Algorithmic Complexity of Computational Problems

Ștefan Ciobâcă, Dorel Lucanu

Faculty of Computer Science
Alexandru Ioan Cuza University, Iași, Romania
dlucanu@info.uaic.ro

PA 2019/2020

1. Recap
2. (Algorithmic) Problems Complexity
3. The Complexity of Sorting
4. The complexity of the divide-et-impera search
5. Polynomial reduction of problems

Plan

1. Recap
2. (Algorithmic) Problems Complexity
3. The Complexity of Sorting
4. The complexity of the divide-et-impera search
5. Polynomial reduction of problems

From the last lecture

- problem solved by an algorithm
  - problem P
    instance (= input): $p \in P$
    result (= output): $P(p)$
  - A solves P:
    σ_p = state encoding $p \in P$
    $\langle A, \sigma_p \rangle \Rightarrow^* \langle \cdot, \sigma' \rangle$, where σ' encodes $P(p)$
- decision problem: (instance, question)
- decidable problem: decision problem solved by a algorithm

From the last lecture

- sizw of an instance: if $p \in P$, then
  $size_d(p) = size_d(\sigma_p) = \sum_{var \mapsto val \in \sigma} size(val)$, $d \in \{log, unif\}$
- space/time complexity:
  $\langle A, \sigma_p \rangle = \langle A_0, \sigma_0 \rangle \Rightarrow \langle A_1, \sigma_1 \rangle \Rightarrow \ldots \Rightarrow \langle A_n, \sigma_n \rangle = \langle \cdot, \sigma' \rangle$
  $T_d(A, p) = \sum_i time_d(\langle A_i, \sigma_i \rangle \Rightarrow \langle A_{i+1}, \sigma_{i+1} \rangle)$
  $S_d(A, p) = \max_i size_d(\langle A_i, \sigma_i \rangle) = \max_i size_d(\sigma_i)$
- worst-case complexity
  $n = \{p \in P \mid size_d(p) = n\}$
  $T_d(A, n) = \max\{T_d(A, p) \mid size_d(p) = n\}$
  $S_d(A, n) = \max\{S_d(A, p) \mid size_d(p) = n\}$

On the size of an instance

Consider the algorithm:

```
//@input: m >= 0
//@output: s == m
s = 0;
for (i = 1; i <= m; ++i)
  s = s + 1;
```

- uniform ($n = O(1)$): execution time constant!!??
- logarithmic ($n = O(\log m)$): time = $\sum_{i=1}^{m} \log m = m \log m = O(2^n)$!!?? (only the comparison were counted)
- linear $n = m$ (the most natural;)

On the size of an instance

- primality test
  ```
  isPrime(n) {
    ...
  }
  ```
  Usual the linear version is used (n)
  Agrawal et al., 2004:
  $T(n) = O(\log^{15/2} n \cdot \mathrm{poly}(\log\log n)) = O(\log^{15/2+\varepsilon} n)$.
- cel mai mare divizor comun
  ```
  gcd(a, b) {
    ...
  }
  ```
  ithe input includes two variables a and b
  $n = a \cdot b$
  the number of multiplications: $O(5 \log_{10} \min(a, b))$
  (k multiplications $\implies \max(a, b) \geq fib(k+2)$,
  $\min(a, b) \geq fib(k+1)$)

Plan

1. Recap
2. (Algorithmic) Problems Complexity
3. The Complexity of Sorting
4. The complexity of the divide-et-impera search
5. Polynomial reduction of problems

Why?

A problem may be solved by many algorithms.

Actually, if there is one algorithm soving it, then there are an infinity. (Why?)

How efficiently a problem can be solved (if any)?

The definition from the efficiency of the algorithms can be extended to problems.

When we know ONE algorithm solving the problem

Consider:

- a problem P
- $n = size(x)$, $x \in P$ an instance of P
- an algorithm A solving P with the worst-time execution time $O(f(n))$

What can we say about the time complexity of P?

The complexity $O(f(n))$ of a problem

It supplies a superior bound for the computational effort needed to solve a problem.

Definition

A problem P has the worst case time complexity $O(f(n))$ if there is an algorithm A that solves P and $T_A(n) = O(f(n))$.

When we want to know something about ALL the algorithms

Consider:

- a problem P
- $n = size(x)$, $x \in P$ an instance of P

What kind of information can we supply about all the algorithms solving P

The complexity $\Omega(f(n))$ of a problem

It supplies a inferior bound for the computational effort needed to solve a problem.

Definition

A problem P has the worst case time complexity $\Omega(f(n))$ if any algorithm A that solves P has $T_A(n) = \Omega(f(n))$.

