APROXIMATION AND RANDOMIZED ALGORITHMS

- Lectures Borut Robič , Exercises Uroš Čibej
  - both Laboratory for algorithmics, LA
- Lectures
  - Transparencies in English
- Literature
  - Transparencies (e-Ucilnica)
  - Aproksimacijski algoritmi (B.Robič) (in Slovene)
  - …

- Projects
  - See next transparency
- Exams
  - Only written exams, with questions related to lectures and exercises

Projects

- Each of you will make two projects: one on approximation algorithms and the other on randomized algorithms.
- A project may present
  - a problem + an approximation/randomized algorithm for solving the problem + results of analysis
  - a theoretical topic concerning approximation/randomized algorithms or corresponding classes
- Sources:
  - books, research papers, internet (wikipedia, …)
- A project must
  - be written in English
  - with LaTeX (preferably, not necessarily)
  - start with your name, title of the project, bibliographic data of the source (with link if existing)
  - extract important info (skip “unimportant” details) so that your colleagues can learn smthg from it
- Presentation of the projects
  - at the end of the semester (last few Wednesdays)
  - send me pdfs of your projects (to get marks)
  - upload pdfs to e-Ucilnica (to be read by your colleagues)
  - prepare yourself for oral presentation (10-15min)

The whole class: establish and maintain (e-Ucilnica) a list of projects you have chosen (so that your projects will differ as much as possible).

Contents

Introduction

Computational Problems

Incomputable Problems

P incomputable

- $\Leftrightarrow \neg\exists$alg A: A solves $\forall$instance p$\in$P
- i.e., $\forall$alg A fails on some p$\in$P

Issue:

- $\neg\exists$alg A*: A* decides whether or not A solves p$\in$P

Computable Problems

P computable

- $\Leftrightarrow\exists$alg A: A solves $\forall$instance p$\in$P
-

Issue:

- running time of A may be too large to be acceptable

Computable Problems

Intractable Problems

Running time of solving p∈P is

- superpolynomial (in size |p|=n of p)
- exponential (in size |p|=n of p)
- …
- i.e., too large to wait for the solution
- e.g., problems in the classes NPH, NPC, NPI, EXPTIME, PSPACE

Tractable Problems

Running time of solving p∈P is

- polynomial (in size |p|=n of p)
- bearable (for small sizes |p|=n of p)

- e.g. problems in **P** (matrix multiplication, **…**)

Intractable Problems

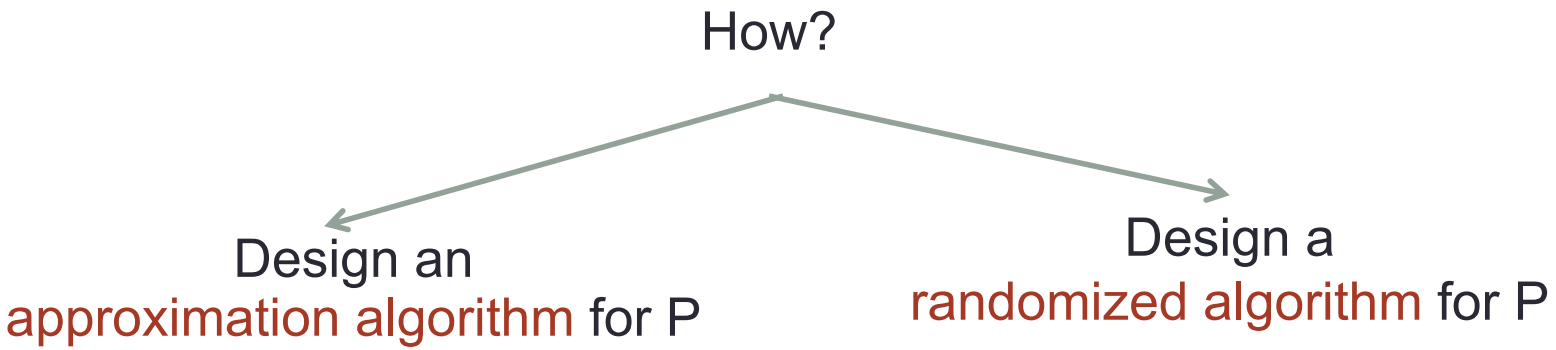
What can we do when we face an intractable problem P?

- Design a (slightly) faster exact algorithm A for P
  - Makes sense if we want to solve a slightly larger $p \in P$
- Design a parallel exact algorithm A for P
  - Makes sense if
    - ∃ parallel computer M with m processors and
    - P is amenable to parallelization
  - Issue: speedup can be at most m (constant)
- Design a quantum algorithm A for P
  - Issue: ∃ of general purpose quantum computer
- Design a heuristic algorithm A for P
  - Such an alg. A applies intuitive, *ad hoc* ideas (which can be good, bad, wrong, irrelevant) and returns suboptimal solutions in reasonable time (possibly polynomial)
  - Issue: intuition can be misleading ⇒ quality of suboptimal solutions may be low

(cont.)

(cont'd)

Idea: **Design a fast heuristic algorithm A for P that will guarantee certain quality of the returned suboptimal solutions**



An approximation algorithm A for P

- is a heuristic algorithm which, for any instance p ∈ P, returns in polynomial time a suboptimal solution to p with bounded error (relative to the optimal solution).
- algorithm A trades exactness for running time while guaranteeing limited error of the returned solutions.

A randomized algorithm A for P

- is a heuristic algorithm which, for any instance p ∈ P, returns in polynomial time a solution whose probability of being false is < 1.
- such an algorithm trades certainty for running time while guaranteeing limited probability of error in the solutions.

Computational Problems in General

Notation:

- P … a computational problem
- I … the set of instances of P
- x … $x \in I$, an instance of P
- S(x) … the set of all feasible solutions of $x \in I$
- $S = \bigcup_{x \in I} S(x)$ … the set of all feasible solutions of all instances of P
- $P \subseteq I \times S$ … a computational problem is a set of pairs (x,s), where $x \in I$ and $s \in S(x)$ is a feasible solution of x

E.g. P = copy a given set of files to the smallest number of CDs of given capacity; x = a set of files of given sizes and CDs of given capacities; S(x) = set of possible mappings of given files to given CDs

(cont'd)

Kinds of computational problems:

- **Search problems**: Find any feasible solution $s \in S(x)$ to instance $x \in I$
  - Example. Copy a given set of files to given CDs
- **Optimization problems**: Find the best feasible solution $s^* \in S(x)$ to $x \in I$
  - Example. Copy a given set of files to minimal number of given CDs
- **Decision problems**: Decide whether or not $x \in I$ meets a given condition
  - Example. Can a given set of files be copied to 5 given CDs?

We will be interested

- in hard optimization problems
  and their solving with algorithms
  - that run in polynomial time
  - and return good suboptimal solutions (not necessarily optimal ones)

Algorithm Analysis

Models of Computation

- Selected characteristic properties of a family of computing machines are gathered in an abstract model of computation.
- Some models: DFA, SA, TM, RAM, μ-recursive functions, λ-calculus, general recursive functions, Markov algorithms, PRAM, …

- We use models to evaluate computation resources (time and space) required to run an algorithm and/or solve a computational problem.
- Models with logarithmic cost criterion
  - time/space required to perform an operation depends on operands' size
  - lore precise and more difficult analyses
- Models with uniform cost criterion
  - time/space required to perform an operation is constant
  - less precise and easier analyses

(cont'd)

Problem instances

Algorithms can only solve instances of a problem.

