNIFORMGAP to MAXGAP.

Given an instance $\{x_1, x_2, \ldots, x_n\}$ and δ of the UNIFORMGAP problem. We first use MAXGAP algorithm to compute the maximum gap Δ of n numbers.

If $\Delta \neq \delta$ then the answer is No. Otherwise, we compute $x_{min} := min\{x_1, x_2, \ldots, x_n\}$ and $x_{max} := max\{x_1, x_2, \ldots, x_n\}$ and check whether $x_{max} - x_{min} = (n-1) \cdot \delta$.

Obviously, this algoirthm solves UNIFORMGAP in $O(n)$ time plus the time consumed by the MAXGAP algorithm.

Corollary (55)

A solution to MAXGAP for n numbers requires $\Omega(n \cdot log(n))$ time in the ACT model.

Adversary Strategy

Adversary strategies are used to come up with a "model" of the worst case. To get an understanding of how this works we take a closer look at our SORTING problem.

Let's consider two player A and B who play the following game: A thinks of n distinct real numbers and B tries to sort those number by comparing paris of two numbers. To accomplish that B is allowed to ask questions of the form "Is the third number greater than the fifth number?" A needs to answer truthfully and consistently, but he's allowed to make B's life as hard as possible. In other words, he is allowed to replace his originally chosen n numbers by new n-tuple at any time, provided that the new numbers are consistent with the answers that A has given so far.

What is a lower bound on the number of comparisons that A can force B to make?

Strategy:

We assume that A stores the n numbers in an array a[1,...,n] and that B will sort the numbers by comparing some element a[i] to some other element a[j] by asking A whether a[i] < a[j].

Since the adversary A is allowed to pick the input, the adversary A keeps a set S of permutations that are consistent with the comparisons B has made so far.

The answer of A to a comparison "Is a[i] < a[j]?" is chosen as follows:

- Let $Y \subset S$ be those permutations that have remained in S and that are also consistent with a[i] < a[j].
- Furthermore, $N := S \setminus Y$
- If $|Y| \geq |N|$ then the adversary A prefers to answer "yes" and then replaces S by Y.
- Otherwise, "no" is answered and S is replaced by N.

This strategy allows A to **keep at least half of the permutations** after every comparison of the algorithm B.

Player B cannot declare the order of the numbers to be known as long as $|S| > 1$.

Thus, B needs at least $\Omega(log(n!)) = \Omega(n \cdot log(n))$ comparisons which establishes the lower bound sought.

Amortized Analysis

Amortized analysis is a worst-case analysis of a sequence of different operations performed on a data structure or by an algorithm. It's applied if a costly operation cannot occur for a series of operations in a row. With traditional worst-case analysis, the resulting bound on the running time of such a sequence of operations is too pessimistic if the execution of a costly operation only happens after many cheap operations have already been carried out.

The goal of amortized analysis is to obtain a bound on the overall or average cost per operation in the sequence which is tighter than what can be obtained by separately analyzing each operation in the sequence.

Amortized analysis, therefore, considers arbitary sequences and, in particular, worst-case sequences. It gives genuine upper bounds: The amortized cost per operation times the number of operations yields to a worst-case bound on the total complexity of those operations. This guarantees the average performance of each operation in the worst case.

Note that an **average-case analysis** typically averages over the input and depends on assumptions on probability distributions to obtain an *estimated cost* per operation.

Dynamic Array

- A *dynamic array* is a data structure of variable size that allows to store and retrieve elements in a random-access mode.
- Elements can also be inserted at and deleted from the end of the array.
- The size of the array is not fixed. New memory can be allocated at any time
- Simple realization of a dynamic array: Use array of fixed size and reallocate whenever needed to increase (or decrease) the size of the array.
  - Size: Number of contiguous elements stored in the dynamic array
  - Capacity: Physical size of the underlying fixed-size array
- Insertion of an element at the end of the array:
  - Constant-time operation if the size is less than the capacity
  - Costly if the dynamic array needs to be resized since this involves allocating a new underlying array and copying each element from the original array

A simple resizing strategy

Suppose that the initial capacity of the array is 1. A simple strategy for increasing the capacity would be to increase the capacity whenever the size of the array gets larger than its capacity.

In the worst case, a sequence of n array operations consists of only insertations at the end of the array, at a cost of k for the insertion of the $k-th$ element into an array of size $k-1$.