An optimal algorithm for a problem

Consider:

- a problem P
- $n = size(x)$, $x \in P$ an instance of P

When an algorithm is optimal for P?

An optimal algorithm for a problem

Consider:

- a problem P
- $n = size(x)$, $x \in P$ an instance of P

When an algorithm is optimal for P?

Definition

A is an optimal algorithm (w.r.t. the worst case time complexity) for P if

- A solves P and
- P has the worst case time complexity $\Omega(T_A(n))$.

Plan

1. Recap
2. (Algorithmic) Problems Complexity
3. The Complexity of Sorting
4. The complexity of the divide-et-impera search
5. Polynomial reduction of problems

Sorting Problem

Consider the particular case of arrays sorting:

SORT
Input n and the array $a = [v_0, \ldots, v_{n-1}]$.
Output an array $a' = [w_0, \ldots, w_{n-1}]$ with the property: $w_0 \leq \cdots \leq w_{n-1}$ and $w = (w_0, \ldots, w_{n-1})$ is a permutation of $v = (v_0, \ldots, v_{n-1})$; .

Notation:
SORTED(a): the array a is sorted (non-decreasingly ordered)
$\mathrm{Perm}(v, w)$: w is a permutation of v

BubbleSort 1/2

A possible definition for *SORTED*(a)

$$SORTED(a) \iff (\forall i)(0 \leq i < n-1) \Rightarrow a[i] \leq a[i+1]$$

where $n = a.size()$. (This is a part of the problem domain.)

From the problem domain to the algorithm:

```
for (i=0; i < n-1; ++i) {
  if (a[i] > a[i+1]) {
    swap (a, i, i+1);
```

`(a[i] > a[i+1])` is called inversion

BubbleSort 2/2

The process from the previous slide must be repeated until no more inversions exist:

```
while (there are possible more inversions) {
  for (i=0; i < n-1; ++i) {
    if (a[i] > a[i+1]) {
      swap (a, i, i+1);
}
```

(This is pseudocode!)

The test *there are possible more inversions* can be checked storing the position of the last inversion: next slide

BubbleSort: the algorithm

```
bubbleSort(a, n) {
  ultim = n-1;
  while (ultim > 0) {
    n1 = ultim;
    ultim = 0;
    for (i=0; i < n1; ++i) {
      if (a[i] > a[i+1]) {
        swap (a, i, i+1);
        ultim = i;
      }
    }
  }
}
```

```
swap(a, i, j) {
  temp = a[i];
  a[i] = a[j];
  a[j] = temp;
}
```

Analysis of the algorithm BubbleSort 1/2

Correctness

`while` loop invariant: $\mathtt{a[ultim+1\ ..\ n-1]}$ include the biggest $\mathtt{n-1-ultim}$ elements in a and $SORTED(\mathtt{a[ultim+1\ ..\ n-1]})$

`for` loop invariant: $\mathtt{a[j]} \leq \mathtt{a[i]}$ for $j = 0, \ldots, i$.

`swap` maintains the property $\mathrm{Perm}(u, u')$, where u is the value of the variable a before swap `swap` and u' the after one

Analysis of the algorithm BubbleSort 1/2

Execution time

- instance size: n (= `a.size()`)
- measured operations: comparisons involving the array elements
- the worst case:

Analysis of the algorithm BubbleSort 1/2

Execution time

- instance size: n $(=$ `a.size()`$)$
- measured operations: comparisons involving the array elements
- the worst case: when the elements of the array are in decreasing order
- the number of the comparisons for this case is

$$(n-1)+(n-2)+\cdots+1=\frac{(n-1)n}{2}=O(n^2)$$

InsertSort 1/2

Basic principle

for j in 1 .. n-1
  insert a[j] in a[0..j-1] s.t. SORT(a[0..j])

(This is pseudo-code!)

InsertSort 2/2

Problem domain analysis

The position i where $a[j]$ to be inserted:

- $i = j$ if $a[j] \geq a[j-1]$;
- $i = 0$ if $a[j] < a[0]$;
- $0 < i < j$ and $a[i-1] \leq a[j] < a[i]$

$\implies$ $a[i..j-1]$ must be moved to right one position!

– the condition for moving to right: $i \geq 0 \wedge a[i] > a[j]$

Algorithmically:

```
i = j - 1;
temp = a[j];
while ((i >= 0) && (a[i] > temp)) {
  a[i+1] = a[i];
  i = i -1;
}
```

InsertSort: the algorithm

```
insertSort(a, n) {
  for (j = 1; j < n;  j = j+1) {
    i = j - 1;
    temp = a[j];
    while ((i >= 0) && (temp < a[i])) {
      a[i+1] = a[i];
      i = i -1;
    }
    if (i != j-1) a[i+1] = temp;
  }
}
```

The analysis of the algorithm InsertSort 1/2

Correctness

`for` loop invariant: $\mathrm{Perm}(u, v) \wedge \mathit{SORTED}(a[0..j-1])$, where u is the current value of a

`while` loop invariant: $a[i{+}1], \ldots, a[j{-}1] > \mathit{temp}$.