- Obtained when we substitute formal parameters of a problem with actual ones
- Actual parameters are encoded in some alphabet Σ, usually $\Sigma = \{0,1\}$.
- An instance $x \in I$ of a problem is encoded into a word $e(x)$ over Σ.
- $|e(x)|$, the length of the code $e(x)$, is the size of $x \in I$. Time and space complexity are defined to be functions $T(n)$ and $S(n)$ of $n = size(x)$.

Worst- , best-, average-case analyses

How fast do time/space complexity $T(n)$ and $S(n)$ grow when n grows?

- Worst-case analysis tells us how fast grow $T(n)$ and $S(n)$ when solving most unfavorable instances. Results are too pessimistic.
- Best-case analysis tells us how fast grow $T(n)$ and $S(n)$ when solving most favorable instances. Results are too optimistic.
- Average-case analysis tells us how fast grow $T(n)$ and $S(n)$ when solving randomly chosen instances. The analysis is usually more difficult. Results are more realistic.

(cont'd)

Asymptotic notation

How fast do T(n) and S(n) grow when n is large and tends to infinity?

- f(n) = O(g(n)) … f grows at most as fast as g
- f(n) = Ω(g(n)) … f grows at least as fast as g
- f(n) = Θ(g(n)) … f grows as fast as g

Computational complexity of an algorithm

- is the asymptotic growth of the execution time T(n) or space S(n) required by an algorithm A to solve instances $x \in I$ of size(x) = n (for n large and growing)

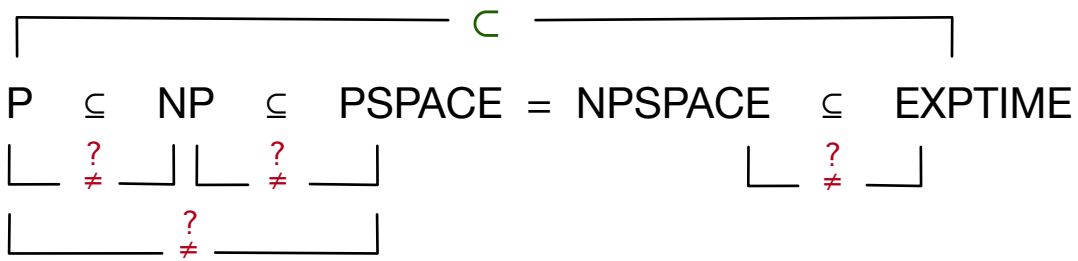
Computational complexity of a problem

- O(g(n)) is an upper bound for time/space complexity of a problem P if ∃alg. A (for P): A has time/space complexity O(g(n)).
- Ω(g(n)) is a lower bound for time/space complexity of a problem P if ∀alg. A (for P): A has time/space complexity Ω(g(n)).
- Θ(g(n)) is the time/space complexity of a problem P if O(g(n)) and Ω(g(n)) are upper and lower bounds for time/space complexity of P

(cont'd)

What is and what isn't known

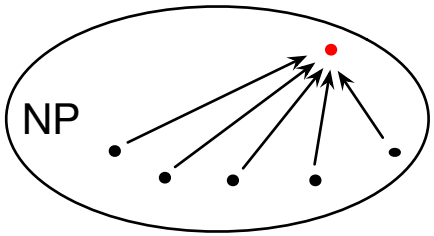
- At least one of the ≠ must be ≠. Currently we don't know which.

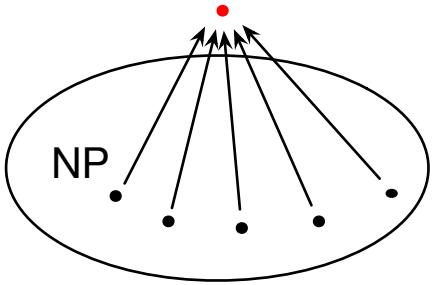


NP-complete and NP-hard problems

- NP-complete problem is in NP and every problem in NP can be reduced to it in polynomial time

Every problem in NP can be reduced to NP-hard problem in polynomial time





Computational Complexity of Optimization problems

Optimization Problems

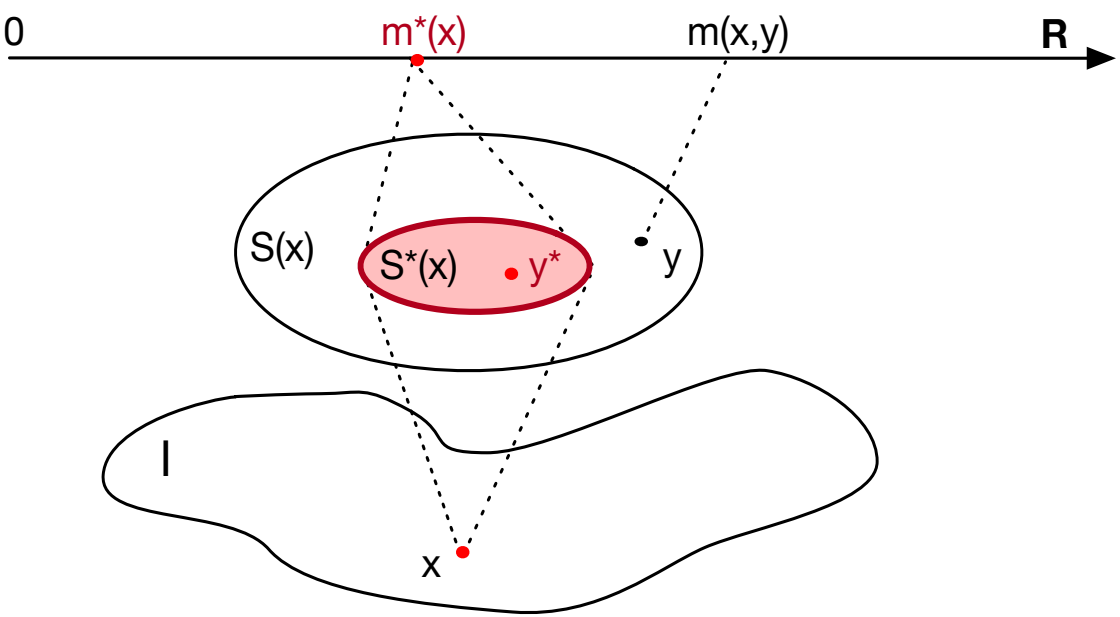
- An optimization problem P is a 4-tuple P = (I, S, m, goal), where
  - I … the set of all instances of P
  - S … a function that maps $x \in I$ to S(x), the set of all feasible solutions to x
  - m … a value function that maps (x,y) to $m(x,y) \in \mathbf{R}^+$, where $y \in S(x)$
  - goal
    - = min … for minimization problems P (searching for a $y \in S(x)$ that minimizes m(x,y))
    - = max … for maximization problems P (searching for a $y \in S(x)$ that maximizes m(x,y))
- An optimal solution of an instance $x \in I$ is a feasible solution $y^* \in S(x)$ such that $\text{goal}_{y \in S(x)}\{v \in \mathbf{R}^+ \mid v = m(x,y)\})$ (i.e., y^* minimizes/maximizes m(x,y)).

Notation: S*(x) … the set of all optimal solutions to x;

m*(x) … optimal value m(x,y*) of an optimal solution y* to x .