Hence, we get $1+2+...+n=\frac{n(n+1)}{2} \in O(n^2)$ as total worst-case complexity for n insert operations.

An improved resizing strategy

To avoid the costs of frequent resizing we expand the array by a constant factor α whenever we run out of space. For example, we could double its size. Amortized analysis allows to show that the (amorized) cost per array operation is reduced to $O(1)$. Please note, that the best value for the growth factor is a topic of frequent discussions.

Example:

- C++ std:vector
  - GCC 5.2: $\alpha = 2$
  - Microsoft VC++: $\alpha = \frac{3}{2}$
- Java ArrayList: $\alpha = \frac{3}{2}$

Amortized cost

The *amortized cost* of one operation out of a sequence of operations (for some $n \in \mathbb{N}$) is the total (worst-case) cost for all operations divided by n.

IMPORTANT: Even if the amortized cost of one opreation is $O(1)$, the worst-case cost of one particular operation may be substantially greater! Hence, studying amortized costs might not be good enough when a guaranteed low worst-case cost per operation is required, for instance, in case of real-time or parallel systems.

Method: Aggregate method for analyzing dynamic arrays

Aggregate analysis determines an upper bound $U(n)$ on the total cost of a sequence of n operations. Then the amortized cost per operation is $U(n)/n$. This means that all types of operations performed in the sequence have the same amortized cost.

- Suppose that the initial capacity of the array is 1.
- The cost c_i of the i-th insertion is
  $$c_i = \begin{cases} i & \text{if } i-1 \text{ is a power of } 2 \\ 1 & \text{otherwise} \end{cases}$$
- Hence, we get for the cost of n insertions

$U(n) = \sum_{i=1}^{n} c_i = (\sum_{i=1;(i-1) \text{ is no power of } 2}^{n} 1) + \sum_{i=1;(i-1) \text{ is power of } 2}^{n} i)$

$= (\sum_{i=1;(i-1) \text{ is no power of } 2}^{n} 1) + \sum_{i=1;(i-1) \text{ is power of } 2}^{n} (i-1)+1)$

$= \sum_{i=1;(i-1) \text{ is no power of } 2}^{n} 1 + \sum_{i=1;(i-1) \text{ is power of } 2}^{n} (i-1) + \sum_{i=1;(i-1) \text{ is power of } 2}^{n} 1$

$= n + \sum_{i=1;(i-1) \text{ is power of } 2}^{n} (i-1)$

$= n + \sum_{i=0;i \text{ is power of } 2}^{n-1} (i)$

$= n + \sum_{i=0;i \text{ is power of } 2}^{n-1} (i)$

$= n + \sum_{i=0}^{\lfloor ld(n-1) \rfloor} 2^i \leq n + \sum_{i=0}^{\lfloor ld(n) \rfloor} 2^i$

$= n + 2^{\lfloor ld(n) \rfloor + 1} - 1 \leq n + 2 \cdot 2^{ld(n)} = n + 2n = 3n$

- Therefore, the amortized cost per (insertion) operation is $\frac{U(n)}{n} = \frac{3n}{n} \in O(1)$

Method: Accounting Method

One of the main shortcomings of aggregate analysis is that different types of operations are assigned the same amortized cost. As a natural improvement, one might want to assign different costs to different operations. The accouting method (aka banker's method) assigns charges to each type of operation.

In contrast, the accounting method seeks to find a payment of a number of extra time units charged to each individual operation such that the sum of the payments is an upper bound on the total actual cost. Intuitively, one can think of maintaining a bank account. Low-cost operations are charged a little bit more than their true cost, and the surplus is deposited into the bank account for later use. High-cost operations can then be charged less than their true cost, and the deficit is paid for by the savings in the bank account. In that way we spread the cost of high-cost operations over the entire sequence. The charges to each operation must be set large enough that the balance in the bank account always remains positive, but small enough that no one operation is charged significantly more than its actual cost.

To get a better understand of how the accounting method works we will now analyze the dynamic array using this banker's method.

Say it costs 1 unit to insert an element and 1 unit to move it when the table is doubled. Clearly a charge of 1 unit per insertion is not enough, because there is nothing left over to pay for the moving. A charge of 2 units per insertion again is not enough, but a charge of 3 might be.

Therefore, we try to set the charge c_i' for the i-th operation to 3 if its an insertion.

NO DEBT!

Denote the (real) cost of the i-th operation by c_i and the amortized cost by c_i'. Then we require

$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} c_i'$

for every $n \in \mathbb{N}$ and every sequence of n operations.