The `while` loop invariant and $\texttt{a[i]} \leq \texttt{temp} \vee \texttt{i} < 0$ ensures the correct computation of i, i.e. $\mathit{SORTED}(a[0..j])$.

The analysis of the algorithm InsertSort 2/2

Execution time

- instance size: n ($=$ `a.size()`)
- measured operations: comparisons involving the array elements
- the worst case:

The analysis of the algorithm InsertSort 2/2

Execution time

- instance size: n (= `a.size()`)
- measured operations: comparisons involving the array elements
- the worst case: when the input sequence is decreasing
  – searching i in a$[0 \,..\, j-1]$ requires $j-1$ comaparisons
- the number of comparisons for this case is
  $$1 + 2 = \cdots + (n-1) = \frac{(n-1)n}{2} = O(n^2)$$

HeapSort

Problem domain analysis The property $MAXHEAP(\mathtt{a})$:

$$\begin{aligned}(\forall i \geq 0) 2i+1 < n &\implies a[i] \geq a[2i+1) \wedge \\ 2(i+1) < n &\implies a[i] \geq a[2(i+1))\end{aligned}$$

$MAXHEAP(\mathtt{a}) \implies \max \mathtt{a} = \mathtt{a}[0]$

The main idea of the algorithm:

- assume $MAXHEAP(\mathtt{a})$
- if we do the interchange `swap(a,0,n-1)`, the new value $a[n-1]$ is on its final position and the remained array to be sorted is $\mathtt{a}[0..n-2]$
- $\mathtt{a}[0..n-2]$ is sorted in the same manner

More algorithmically

```
heapSort(a, n) {
  establish MAXHEAP( a)
  for (r = n-1; r > 0; --r) {
    swap(a, 0, r);
    re-establish MAXHEAP( a[0..r − 1])
}
```

(This is pseudocode!)

Establishing the max-heap property

Problem domain

- $MAXHEAP(\mathrm{a}, \ell)$:

$$(\forall i \geq \ell) 2i + 1 < n \implies a[i] \geq a[2i+1) \wedge$$
$$2(i+1) < n \implies a[i] \geq a[2(i+1))$$

- $\ell \geq n/2 \implies MAXHEAP(\mathrm{a}, \ell)$
- if $MAXHEAP(\mathrm{a}, \ell - 1)$ the we obtain $MAXHEAP(\mathrm{a}, \ell)$ inserting $a[\ell - 1]$ in $\mathrm{a}[\ell..n-1]$

Algorithmically:

```
j = ℓ;
while (exists children of j) {
  k = the index of the child with the maximum value;
  if (a[j] < a[k]) swap(a, j, k);
  j = k;
}
```

HeapSort: the algorithm

```
insertInHeap(a, n, ℓ) {
  isHeap = false; j = ℓ;
  while (2*j+1 <= n-1 && ! isHeap) {
    k = 2*j +1;
    if ((k < n-1) && (a[k] < a[k+1]))  k = k+1;
    if (a[j] < a[k]) swap(a, j, k); else  isHeap = true;
    j = k;
  }
}
heapSort(a, n) {
  for (l = (n-1)/2; l >= 0; l = l-1)
    insertInHeap(a, n, l);
  r = n-1;
  while (r >= 1) {
    swap(a, 0, r);
    insertInHeap(a, r, 0);
    r = r - 1;
  }
}
```

HeapSort: analysis 1/2

Correctness It is based on the correctness of the operations of the data-structure *max-heap*.

the `while` invariant in `insertInHeap`: $(\forall i \geq \ell)$ if j is not in the tree with the root in i, then $MAXHEAP(\mathtt{a}, i)$

the `for` invarian in `heapSort`: $MAXHEAP(\mathtt{a}, \ell)$

the `while` invariant in `heapSort`: $MAXHEAP(\mathtt{a}[0..r-1]) \wedge SORTED(\mathtt{a}[r..n-1])$

HeapSort: analysis 2/2

Execution time

- instance size: n ($=$ `a.size()`)
- operations measured: comparisons involving the array elements
- the worst case:

HeapSort: analysis 2/2

Execution time

- instance size: n (= `a.size()`)
- operations measured: comparisons involving the array elements
- the worst case: hard to say
  - time complexity for `insertInHeap`: $O(\log n)$
  - but the construction of the whole heap requires $O(n \log n) = O(\log \frac{n-1}{2}) + \cdots + O(\log n)$ (in fact $\Theta(n)$, see Cormen et al., 6.3)
  - time complexity for `while`: $O(log(n-1) + O(\log n - 2) + \cdots + O(\log 1) = O(n \log n)$
- the total number of comparisons $O(n \log n)$

Other sorting algorithms

Exercises for seminar.