(cont'd)

Graphically:



Forms of optimization problems:

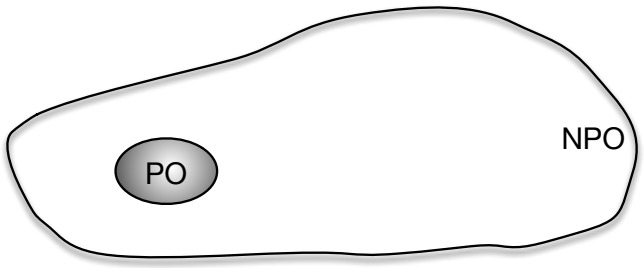
- constructive, P_{con} : compute m*(x) and y*(x), for a given x ∈ I
- nonconstructive, P_{non} : compute m*(x) , for a given x ∈ I
- decision, P_{dec} : decide, for given x ∈ I and K ∈ **R**, whether
  - $m^*(x) \leqslant K$ [if goal = min]
  - $m^*(x) \geqslant K$ [if goal = max]

Complexity Classes of Optimization Problems

- **NPO** is the class of optimization problems P = (I,S,m,goal) such that
  - the question $x \in^{?} I$ can be decided in poly(|x|) time;
  - the question $y \in^{?} S(x)$ can be decided in poly(|x|) time;
  - |y| = poly(|x|), for $y \in S(x)$;
  - m(x,y) can be computed in poly(|x|,|y|) time.

  So, it is only the computation of m*(x) and y*(x) that matters and dictates the time complexity of solving the problem P (i.e. its instances). (All other checking is fast.)

- **PO** is a subclass of **NPO**; it contains all optimization problems whose constructive forms P_{con} are solvable in polynomial time; that is, $(\forall P_{con} \in \mathbf{PO})\ (\exists \text{alg } A)\ (\forall x \in I)$: A returns y*(x) and m*(x) in poly(|x|) time.



- We will find more classes in the following.

NP-hard Optimization Problems

For the proofs of the following theorems see the book Borut Robič, *Aproksimacijski algoritmi*, Založba UL FRI, 2009.

- **Theorem.** $P \in \mathbf{NPO} \Rightarrow P_{dec} \in \mathbf{NP}$
- **Definitions.** An oracle for a problem P is an abstract device which, for any instance $x \in I$ of P, returns in one step the solution to x. A problem P_1 is Turing-reducible to a problem P_2, written $P_1 \preccurlyeq P_2$, if there is an algorithm R that solves P_1 by using the results of finitely many calls to an oracle for P_2. If such an R has polynomial time complexity, then we say that P_1 is polynomially Turing-reducible to P_2, and denote this by $P_1 \preccurlyeq^p P_2$.
- **Definition.** An optimization problem P is NP-hard if $P' \preccurlyeq^p P$, for every $P' \in \mathbf{NPO}$.
- **Consequence.** P_{dec} is NP-complete $\Rightarrow$ P is NP-hard
- **Theorem**. $\mathbf{P} \neq \mathbf{NP} \Rightarrow \mathbf{PO} \neq \mathbf{NPO}$

- **Comment.** Since we strongly suspect that $\mathbf{P} \neq \mathbf{NP}$, we believe that there are optimization problems which cannot be solved in polynomial time. The obvious candidates for such problems are the NP-hard optimization problems.

APPROXIMATE SOLVING OF OPTIMIZATION PROBLEMS

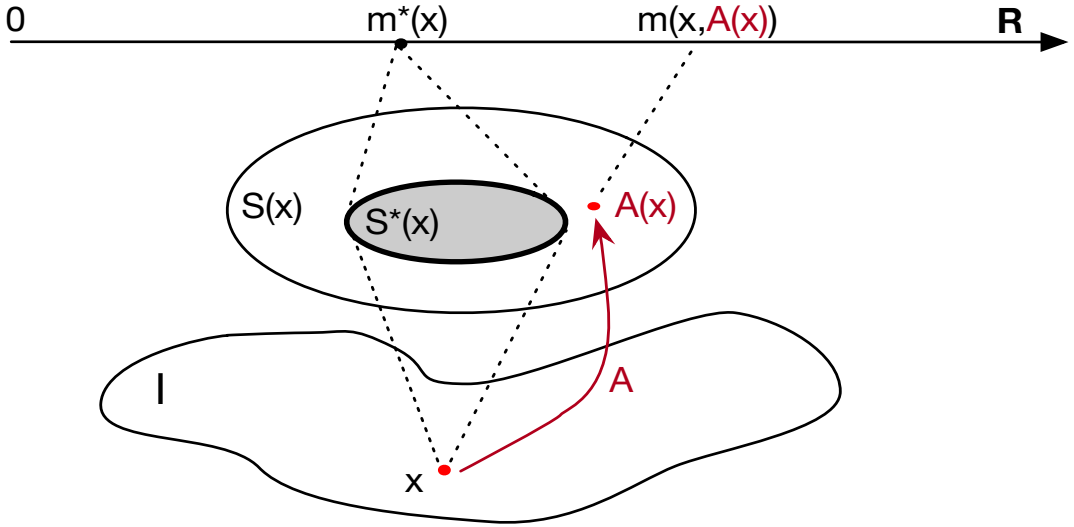
When we face an NP-hard optimization problem we usually give up searching for an exact polynomial-time algorithm.

Instead, we may search for a heuristic algorithm that trades precision of solutions for execution time. Such an inexact algorithm is called the approximation algorithm.

Approximation Algorithms

Definition

- An algorithm A is an approximation algorithm (a.a.) for P = (I,S,m,goal) if, for any x ∈ I, the algorithm A returns a feasible solution A(x) ∈ S(x).
- Graphically (goal = min):



- We will only be interested in a.a.'s A with polynomial time complexity in |x| (= n).

(cont'd)

Quality of Approximate (i.e. Sub-optimal) Solutions

- A(x) is not necessarily optimal solution (i.e. not necessarily $A(x) \in S^*(x)$), so we may have $m(x,A(x)) \neq m^*(x)$.
- We want to find an a.a. A such that the "difference" between m(x,A(x)) and m*(x) is as "small" as possible (i.e. A(x) is as "close" as possible to some $y^* \in S^*(x)$).
- Therefore, we must somehow evaluate the "difference" between m(x,A(x)) and m*(x), i.e. the quality of the approximate solution A(x).
- **Question**: How can we possibly measure the quality of A(x) (and hence of A) in terms of the optimal value m*(x), when m*(x) is unknown?
- **Answer.** There are three approaches to the definition of this quality:
  - Absolute error
  - Relative error
  - Performance quotient.

(cont'd)

Absolute Error

- **Definition**. Absolute error of a feasible solution $y \in S(x)$ is defined to be
$$D(x,y) = |m^*(x) - m(x,y)|.$$
- In case of a.a. A, we are interested in the situation where, for $\forall x \in I$, $D(x,A(x))$ is bounded above with a constant.
- **Definition**. A is absolute approximation alg. for P if $\exists k \geqslant 0\ \forall x \in I$: $D(x,A(x)) \leqslant k$.
  Intuitively, for every instance of P, the absolute error made by A is at most k; i.e., the difference between the optimal value m*(x) and the suboptimal value m(x,A(x)) will never be greater than k.
- Unfortunately, NP-hard optimization problems often have no absolute error.
- **Hence**: Absolute error D(x,y) is too sharp a definition to be generally useful.
  We must find a better definition.

(cont'd)

Relative Error

- **Definition**. Relative error of a feasible solution y ∈ S(x) is defined to be

$$E(x,y) = \frac{|m^*(x) - m(x,y)}{\max\{m^*(x), m(x,y)\}}$$

- Note:

$$\text{goal} = \begin{cases} \min & \Rightarrow E(x,y) = 1 - \frac{m^*(x)}{m(x,y)} \\ \max & \Rightarrow E(x,y) = 1 - \frac{m(x,y)}{m^*(x)} \end{cases}$$

, so always $0 \leqslant E(x,y) \leqslant 1$.

In fact: E(x,y) = 0 ⇔ y is optimal. E(x,y) = 1 ⇔ no guarantees for the quality of y.

- In case of a.a. A, we are interested in the situation where, for ∀x ∈ I, E(x,A(x)) is bounded above with a constant.