Example:

1. Insert Charge: 3 Cost: 1 Bank account: 2 Capacity: 1 Space left: 0
2. Insert Charge: 3 Cost: 2 Bank account: 3 Capacity: 2 Space left: 0
3. Insert Charge: 3 Cost: 3 Bank account: 3 Capacity: 4 Space left: 1
4. Insert Charge: 3 Cost: 1 Bank account: 5 Capacity: 4 Space left: 0
5. Insert Charge: 3 Cost: 5 Bank account: 3 Capacity: 8 Space left: 3
6. Insert Charge: 3 Cost: 1 Bank account: 5 Capacity: 8 Space left: 2

7. Insert Charge: 3 Cost: 1 Bank account: 7 Capacity: 8 Space left: 1
8. Insert Charge: 3 Cost: 1 Bank account: 9 Capacity: 8 Space left: 0
9. Insert Charge: 3 Cost: 9 Bank account: 3 Capacity: 16 Space left: 7
10. Insert Charge: 3 Cost: 1 Bank account: 5 Capacity: 16 Space left: 6

Formal proof:

We know need to proof formally that the bank account is always positive after the insertion of the i-th element, for all $i \in \mathbb{N}$.

I.B.: For $i = 1$ there is one element in the array and $c_1' - c_1 = 2$ in the bank account. For $i = 2$ we have two elements in the array and $2 + 3 - 2 = 3$ in the bank account.

I.H.: The bank account is positive after the insertion of the i-th element, for some arbitary but fixed $i \in \mathbb{N}$ with $i \geq 2$.

Suppose that $i = 2^k$ for $k \in \mathbb{N}$. By the I.H. we know that the bank account was not negative when we double from a capacity of 2^{k-1} to 2^k. After doubling we insert $2^k - 2^{k-1} = 2^{k-1}$ elements into the table of capacity. After the 2^{k-1} inserts we have at least $2 \cdot 2^{k-1} = 2^k$ "coins" on the bank account. Resizing the array from 2^k to 2^{k+1} requires to move 2^k elements. This can be accomplished by the 2^k coins on the bank account. The new charge ensures that we have at least $-1 = 2$ coins on the bank account after the insertion of element with number $i + 1 = 2^k + 1$.

Method: Potential Method

In case of the potential method we define a so-called *potential function* $\phi : \{A_i : 0 \leq i \leq n\} \to \mathbb{R}$ for a sequence of n operations. A_i is referred as data structure. The function needs to preserve the following properties:

- $\phi(A_0) = 0$
- $\phi(A_i) \geq 0$ for all $i \in \mathbb{R}$

Intuitively, the potential function will keep track of the precharged time at any point in the computation. It measures how much saved-up time is available to pay for expensive operations. It is analogous to the bank balance in the banker's method. But interestingly, it depends only on the current state of the data structure, irrespective of the history of the computation that got it into that state.

We determine the amortized costs c_i' as follows: $c_i' := c_i + \Delta\phi_i$

The charge of the i-th operation $\Delta\phi_i$ can be computed as follows: $\Delta\phi_i := \phi(A_i) - \phi(A_{i-1})$

If we consider a sequence of n operations of cost c_0, c_1, c_2, ... that produces data structures A_0, A_1, etc. then the total amortized cost of n operations is

$\sum_{i=1}^{n} c_i' = \sum_{i=1}^{n}(c_i + \phi(A_i) - \phi(A_{i-1})) = (\phi(A_n) - \phi(A_0)) + \sum_{i=1}^{n} c_i \geq \sum_{i=1}^{n} c_i$

This, if $\phi(A_i) \geq 0$ for all $n \in \mathbb{N}$ and $\phi(A_0) = 0$ then the total amortized costs are an upper bound on the total true costs.

But how to choose a proper potential function? It is obvious that an elaborate and clever choice of the potential function is a prerequisite for achieving a tight bound. Unfortunately, good potential functions tend to be hard to find . . .

Example:

For analyzing our dynamic array with the potential method we define the potential as follows:

$\phi(A_i) := 2i - 2^{\lceil log(i) \rceil}$ for $i \in \mathbb{N}_0$ Note that we interprete $2^{log(0)}$ as 0.

However, before we study the amortized costs is always good to verify that it preserves the properties required by the a potential function. In other words, $\phi(A_0)$ needs to be 0 and $\phi(A_i)$ needs to be greater or equal to 0.