Two questions regarding the sorting algorithms

- what is the minimal number of comparisons in the worst case?
- which sorting algorithms requires the minimal number of comparisons?

To answer these questions we have to formally define the computational model of comparisons-based algorithms.

Decision trees for sorting: intuitive

Assumption: $a_i \neq a_j$ if $i \neq j$

Notation: $i\,?\,j \equiv$ a[i] and a[j] are compared

A decision tree for sorting includes the comparisons made by the algorithm:

- an internal mode is labelled with $i\,?\,j$;
- the left subtree of $i\,?\,j$ includes the comaprisons for $a_i < a_j$;
- the right subtree of $i\,?\,j$ includes the comaprisons for $a_i > a_j$;
- the external (frontier) nodes are labelled with permutations

The algorithms represented as decision trees

Definition

Decision tree for n elements:

- internal nodes: $i\,?\,j$
- external (frontier) nodes: permutations of the set $\{0, 1, \ldots, n-1\}$

Definition

A computation of a decision tree t for the input $a = (a_0, \ldots, a_{n-1})$:
a path from the root to the fontier with the property

- if $a_i < a_j$: the left child of $i\,?\,j$ the current node;
- otherwise the right child becomes the current node
- the computation (should) terminate on the frontier

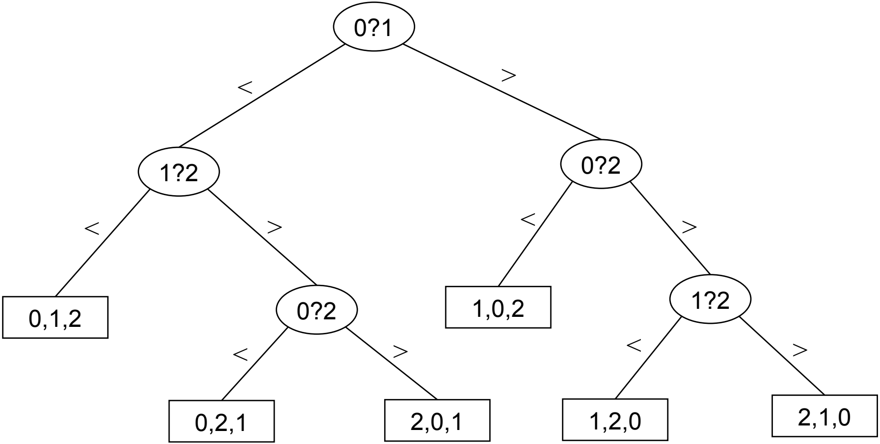
Decision trees for sorting

Definition

t solves the sorting problem:
$\forall$ input $a = (a_0, \ldots, a_{n-1})$
the computation of t for a terminates in π s.t. $a_{\pi(0)} < \cdots < a_{\pi(n-1)}$

decision tree for sorting: decision tree that solves the sorting problem

A decision tree representing InsertSort



Time complexity of sorting

Notations:
$ADS(n)$ = the set of decision trees for sorting sequnces of length n
$Fr(t)$ = the frontier of the decision tree t
$length(\pi, t)$ = the length in t of the path from the root to $\pi \in Fr(t)$ The time complexity for the worst case:

$$T(n) = \min_{t \in ADS(n)} \max_{\pi \in Fr(t)} \mathrm{length}(\pi, t)$$

Theorem

The sorting problem has the worst case time complexity $\Omega(n \log n)$ in the computational model of the decision trees for sorting.

Corollary

HeapSort is optimal in the computational model of the decision trees for sorting.

Plan

1 Recap

2 (Algorithmic) Problems Complexity

3 The Complexity of Sorting

4 The complexity of the divide-et-impera search

5 Polynomial reduction of problems

Searching problem

Instance	a universe set $\mathcal{U}$, a subset $S \subseteq \mathcal{U}$ and an element a in $\mathcal{U}$;
Question	$a \in S$?

Assume that $\mathcal{U}$ is totally ordered and S is represented by an array $\mathtt{s[0..n{-}1]}$ with $s[0] < \cdots < s[n-1]$.