- **Definition**. A is ε-approximation algorithm for P if $\exists \varepsilon \in [0,1]$ $\forall x \in I$: $E(x,A(x)) \leqslant \varepsilon$.

  Intuitively, for every instance of P, relative error made by A is at most ε; the relative error of the suboptimal value m(x,A(x)) (relative to the optimal value m*(x)) will never be greater than ε.

- Of course, we want ε to be as close as possible to 0.

(cont'd)

Performance quotient

- **Definition**. Performance quotient of a feasible solution $y \in S(x)$ is defined to be

$$R(x,y) = \max\left\{ \frac{m(x,y)}{m^*(x)}, \frac{m^*(x)}{m(x,y)} \right\}$$

- Note:

$$\text{goal} = \begin{cases} \min & \Rightarrow R(x,y) = \frac{m(x,y)}{m^*(x)} \\ \max & \Rightarrow R(x,y) = \frac{m^*(x)}{m(x,y)} \end{cases}$$

, so always $1 \leqslant R(x,y)$.

In fact: $R(x,y) = 1 \Leftrightarrow$ y is optimal. $R(x,y) \to \infty \Leftrightarrow$ no guarantees for the quality of y.

- In case of a.a. A, we are interested in the situation where, for $\forall x \in I$, $R(x,A(x))$ is bounded above with a constant.

- **Definition**. A is r-approximation alg. for P if $\exists r \geqslant 1\ \forall x \in I$: $R(x,A(x)) \leqslant r$.

  Intuitively, for every instance of P, performance quotient of the suboptimal value m(x,A(x)) (relative to the optimal value m*(x)) will never be greater than r.

- We want r to be as close as possible to 1.

(cont'd)

Relations between E(…) and R(…)

- It is easy to show that $E(x,y) = 1 - \frac{1}{R(x,y)}$.
- So: If A is r-a.a., then A is ε-a.a. with $\varepsilon = 1 - \frac{1}{r}$.

  If A is ε-a.a., then A is r-a.a. with $r = \frac{1}{1-\varepsilon}$.

- Therefore, it is not (very) important which of E or R we use.
  In literature, both are used.
  How do we find which is meant?
  That is easy. Example:

  "… 0.5-approximation algorithm …" --- E is meant (since always $0 \leqslant E \leqslant 1$)
  "… 1.5-approximation algorithm …" --- R is meant (since always $1 \leqslant R$)

Approximable problems and the class APX

- We are interested in optimization problems in **NPO** (due to practical importance). Which of these problems can be efficiently solved by approximation algorithms?
- **Definition**. An optimization problem P ∈ **NPO** is ε-approximable if there is a polynomial-time ε-a.a. with $\varepsilon < 1$ for P. (Similarly: P is r-approximable if there is a polynomial-time r-a.a. with $r < \infty$ for P.)
- **Theorem**. Let P ∈ **NPO.** Then: P is ε-approximable $\Leftrightarrow$ P is r-approximable
- Let us gather all ε-approximable problems P ∈ **NPO** in a new class.
- **Definition**. **APX** is the class of all ε-approximable problems in **NPO**.
- **Question**. Is every problem in **NPO** also in **APX**?

  That is: Is every optimization problem in NPO ε-approximable? Equivalently: APX =? NPO. (That would be nice and extremely useful in practice.)
- **Theorem**. **P** ≠ **NP** $\Rightarrow$ **APX** $\subsetneq$ **NPO**.
- Currently we believe that **P** ≠ **NP**, so we believe that there are optimization problems in NPO that have no polynomial-time ε-a.a. with $\varepsilon < 1$ (i.e. efficient ε-a.a).

(cont'd)

Proof. Assume that **P** ⊊ **NP** and D ∈ **NP-P** a decision problem. So there is a nondet. poly-time alg. N which, for ∀x ∈ I_D , computes the answer Yes/No in poly-time p(|x|). Consequently, the length of N's computation of the answer to x is at most p(|x|).

Now define an optimization problem P =(I,S,m,goal) as follows:

- I = {x| |x| ⩾ 2}
- S(x) = {y| y is a code representing N's computation on input x}

  So, if y ∈ S(x), then y is a p(|x|)-long computation containing the answer Yes/No to x.

  Of course, all y ∈ S(x) contain the same answer to x.

- $\mathrm{m(x,y)} := \begin{cases} |x| & \text{if } y \text{ answers Yes} \\ 1 & \text{if } y \text{ answers Yes} \end{cases}$ (Note: m(x,y) = |x| ⟺ answer is yes)
- goal: irrelevant.

Suppose that P ∈ **APX**. Then ∃poly-time ε-a.a. A with ε<1. Now let n_0 ∈ **N** be such that

$$n_0 > \frac{1}{1-\varepsilon} \quad (\text{hence} \frac{n_0 - 1}{n_0} > \varepsilon)$$

We can prove that |x| ⩾ n_0 ⇒ m(x,A(x)) = m*(x). [Omitted]

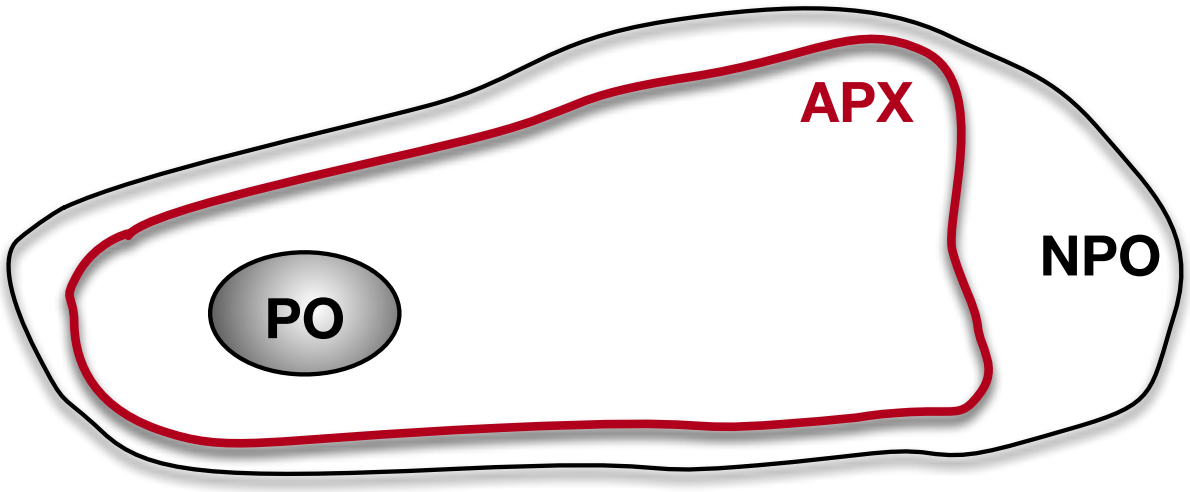
What does that mean? Algorithm A can be used to decide the problem D (i.e., any x ∈ I_D) deterministically in polynomial time. This means that D ∈ **P**. But we assumed D ∈ **NP-P** !

If **P** ⊊ **NP**, the contradiction can only be avoided by abandoning supposition P ∈ **APX.** □

(cont'd)

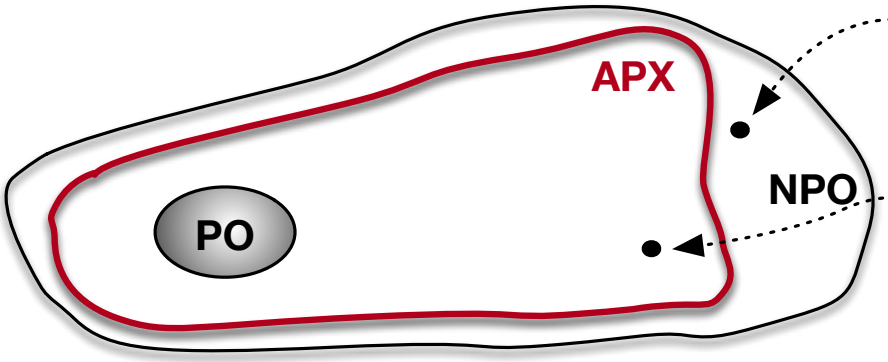
- We now know this:

P ≠ NP ⇒



- We believe that **P** ≠ **NP**, so we believe that there are optimization problems in NPO that have no polynomial-time ε-a.a. with $\varepsilon < 1$ (i.e. no efficient ε-a.a)
- Such a problem is the TRAVELING SALESPERSON problem.