- $\phi(A_0) := 2 \cdot 0 - 2^{\lceil 0 \rceil}$
- $\phi(A_i) = 2i - 2^{\lceil log(i) \rceil} \geq 2i - 2^{1+log(i)} = 2i + 2^1 \cdot 2^{log(i)} = 2i - 2i = 0$

Finally, we are ready to calculate the amortized cost for dynamic array insertion.

Recall: $c_i = \begin{cases} i & \text{if } i-1 \text{ is a power of } 2 \\ 1 & \text{otherwise} \end{cases}$

Amortized cost of the i-th insertion:

$c_i' = c_i + \Delta\phi_i = c_i + \phi(A_i) - \phi(A_{i-1})$

$c_i' = c_i + (2i - 2^{\lceil log(i) \rceil}) - (2(i-1) - 2^{\lceil log(i-1) \rceil}) = c_i + 2 - 2^{\lceil log(i) \rceil} + 2^{\lceil log(i-1) \rceil}$

Case $i - 1 = 2^k$: Then $\lceil log(i) \rceil = k + 1$ and we get: $c_i' = (2^k + 1) + 2 - 2^{k+1} + 2^k = 3$

Case $2^{k-1} < i - 1 < 2^k$**:** Then $\lceil log(i) \rceil = \lceil log(i-1) \rceil = k$ and we get: $c_i' = 1 + 2 - 2^k + 2^k = 3$

Amortized analysis of increments of a binary counter

If we need to store a (possible large) binary counter then it is natural to resort to an array and let the array element A[i] store the i-th bit of the counter. The standard way of incrementing the counter is to toggle the lowest-order bit. If that bit switches to a 0 then we toggle the next higher-order bit, and so forth unitl the bit that we toggle switches to a 1 at which point we can stop.

What is the complexity of n increment operations?

- If we have n operations on a k-bit counter then the overall complexity is at most $O(k \cdot n)$. Note that, possibly $k >> n$.
- We can improve this boundary if we take into account that the i-th increment is the number i. Hence, after n increments at most $O(log(n))$ bits can have been toggled per increment, yielding a total of $O(n \cdot log(n))$ bits that need to be toggled.

Amortized Analysis:

When does the i-th bit need to be toggled?

- A[0] is toggled very time
- A[1] is toggled every second time
- A[1] is toggled very fourth time

Therefore, we can conclude that incrementing the counter n times toggles the ...

- A[0] n times
- A[1] $n/2$ times
- A[2] $n/4$ times

- ...

Hence, the sum of toggled bits after n k-bit counter increments becomes:

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

$n + \sum_{i=1}^{k-1} \frac{1}{2^i} \leq n + n \cdot \sum_{i=1}^{\infty} \frac{1}{2^i} = n + \frac{1/2}{1-1/2} n = n + n = 2n$

According to amortized analysis: $\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} c'_i$

$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} c'_i \leq \sum_{i=0}^{\lfloor log(n) \rfloor} \frac{n}{2^i} \leq 2n$

Hence, we get 2 as the amortized cost of one increment.

Practical relevance of log-terms

Since $2^{20} = 1048576$ and $2^{25} = 33554432$, in most applications the value of $log(n)$ will hardly be significantly greater than 25 for practically relevant values. Hence, shaving off a log-term might constitute an important accomplishment when seen from a purely theoretical point of view, but its practical impact is likely to be much more questionable. In particular, multiplicative constants hidden in the O-terms may easily diminish the actual difference in speed between, say, an $O(n)$-algorithm and an $O(n \cdot log(n))$-algorithm.

IMPORTANT! Do not rely purely on experimental analysis to "detect" a log-factor: The difference between $log(1024) = log(2^{10})$ and $log(1048576) = log(2^{20})$ is just a multiplicative factor of two!

Impact of Compile-Time Optimization

Optimizing compilers try to minimize important characteristics of a program, such as its CPU-time consumption. Typically, heuristics are employed that transform a program to a (hopefully) sementically equivalent program since some problem related to code optimization are NP-complete or even undecidable. For instance, an optimizing compiler will attempt to keep frequently used variables in registers rather than in main memory.

IMPORTANT! In general, an optimized code will run faster. But optimization is not guaranteed to improve performance in all cases! It may even empede performance.