A generic divide-et-impera algorithm for searching: the idea

Generalize: s[p..q]].
The algorithm:

- choose m with $p \leq m \leq q$;
- if $a = s[m]$ then the searching successfully terminates;
- if $a < s[m]$ then the searching continues for $(s[p], \ldots, s[m-1])$;
- if $a > s[m]$ then the search continues for $(s[m+1], \ldots, s[q])$;

The most known algorithms:

- Linear searching: $m = p$.
- Binary search: $m = \lceil \frac{p+q}{2} \rceil$.
- Fibonacci search: $q + 1 - p = Fib(k) - 1$

A generic divide-et-impera algorithm for searching

```
pos(s, n, a) {
  p = 0; q = n - 1;
  2: choose m between p and  q
  while ( (a != s[m]) && (p < q)) {
    if (a < s[m]) q = m -1; else  p = m + 1;
    5: choose m between p and  q
  }
  if (a == s[m]) return m; else return -1;
}
```

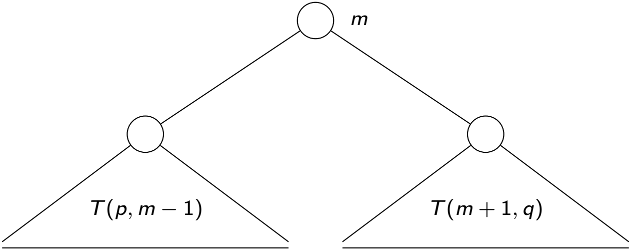
Searching algorithms represented as decision trees 1/2

Definition

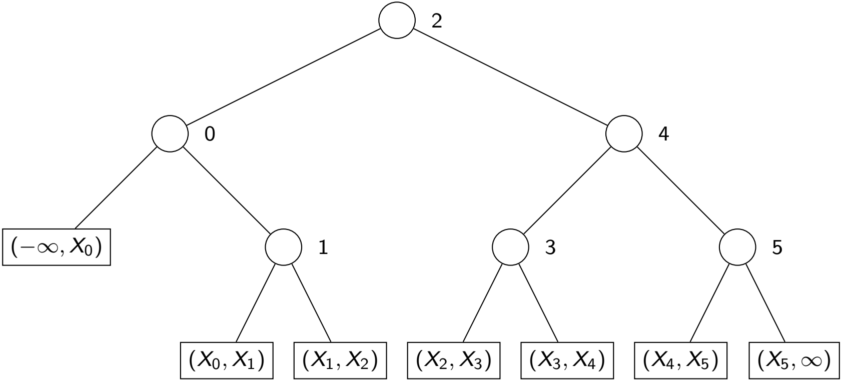
The decision tree for searching of dimension n: $T(0, n-1)$, where $T(p, q)$ is defined as follows:

- if $p > q$ then $T(p, q)$ is the empty tree;
- otherwise the root is m given by the instr. 2 or 5, the left subtree is $T(p, m-1)$ and the right one is $T(m+1, q)$
- the frontier nodes: $(-\infty, X_0), (X_0, X_1), \ldots, (X_{n-1}, +\infty)$ in this order from left to right

$T(p, q)$ graphically



Example of decision tree for searching



Searching algorithms represented as decision trees 2/2

Definition

The computation of a decision tree for the input $x_0, \ldots, x_{n-1}, a$:
a path from the root towards frontier with the property

1. if the current node is (X_i, X_{i+1}) (on the frontier), then $a \in (x_i, x_{i+1})$ and the computation unsuccessfully terminates;
2. if the current node is m and $a = x_m$, then the computation successfully terminates;
3. if the current node is m and $a < x_m$ then the root of the left subtree becomes the current node;
4. if the current node is m and $a > x_m$ then the root of the right subtree becomes the current node.

The particular case of the bunary search

Lemma

Let t be a decision tree for binary searching. If $2^{h-1} \leq n < 2^h$, then the height of t is h.

Corollary

The execution time for the binary search is $O(\log_2 n)$.

Proprieties of the decision trees for searching

Definition

The internal length of t: $\mathrm{IntLength}(t) =$ the sum of the lengths of the paths from the root to the internal nodes
The external length of t: $\mathrm{ExtLength}(t) =$ the sum of the lengths of the paths from the root to the frontier

Lemma

Let t be a decision tree for searching with n internal nodes. Then:

$$\mathrm{ExtLength}(t) - \mathrm{IntLength}(t) = 2n.$$

Lemma

The minimal length of a decision tree for searching with n internal nodes is

$$(n+1)(h-1) - 2^h + 2$$

The complexity of searching

Theorem

The searching problem has the worst case time complexity $\Omega(\log n)$ in the model of the decision trees for searching.

Corollary

The binary search is optimal in the model of the decision trees for searching.