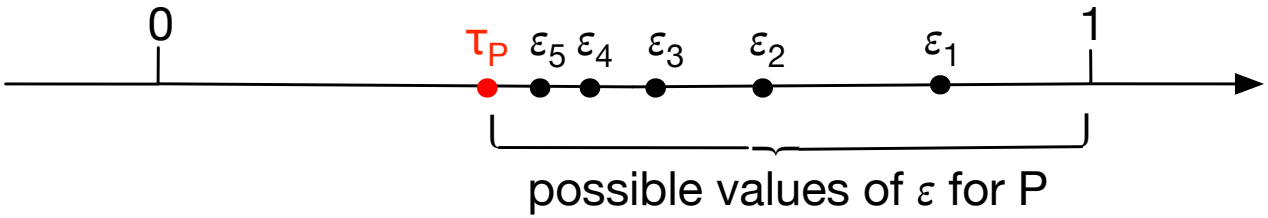
Approximation Threshold



If P is here, then probably (**P≠NP**) there is no poly-time ε-a.a. with $\varepsilon<1$.

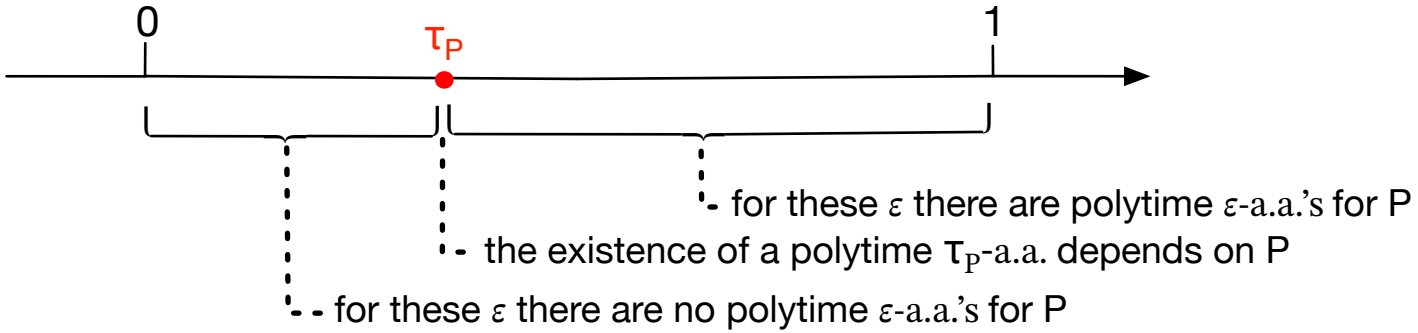
If P is here, then there exists a poly-time ε_1-a.a. with some $\varepsilon_1<1$.

- Then it is natural to ask: Is there a poly-time ε_2-a.a. with $\varepsilon_2<\varepsilon_1$ for P?
  In other words: Is there a more accurate poly-time a.a. for P?
- **Answer**. It depends on the problem P and on the current ε_1 .
- In principle, ε can be reduced to a certain limit value $\tau_P \in [0,1)$, which is associated to (and depends on) the problem P. We call τ_P the approximation threshold of the problem P.



(cont'd)

- So, problems in **APX** differ in their approximation thresholds, and hence in how well their optimal solutions can be approximated by poly-time ε-a.a.'s.
- This means that polynomial-time Turing reductions $\preccurlyeq^p$ between optimization problems do not, generally, preserve their approximation thresholds.
- The lower τ_P, the better approximation of optimal solution of x∈I is possible.



- To prove nonexistence of a poly-time r-a.a. with r in a given interval [1,1+g] (or of a poly-time ε-a.a. with ε in the corresponding interval) we can use the Gap Theorem.
- In case the quality is defined by R (performance quotient), the theorem tells us that

under certain conditions, // see the details in my book //

$r \in [1, 1+g] \Rightarrow \neg\exists$ A: A is a poly-time r-a.a. with $r \in [1, 1+g]$ for P

(unless **P** = **NP**).

The value g is called the gap.

(cont'd)

Example (BIN PACKING).

- BIN PACKING is a minimization problem defined as follows.

  Pack n objects of sizes s_i, i=1,2,…,n, in the smallest number of bins each of capacity 1.

- By applying the gap theorem, it can be found that g = ½. Therefore, there is no poly-time r-a.a. with r<1+g = 3/2 for this problem [unless **P** = **NP**]
- This means that every poly-time r-a.a (or ε-a.a.) for BIN PACKING will return suboptimal solutions that are at least 50% worse than the optimal solution.

Proof. For minimization problems R = m/m*. From R ≥ 1+g = 3/2, we obtain m/m* ≥ 3/2 and then m ≥ 3/2 m* and finally m ≥ m* + 1/2 m*. We see that any computed suboptimal value m will exceed the minimal value m* by at least 1/2m* = 50% of m*. □

Approximation Schemes

Motivation

- There exist optimization problems P ∈ **APX** that have approximation thresholds $\tau_P = 0$. These problems allow arbitrarily accurate poly-time ε-a.a.'s.

- Let P be such a problem. Then, for any given sequence $\varepsilon_1, \varepsilon_2, \varepsilon_3, \ldots, \varepsilon_k$, where $0 < \varepsilon_k < \ldots < \varepsilon_3 < \varepsilon_2 < \varepsilon_1 < 1$, there exists a sequence $A_{\varepsilon 1}, A_{\varepsilon 2}, A_{\varepsilon 3}, \ldots, A_{\varepsilon k}$ of poly-time ε_i-a.a.'s (i = 1,2,…,k), whose accuracies improve as i increases.

- Two questions naturally arise:
  - How similar are the algorithms $A_{\varepsilon i}$ (i = 1, 2, …, k) to each other?
  - How much additional time requires $A_{\varepsilon(i+1)}$ compared with $A_{\varepsilon i}$ (i = 1,2, …,k-1)?

Polynomial Approximation Schemes and the Class PTAS

- If the algorithms $A_{\varepsilon 1}$, $A_{\varepsilon 2}$, $A_{\varepsilon 3}$, …, $A_{\varepsilon k}$ completely differ one from another then this is not fine, as we must must design, for each smaller ε_{i+1}, a new algorithm $A_{\varepsilon(i+1)}$ – this requires inventiveness, creativity, lucky ideas, time, …
- However, it may happen that $A_{\varepsilon 1}$, $A_{\varepsilon 2}$, $A_{\varepsilon 3}$, …, $A_{\varepsilon k}$ are similar one to another, even so similar that they share a common framework (have uniform structure) $\mathcal{A}(\varepsilon)$, where ε is a formal parameter. Then we can write $A_{\varepsilon i} = \mathcal{A}(\varepsilon_i)$.
- So, to obtain a more accurate approximation algorithm $A_{\varepsilon(i+1)}$ for this problem we only need to plug $\varepsilon_{(i+1)}$ into $\mathcal{A}(\varepsilon)$ (i.e., $\varepsilon := \varepsilon_{(i+1)}$). Then, $A_{\varepsilon(i+1)} = \mathcal{A}(\varepsilon_{(i+1)})$.
- **Definition**. A framework $\mathcal{A}(\varepsilon)$ described above is called the polynomial approximation scheme (PAS) for the optimization problem P ∈ **APX**.