Dealing with Floating-Point Computations

The floating-point unit on x86 processors use 80bit registers and operators while standard "double" variables are stored in 64bit memory cells. Hence, rounding to a lower precision is necessary whenever a floating-point variable is transferred from register to memory. Optimizing compilers analyze code and keep variables within the registers whenever this makes sense without storing intermediate results in memory.

Hence, the result of floating-point computations may depend on the compile-time options.

$$\sum_{i=1}^{1\,000\,000} 0.001 = 1000.0000000000009095 \qquad \text{when } \texttt{gcc -O2} \text{ was used, and}$$

$$\sum_{i=1}^{1\,000\,000} 0.001 = 999.9999999832650701 \qquad \text{when } \texttt{gcc -O0} \text{ was used.}$$

Theory tells us that we can approximate the first derivative f' of a function f at the point x_0 by evaluating $\frac{f(x_0+h)-f(x_0)}{h}$ for sufficiently small values of h.

Consider $f(x) := x^3$ and $x_0 := 10$:

$h := 10^{0}$:	$f'(1) \approx 331.0000000$	$h := 10^{-1}$:	$f'(1) \approx 303.0099999$
$h := 10^{-2}$:	$f'(1) \approx 300.3000999$	$h := 10^{-3}$:	$f'(1) \approx 300.0300009$
$h := 10^{-4}$:	$f'(1) \approx 300.0030000$	$h := 10^{-5}$:	$f'(1) \approx 300.0002999$
$h := 10^{-6}$:	$f'(1) \approx 300.0000298$	$h := 10^{-7}$:	$f'(1) \approx 300.0000003$
$h := 10^{-8}$:	$f'(1) \approx 300.0000219$	$h := 10^{-9}$:	$f'(1) \approx 300.0000106$
$h := 10^{-10}$:	$f'(1) \approx 300.0002379$	$h := 10^{-11}$:	$f'(1) \approx 299.9854586$
$h := 10^{-12}$:	$f'(1) \approx 300.1332515$	$h := 10^{-13}$:	$f'(1) \approx 298.9963832$
$h := 10^{-14}$:	$f'(1) \approx 318.3231456$	$h := 10^{-15}$:	$f'(1) \approx 568.4341886$
$h := 10^{-16}$:	$f'(1) \approx 0.000000000$	$h := 10^{-17}$:	$f'(1) \approx 0.000000000$

This gap between the theory of the reals and floating-point practice has important and severe consequences for the actual coding practice when implementing (geometric) algorithms that require floating-point arithmetic:

- The correctness proof of the mathematical algorithm does not extend to the program, and the program can fail on seemingly appropriate input data.
- Local consistency need not imply global consistency

IMPORTANT: Numerical analysis and adequate coding are a must when implementing algorithms that deal with real numbers. Otherwise, the implementation of an algorithm may turn out to be absolutely useless in practice, even if the algorithm (and even its implementation) would come with a rigorous mathematical proof of correctness!

Impact of Cache Misses

Today's computers perform arithmetic and logical operations on data stored in registers. In addition to main memory, data can also be stored in a Level 1 cache or a Level 2 cache. (Multi-core machines tend to have also L3 caches).

Note that:

- A cache is a fast but expensive memory which holds the values of standard memory locations.
- If the CPU requests the value of a memory location and if that value is available in some level of cache, then the value is fetched from the cache, at a cost of a few cycles: **cache hit**
- Otherwise, a block of consecutive memory locations is accessed and brought into the cache: **cache miss**
- A cache miss is much costlier than a cache hit!

Since the gap between CPU speed and memory speed gets wider and wider, good cache management and programs that exhibit good *locality* become increasingly more important.

Impact on Matrix Multiplication

In C/C++ elements within the same row of a matrix are stored in consecutive memory locations while elements in the same column are far apart in main memory. The standard implementation of matrix multiplication causes the elements of **A** and **C** to be accessed row-wise, while the elements of **B** are accessed by column. This will obviously result in a lot of cache misses if **B** is too large to fit into the (L2) cache.

Hence, it makes sense to rewrite the standard multiplication algorithm ("ijk-order"). Re-ordering of the inner loops will cause the matrices **B** and **C** to be accessed row-wise within the inner-most loop, while the indices, i, k of the (i,k)-th element of **AA** remain constant: "ikj-order"

IMPORTANT: Algorithm engineering should be standard when designing and implementing an algorithm! Decent algorithm engineering may pay off more significantly than attempting to implement a highly complicated algorithm just because its theoretical analysis predicts a better running time.