Plan

1 Recap

2 (Algorithmic) Problems Complexity

3 The Complexity of Sorting

4 The complexity of the divide-et-impera search

5 Polynomial reduction of problems

Motivation

Mentality: "If I know to solve a problem Q, then I can use this algorithm to solve P?"

Motivation

Mentality: "If I know to solve a problem Q, then I can use this algorithm to solve P?"

Intuition: A problem P reduces to Q if the algorithms for Q can help to solve P.

Motivation

Mentality: "If I know to solve a problem Q, then I can use this algorithm to solve P?"

Intuition: A problem P reduces to Q if the algorithms for Q can help to solve P.

Aplication:

- algorithm design
- proof of the limits: if P is difficult then Q is difficult as well
- problem classification

Turing/Cook Reduction

A problem P polynomially reduces to (a solvable problem) Q, write $P \propto Q$, if we can design an algorithm for P as follows:

Turing/Cook Reduction

A problem P polynomially reduces to (a solvable problem) Q, write $P \propto Q$, if we can design an algorithm for P as follows:

1. let p be an instance of P;

Turing/Cook Reduction

A problem P polynomially reduces to (a solvable problem) Q, write $P \propto Q$, if we can design an algorithm for P as follows:

1. let p be an instance of P;
2. preprocesss in polynomial time the input p to obtain an instance of Q

Turing/Cook Reduction

A problem P polynomially reduces to (a solvable problem) Q, write $P \propto Q$, if we can design an algorithm for P as follows:

1. let p be an instance of P;
2. preprocesss in polynomial time the input p to obtain an instance of Q
3. call an algorithm for Q, possible of several times (but of polynomial times)

Turing/Cook Reduction

A problem P polynomially reduces to (a solvable problem) Q, write $P \propto Q$, if we can design an algorithm for P as follows:

1. let p be an instance of P;
2. preprocesss in polynomial time the input p to obtain an instance of Q
3. call an algorithm for Q, possible of several times (but of polynomial times)
4. postprocess the output(s) given by Q in polynomial time to get the answer $P(p)$

Turing/Cook Reduction

A problem P polynomially reduces to (a solvable problem) Q, write $P \propto Q$, if we can design an algorithm for P as follows:

1. let p be an instance of P;
2. preprocesss in polynomial time the input p to obtain an instance of Q
3. call an algorithm for Q, possible of several times (but of polynomial times)
4. postprocess the output(s) given by Q in polynomial time to get the answer $P(p)$

If the (pre+post)processing time requires $O(g(n))$ time, then we write $P \propto_{g(n)} Q$.

Example: MAX $\propto$ SORT

FieLet MAX be the following problem:

Input	A set S totally ordered.
Output	The bigests element in S.

The following algorithm solves MAX:

1. represent S with an array s (preprocessing);
2. call a sorting algorithm for s;
3. return the last element in s (postprocessing);

$\propto$ does not necessarily mean "the reduction from a complex problem to an easier one"!!!

$\propto$ is rather a "transformation" ...

Variants for the subset sum problem

SSD1

Input	A set S of integers, M a positive integer.
Output	The bigest integer M^* s.t. $M^* \leq M$ and $\exists\ S' \subseteq S$ with $\sum_{x \in S'} x = M^*$.

SSD2

Instance	A set S of integers, M, K two positive integers with $K \leq M$.
Question	Does it exists M° s.t. $K \leq M^\circ \leq M$ and $\sum_{x \in S'} x = M^\circ$ for a certain set $S' \subseteq S$?

SSD3

Instance	A set S of integers, M a positive integer.
Question	Does it exists a subset $S' \subseteq S$ with $\sum_{x \in S'} x = M$?

Example: SSD1 $\propto$ SSD2

SSD1

Input A set S of integers, M a positive integer.

Output The bigest integer M^* s.t. $M^* \leq M$ and $\exists\ S' \subseteq S$ with $\sum_{x \in S'} x = M^*$.

SSD2

Instance A set S of integers, M, K two positive integers with $K \leq M$.

Question Does it exists M° s.t. $K \leq M^\circ \leq M$ and $\sum_{x \in S'} x = M^\circ$ for a certain set $S' \subseteq S$?

Example: SSD1 $\propto$ SSD2

SSD1

Input A set S of integers, M a positive integer.

Output The bigest integer M^* s.t. $M^* \leq M$ and $\exists\, S' \subseteq S$ with $\sum_{x \in S'} x = M^*$.

SSD2

Instance A set S of integers, M, K two positive integers with $K \leq M$.

Question Does it exists M° s.t. $K \leq M^\circ \leq M$ and $\sum_{x \in S'} x = M^\circ$ for a certain set $S' \subseteq S$?