(cont'd)

- It is good if $P \in$ **APX** has a PAS. Let us gather all such problems in a class:

  Definition. The class **PTAS** contains all problems $P \in$ **NPO** that have PAS.

  The acronym PTAS stands for ""polynomial-time approximation schemes".

- **Theorem. PTAS** $\neq \emptyset$ (i.e., the class is not empty).

  Proof. Later we will show that the problem KNAPSACK is in **PTAS**. □

- **Theorem**. **PTAS** $\subseteq$ **APX**

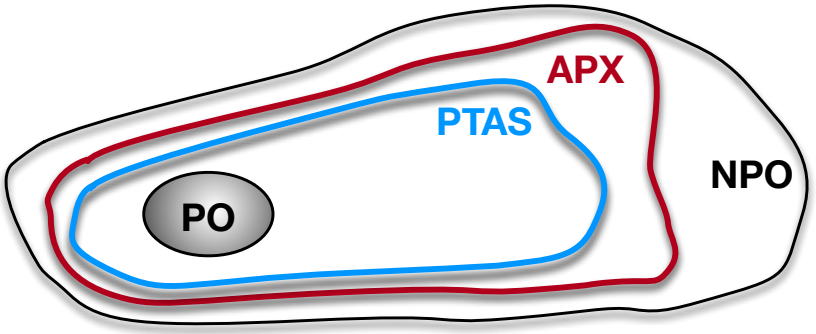
  Proof. If a problem has PAS, then by definition P is in **APX.** □

- **Theorem**. **P** $\neq$ **NP** $\Rightarrow$ **PTAS** $\subsetneq$ **APX**

  Proof. We will see that BIN PACKING is not in **PTAS** (we already know that it is in **APX**.) □

- Graphically:

P ≠ NP $\Rightarrow$



Fully Polynomial Approximation Schemes and the Class FPTAS

- Let P ∈ **NPO** and $A_{\varepsilon 1}$, $A_{\varepsilon 2}$, …, $A_{\varepsilon k}$ poly-time ε-a.a.'s for P where $\varepsilon_i > \varepsilon_{(i+1)}$. Now, $1/\varepsilon$ represents the accuracy of A_ε: the smaller ε, the greater accuracy $1/\varepsilon$.

- But greater accuracy requires more computation and more time. We want the execution time of A_ε to be a polynomial function in $1/\varepsilon$, e.g. $\mathcal{O}(n^2 \log n \cdot (1/\varepsilon)^2)$ or $\mathcal{O}(n^3 \cdot 1/\varepsilon)$, and not exponential function in $1/\varepsilon$, e.g. $\mathcal{O}(n^2 \cdot 2^{1/\varepsilon})$. Then greater accuracy requires at most polynomial additional time to compute more precise suboptimal results.

- Such a PAS is called the fully polynomial approximation scheme (FPAS) for P.

(cont'd)

- Let us gather all optimization problems that have FPAS into a new class:

  Definition. The class **FPTAS** contains all problems P ∈ **NPO** that have FPAS.

  The acronym FPTAS stands for "fully polynomial-time approximation schemes".

- If P is NP-hard, then it is very useful if P ∈ **FPTAS**. Why? The optimal solution to P can be approximated by FPAS $\mathcal{A}(\varepsilon)$ to arbitrary precision $1/\varepsilon$ in poly-time.

- **Theorem. FPTAS ≠ ∅** (the class is not empty).

  Proof. KNAPSACK is in **FPTAS**. □

- **Theorem**. **FPTAS ⊆ PTAS**

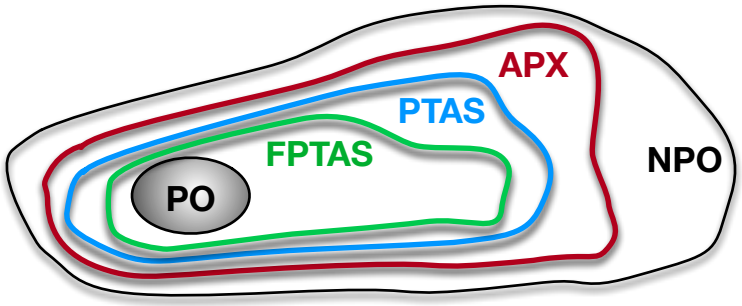
  Proof. Trivial. □

- **Theorem**. **P ≠ NP ⇒ FPTAS ⊊ PTAS**

  Proof.(idea) We show that if **P ≠ NP** then INDEPENDENT SET ∈ **PTAS - FPTAS**. □

- Graphically:

P ≠ NP ⇒



Limits of Approximation

- Let P ∈ **FPTAS**. There is a FPAS $\mathcal{A}(\varepsilon)$ for P. If $\varepsilon \to 0$, the accuracy $1/\varepsilon$ of $\mathcal{A}(\varepsilon)$ increases and, consequently, $\mathcal{A}(\varepsilon)$ returns in poly-time suboptimal solutions that tend to the optimal solution.
- **Question**. Does that mean that $\mathcal{A}(0)$ is exact poly-time algorithm for P?

  Answer. No, our intuition is misleading. In fact, the following holds.

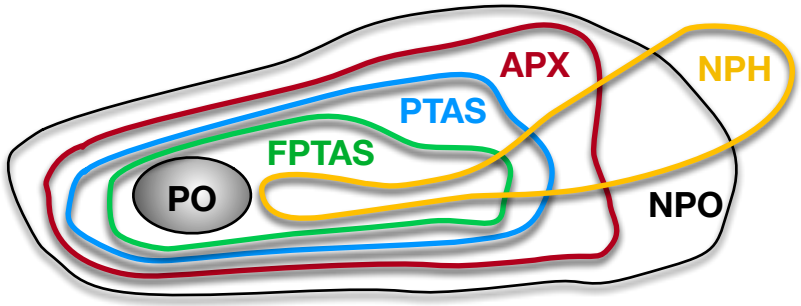
  Theorem. **PO ⊆ FPTAS**

  Proof. □

  Theorem. **P ≠ NP ⇒ PO ⊊ FPTAS**

  Proof. □
- Therefore, if **P ≠ NP**, there are optimization problems which have FPAS, but are <u>not</u> <u>exactly solvable in polynomial time</u>.

P ≠ NP ⇒



The Design of Approximation Algorithms

Introduction

- Approximation algorithms are heuristic algorithms.
- There exist several design methods that we can use to develop an approximation algorithm for a given optimization problem P:
  - Greedy method
  - Focusing on sub-problems of P
  - Sequential partitioning
  - Dynamic programming

Greedy Method

Idea of the Method

- During the construction of a suboptimal solution to an instance of a problem P, select and pick, whenever possible, currently the "best option" available. Here, the "best option" is the one that maximally improves (depending on the goal of P) the value of the current partially constructed solution.
- The method advocates local optimization during the construction of solutions.
- In general, local optimization (greediness) doesn't return optimal solutions. Sometimes, picking an option that is not locally optimal may be paid off later!
- Nevertheless, greedy method may return good suboptimal solutions.
- Consequently, it can be used to design approximation algorithms.