1. no preprocessing;

Example: SSD1 $\propto$ SSD2

SSD1

Input A set S of integers, M a positive integer.

Output The bigest integer M^* s.t. $M^* \leq M$ and $\exists\ S' \subseteq S$ with $\sum_{x \in S'} x = M^*$.

SSD2

Instance A set S of integers, M, K two positive integers with $K \leq M$.

Question Does it exists M° s.t. $K \leq M^\circ \leq M$ and $\sum_{x \in S'} x = M^\circ$ for a certain set $S' \subseteq S$?

1. no preprocessing;
2. find M^* in $(0, M]$ calling an algorithm that solves SSD2 in a binary search manner;

Example: SSD1 $\propto$ SSD2

SSD1

Input	A set S of integers, M a positive integer.
Output	The bigest integer M^* s.t. $M^* \leq M$ and $\exists\, S' \subseteq S$ with $\sum_{x \in S'} x = M^*$.

SSD2

Instance	A set S of integers, M, K two positive integers with $K \leq M$.
Question	Does it exists M° s.t. $K \leq M^\circ \leq M$ and $\sum_{x \in S'} x = M^\circ$ for a certain set $S' \subseteq S$?

1. no preprocessing;
2. find M^* in $(0, M]$ calling an algorithm that solves SSD2 in a binary search manner;

This is an example when an optimisation problem is reduced to decision problem.

Example: SSD2 $\propto$ SSD1

SSD1

Input	A set S of integers, M a positive integer.
Output	The bigest integer M^* s.t. $M^* \leq M$ and $\exists\ S' \subseteq S$ with $\sum_{x \in S'} x = M^*$.

SSD2

Instance	A set S of integers, M, K two positive integers with $K \leq M$.
Question	Does it exists M° s.t. $K \leq M^\circ \leq M$ and $\sum_{x \in S'} x = M^\circ$ for a certain set $S' \subseteq S$?

Example: SSD2 $\propto$ SSD1

SSD1

Input A set S of integers, M a positive integer.

Output The bigest integer M^* s.t. $M^* \leq M$ and $\exists\ S' \subseteq S$ with $\sum_{x \in S'} x = M^*$.

SSD2

Instance A set S of integers, M, K two positive integers with $K \leq M$.

Question Does it exists M° s.t. $K \leq M^\circ \leq M$ and $\sum_{x \in S'} x = M^\circ$ for a certain set $S' \subseteq S$?

1. no preprocessing;

Example: SSD2 $\propto$ SSD1

SSD1

Input A set S of integers, M a positive integer.

Output The bigest integer M^* s.t. $M^* \leq M$ and $\exists\ S' \subseteq S$ with $\sum_{x \in S'} x = M^*$.

SSD2

Instance A set S of integers, M, K two positive integers with $K \leq M$.

Question Does it exists M° s.t. $K \leq M^\circ \leq M$ and $\sum_{x \in S'} x = M^\circ$ for a certain set $S' \subseteq S$?

1. no preprocessing;
2. compute $M^* \leq M$ calling an algorithm solving SSD1;

Example: SSD2 $\propto$ SSD1

SSD1

Input	A set S of integers, M a positive integer.
Output	The bigest integer M^* s.t. $M^* \leq M$ and $\exists\ S' \subseteq S$ with $\sum_{x \in S'} x = M^*$.

SSD2

Instance	A set S of integers, M, K two positive integers with $K \leq M$.
Question	Does it exists M° s.t. $K \leq M^\circ \leq M$ and $\sum_{x \in S'} x = M^\circ$ for a certain set $S' \subseteq S$?

1. no preprocessing;
2. compute $M^* \leq M$ calling an algorithm solving SSD1;
3. if $M^* \geq K$ then return 'YES', otherwise return 'NO';

Example: SSD3 $\propto$ SSD1

SSD1

Input	A set S of integers, M a positive integer.
Output	The bigest integer M^* s.t. $M^* \leq M$ and $\exists\ S' \subseteq S$ with $\sum_{x \in S'} x = M^*$.

SSD3

Instance	A set S of integers, M a positive integer.
Question	Does it exists a subset $S' \subseteq S$ with $\sum_{x \in S'} x = M$?

Example: SSD3 $\propto$ SSD1

SSD1

Input A set S of integers, M a positive integer.

Output The bigest integer M^* s.t. $M^* \leq M$ and $\exists\ S' \subseteq S$ with $\sum_{x \in S'} x = M^*$.

SSD3

Instance A set S of integers, M a positive integer.

Question Does it exists a subset $S' \subseteq S$ with $\sum_{x \in S'} x = M$?