PROBLEM: KNAPSACK

- **Definition**. Fill a knapsack of given load-capacity with most valuable subset of a given set of objects. Formally:
  - Instance of the problem:
    - $X = \{x_1,\ldots,x_n\}$ … set of n objects
    - $a_i \in \mathbf{N}$ … weights of objects i=1,2,…,n
    - $p_i \in \mathbf{N}$ … values of objects i=1,2,…,n
    - $b \in \mathbf{N}$ … load-capacity
  - Feasible solution: every $Y \subseteq X$ such that $\sum_{x_i \in Y} a_i \leqslant b$ … every subset Y of X not heavier than b
  - Quality of a feasible solution Y: $m(Y) = \sum_{x_i \in Y} p_i$ … value of Y
  - Goal: Find a feasible solution Y* which maximizes m(Y).
- KNAPSACK is NP-hard optimization problem.
- Can we design an approximation algorithm for it?

Theorem. **P** ≠ **NP** ⇒ KNAPSACK has no poly-time absolute a.a.

Proof. See my booklet. □

- So, let us try to design a poly-time r-a.a. for KNAPSACK. Since goal = max, a feasible solution Y will be “good” if the performance quotient R = m*/m(Y) is small, close to 1.
- We will describe two attempts: algorithm G and algorithm H.

(cont'd)

Algorithm G

- Idea. Sort X by descending values of ratios (value densities) p_i/a_i, i=1,2,…,n.

  Rename the objects to obtain $x_1, x_2, \ldots, x_n$, where $p_i/a_i \geq p_{i+1}/a_{i+1}$.

- Algorithm G.

```
Y_G := ∅ ;
for i := 1 to n do
  if x_i can be added to Y_G     // i.e., if b ≥ a_i (b = the remaining capacity)
    then Y_G := Y_G U {x_i} ; b := b - a_i
endfor;
return(Y_G).
```

- Algorithm G greedily adds to Y_G, at each step, the object with largest value density.
- Time complexity of algorithm G:
  - sorting requires Θ(n log n) time;
  - adding elements to Y_G requires O(n) time;
  - overall, time complexity is Θ(n log n).
- Hence, algorithm G is poly-time.

(cont'd)

- Quality of suboptimal solutions Y_G

  How much is the performance quotient $R(Y_G) = m^*/m(Y_G)$?

  Theorem. For every $k \in \mathbb{N}$ there exists a (worst-case) instance of KNAPSACK, such that $R(Y_G) = m^*/m(Y_G) > k$.

  Intuitively: Pick a large k, say k = 100. From the above relation $m^*/m(Y_G) > k$ follows that $m(Y_G) < m^*/k = m^*/100$. So, we can obtain, in the worst case, a suboptimal solution Y_G whose value $m(Y_G)$ is k=100-times smaller than the optimal m^*. Since k is arbitrarily large, algorithm G may return arbitrarily bad sub-optimal solutions Y_G.

  Proof. Consider instances with

  $$p_1 = p_2 = \ldots = p_{n-1} = 1 \text{ and } p_n = b-1$$
  $$a_1 = a_2 = \ldots = a_{n-1} = 1 \text{ and } a_n = b.$$

  Then algorithm G returns $Y_G = \{ x_n \}$ whose value is $m(Y_G) = m(\{ x_n \}) = b-1$. The optimal solution is, for example, $\{x_1,x_2,\ldots,x_b\}$, and its value is $m^* = b$. The performance ratio is $m^*/m(Y_G) = b/(b-1)$. Pick an arbitrary $k \in \mathbb{N}$. To complete the proof, there must be an instance meeting the requirement $m^*/m(Y_G) > k$. This is easy: the instance must satisfy the relation $b/(b-1) > k$, so $b < 1 + 1/(k-1)$.

  Note: We have estimated worst-case value of Y_G relative to the unknown optimal value m^*.

  □

Algorithm H

- Idea. Algorithm G could add the most valuable object to Y_G even if its value density was the smallest. Algorithm H uses algorithm G and corrects this.
- Algorithm H.

$Y_H := Y_G$; // apply algorithm G to compute initial Y_H
x_{max} := the most valuable object in X ; // its value is p_{max}
if $m(Y_H) < p_{max}$
then $Y_H := \{x_{max}\}$;
return(Y_G). // So, $m(Y_H) = \max\{m(Y_G), p_{max}\}$.

- Time complexity of algorithm H is $\Theta(n \log n)$. So H is a poly-time algorithm.
- Quality of suboptimal solutions Y_H

What is the performance quotient $R(Y_H) = m^*/m(Y_H)$?

Theorem. $R(Y_H) = m^*/m(Y_H) < 2$.

Proof. See my booklet. □

- **Comment.** Since goal = max, $m(Y_H) \leqslant m^*$ holds. The relation $R(Y_H) = m^*/m(Y_H) < 2$ from the theorem implies $m^*/2 < m(Y_H)$. Therefore, $m^*/2 < m(Y_H) \leqslant m^*$. In other words, algorithm H guarantees to return Y_H whose value $m(Y_H)$ is at 50% of the maximal (optimal) value m^*.
Can we do better? Yes, by using different method of algorithm design, the dynamic programming.

PROBLEM: INDEPENDENT SET

- **Definition**. Find the largest group of guests such that no two guests know each other (largest independent set of guests). Formally:
  - Instance of the problem:
    - Undirected graph G(V,E) … V set of guests; $\{u,v\} \in E \Leftrightarrow$ u and v know each other
  - Feasible solution: every set $W \subseteq V$ such that $u,v \in V \Rightarrow \{u,v\} \notin E$ … every independent set W
  - Quality of a feasible solution W: $m(W) = |W|$ … cardinality of W
  - Goal: Find a feasible solution W* which maximizes m(W).

- INDEPENDENT SET is NP-hard optimization problem.
- Can we design an approximation algorithm for it?

- We will describe a poly-time r-a.a. for INDEPENDENT SET. Since goal = max, a feasible solution W will be “good” if the performance quotient $R = m^*/m(W)$ is small, close to 1.
- We will describe one such algorithm A and consider some possible improvements to it.

(cont'd)

Algorithm A

- Idea.
  1. Find in G the vertex x with smallest degree.
  2. Add x to W.
  3. Delete x and its neighbors from G together with all incident edges.
  4. If G is not empty, go to 1.

- Algorithm A**.**

```
W := ∅ ;
while V ≠ ∅  do
    x := vertex with smallest degree in G(V,E) ;
    W := W U { x } ;
    delete x, its neighbors, and all incident edges from G(V,E) ;
endwhile;
return(W).
```

- In each step, algorithm A greedily adds to W the guest who knows as few as possible other guests.

(cont'd)

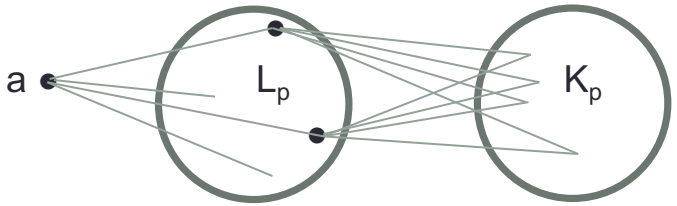
- Time complexity of algorithm A:
  - The loop executes O(|V|) times.
  - Search for x takes O(|V|).
  - Adding x to W and deleting from G(V,E) takes O(|V|) time
  - Overall, time complexity is $\Theta(|V|^2)$. Hence, A is poly-time algorithm.

- Quality of suboptimal solutions W

  How much is the performance quotient R(W) = m*/m(W)?

  Theorem. For every $k \in \mathbb{N}$ there is a (worst-case) instance of INDEPENDENT SET for which R(W) = m*/m(W) > k.

  Proof. See my booklet. Here we mention that the instance (graph G) is of the form



vertex a is linked with every vertex in L_p
L_p … p vertices, completely unconnected
K_p … p vertices, completely connected □

  Intuitively. Algorithm A may return arbitrarily bad sub-optimal solutions W.

  (The reasoning is very much similar to that for the problem KNAPSACK.)