1. no preprocessing;

Example: SSD3 $\propto$ SSD1

SSD1

Input A set S of integers, M a positive integer.

Output The bigest integer M^* s.t. $M^* \leq M$ and $\exists\ S' \subseteq S$ with $\sum_{x \in S'} x = M^*$.

SSD3

Instance A set S of integers, M a positive integer.

Question Does it exists a subset $S' \subseteq S$ with $\sum_{x \in S'} x = M$?

1. no preprocessing;
2. compute $M^* \leq M$ calling an algorithm solving SSD1;

Example: SSD3 $\propto$ SSD1

SSD1

Input	A set S of integers, M a positive integer.
Output	The bigest integer M^* s.t. $M^* \leq M$ and $\exists\ S' \subseteq S$ with $\sum_{x \in S'} x = M^*$.

SSD3

Instance	A set S of integers, M a positive integer.
Question	Does it exists a subset $S' \subseteq S$ with $\sum_{x \in S'} x = M$?

1. no preprocessing;
2. compute $M^* \leq M$ calling an algorithm solving SSD1;
3. if $M^* = M$ return 'YES', otherwise return 'NO';

Karp reduction

Let P and Q be decision problems.
The problem P polynomially reduces to (the solvable) Q, write $P \propto Q$, if we can design an algorithm that solves P as follows:

Karp reduction

Let P and Q be decision problems.
The problem P polynomially reduces to (the solvable) Q, write $P \propto Q$, if we can design an algorithm that solves P as follows:

1. let p be an instance of P;

Karp reduction

Let P and Q be decision problems.
The problem P polynomially reduces to (the solvable) Q, write $P \propto Q$, if we can design an algorithm that solves P as follows:

1. let p be an instance of P;
2. preprocess the input p in polynomial time

Karp reduction

Let P and Q be decision problems.
The problem P polynomially reduces to (the solvable) Q, write $P \propto Q$, if we can design an algorithm that solves P as follows:

1. let p be an instance of P;
2. preprocess the input p in polynomial time
3. call (once) an algorithm solving Q

Karp reduction

Let P and Q be decision problems.
The problem P polynomially reduces to (the solvable) Q, write $P \propto Q$, if we can design an algorithm that solves P as follows:

1. let p be an instance of P;
2. preprocess the input p in polynomial time
3. call (once) an algorithm solving Q
4. the answer for Q is the same with the answew of P for p (no post processing)

Karp reduction

Let P and Q be decision problems.
The problem P polynomially reduces to (the solvable) Q, write $P \propto Q$, if we can design an algorithm that solves P as follows:

1. let p be an instance of P;
2. preprocess the input p in polynomial time
3. call (once) an algorithm solving Q
4. the answer for Q is the same with the answew of P for p (no post processing)

If the preprocessing time is $O(g(n))$, then we write $P \propto_{g(n)} Q$.

The Karp reduction is a particular case of the Turing/Cook reduction.

Examplu: SSD3 $\propto$ SSD2

SSD2

Instance A set S of integers, M, K two positive integers with $K \leq M$.

Question Does it exists M° s.t. $K \leq M^\circ \leq M$ and $\sum_{x \in S'} x = M^\circ$ for a certain set $S' \subseteq S$?

SSD3

Instance A set S of integers, M a positive integer.

Question Does it exists a subset $S' \subseteq S$ with $\sum_{x \in S'} x = M$?

1. no preprocessing;
2. call an algorithm solving SSD2 for the instance S, M, M;

Example: 3-SUM $\propto$ 3-COLLINEAR

3-SUM
Instance A set S of n integers.
Question Exist there 3 numbers in S s.t. their sum is 0?

3-COLLINEAR
Instance A set S of n points in plane.
Question Exist there three points in S that are colinear?

3-SUM $\propto$ 3-COLLINEAR:

1. sconsider a input $S = \{a_0, a_1, \ldots, a_{n-1}\}$ of 3-SUM;
2. compute $t(S) = \{(a_0, a_0^3), (a_1, a_1^3), \ldots, (a_{n-1}, a_{n-1}^3)\}$
3. return the result given by an algorithm solving 3-COLLINEAR for $t(S)$.

Lemma

If a, b, c are distinct, then $a + b + c = 0$ iff (a, a^3), (b, b^3) and (c, c^3) are colinear.

Reduction: properties

Theorem

a) If P has the time complexity $\Omega(f(n))$ and $P \propto_{g(n)} Q$ (Karp version) the Q has the time complexity $\Omega(f(n) - g(n))$.

b) If Q has the time complexity $O(f(n))$ and $P \propto_{g(n)} Q$ (Karp version) then P has the time complexity $O(f(n) + g(n))$.