(cont'd)

- Such pathological, worst-case graphs are relatively rare. For this reason the above theorem does not tell us how algorithm A behaves on average.
- **Questions**. Can we tell anything about the average quality of W? Can we use some property of G(V,E) that will allow us to find more about the quality of W?
- **Some answers**.

Definition. The density δ of a graph G(V,E) is defined as $\delta = |E|/|V|$.

Theorem. The quality m(W) is bounded below by density of G(V,E):

$$|V|/(2\delta + 1) \leqslant m(W).$$

Proof. See my booklet. □

Intuitively. Theorem tells us that m(W) is not always arbitrarily small. For example, for sparse graphs with, say $\delta = 3$, m(W) is at least |V|/7.

With δ we can deduce a relation among m(W), m* and δ:

$$W \text{ is feasible solution} \Rightarrow m^*/(\delta + 1) \leqslant m(W)$$

We can also deduce a relation among m(W), m* and d_{max} (max. degree in G):

$$W \text{ is feasible solution} \Rightarrow m^*/(d_{max} + 1) \leqslant m(W)$$

Example. if $d_{max} = 3$, then $m^*/4 \leqslant m(W)$.

PROBLEM: TRAVELING SALESPERSON (TSP)

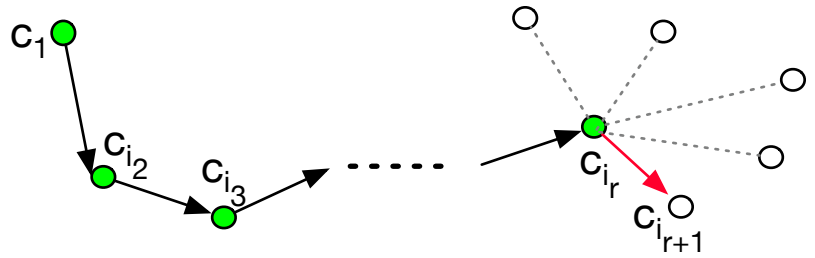
- **Definition**. Cities $c_1,c_2,\ldots,c_n$ are connected by roads of lengths $d_{i,j} \in \mathbb{R}^+ \cup \{\infty\}$. A person must start in c_1, visit every other city exactly once, and return to c_1. In what order must he visit the cities to make a shortest possible cycle? Formally:
-
  - Instance of the problem:
    - $\{c_1,c_2,\ldots,c_n\}$ … set of n cities
    - $D = (d_{i,j})_{nxn}$ … matrix of distances $d_{i,j}$ from c_i to c_j , where $d_{i,j} = \infty \Leftrightarrow c_i, c_j$ are not connected
    - Feasible solution: any permutation $p = (c_{i1}, c_{i2}, \ldots, c_{in})$ of the cities
  - Quality of a feasible solution p: m(p) = the length of the cycle $c_{i1} \rightarrow c_{i2} \rightarrow \ldots \rightarrow c_{in} \rightarrow c_{i1}$.
  - Goal: Find a permutation p* that minimizes m(p).
- TSP is NP-hard optimization problem.
- Can we design an approximation algorithm for it?
- We try with the following intuitively appealing algorithm A.

(cont'd)

Algorithm A

- Idea.
  1. Start in c_1.
  2. In each step, move to the nearest unvisited city.
  3. If all cities are visited, move to c_1 .

- Algorithm A.




```
p := (c1) ;
for r :=1 to n-1 do
    (cir , ci(r+1) ) := shortest edge from cir to an unvisited neighbor;
     p := p U {ci(r+1)}
endfor;
return(p).
```

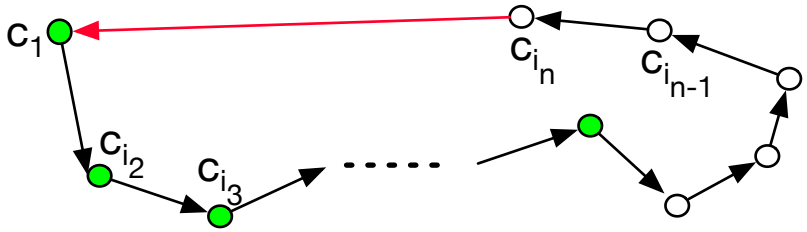
- In each step, algorithm A greedily adds the nearest unvisited city to the end of the currently constructed path.

(cont'd)

- Time complexity of algorithm A:
  - The loop executes n-1 times.
  - The body of the loop takes O(n) time.
  - Overall, time complexity is $O(n^2)$. So A is a poly-time algorithm.

- Quality of suboptimal solutions p

  At each city, the salesperson picks the nearest unvisited city. This greediness may revenge in the last step, when he must return to c_1 via the only road connecting c_{in} to c_1. Since the length of this road can be arbitrarily large (depending on the instance), it can make an arbitrarily large contribution to the length of the constructed shortest path $c_1 \to c_{i2} \to \ldots \to c_{in}$.



Theorem. For every $k \in \mathbb{N}$ there is an instance of TSP such that $m(p) \geq km^*$.

Proof. See my booklet. □

Intuitively. Algorithm A may return arbitrarily bad suboptimal solutions p.

(cont'd)

We must find a better a.a. for TSP – if it exists at all. Does it? Is TSP ∈ **APX**?

Let us make a **hypothesis** $\mathcal{H}$ ≡ TSP ∈ **APX** and see what $\mathcal{H}$ implies.

$\mathcal{H}$ ⇒ ∃ r-a.a. H with r < ∞ for TSP. Then:

- Let G(V,E) be an arbitrary undirected graph with V = $\{c_1,c_2,\ldots,c_n\}$ and let $d_{i,j}$ = 1 if (c_i,c_j) ∈ E, and $d_{i,j}$ = r|V| if (c_i,c_j) ∉ E. Applying the algorithm H, we obtain a suboptimal solution p, for which m(p) ≥ |V|. (Why? Exercise.)
- There are two possibilities for m(p) ≥ |V|: (a) m(p) = |V|, and (a) m(p) > |V|.
- We can show: (a) ⇒ G is Hamiltonian; and (b) ⇒ G is not Hamiltonian. (How? Exercise.)
- So H is a decision algorithm for the problem HAMILTONIAN GRAPH (= "Is graph G Hamiltonian?").
- In addition, H is a poly-time algorithm (since it is r-a.a. due to $\mathcal{H}$).
- But the decision problem HAMILTONIAN GRAPH is known to be NP-complete!
- It follows that **P** = **NP**.

We found that $\mathcal{H}$ ⇒ **P** = **NP**.

- That is: **P** ≠ **NP** ⇒ ¬$\mathcal{H}$, i.e. **P** ≠ **NP** ⇒ TSP ∉ **APX**. And we currently believe that **P** ≠ **NP**.

Therefore: Since we believe that **P** ≠ **NP**, we believe that TSP has no r-a.a. with r < ∞.

If **P** ≠ **NP,** there exist non-approximable NP-hard problems!

Focusing on Subproblems

Idea of the Method

- Since it is likely that there exist non-approximable NP-hard problems, we need to find some other way to deal with such problems.
- Sometimes, an additional restriction imposed upon such a problem P may turn the problem P into an approximable one.
- The obtained problem P' is a subproblem of the original, more general P.
- If the restriction is not too severe, the new problem P' may still be of high practical importance.
- For example, a problem P which can be dealt with in this way is the TSP. The subproblem P' of P that we describe in the following is the METRIC TSP.

PROBLEM: METRIC TRAVELING SALESPERSON (ΔTSP)

- **Definition**. The problem ΔTSP is defined as TSP + two additional restrictions: symmetry (A→B and B→A are equally long) and triangle inequality (A→C→B is at least as long as A→B), where A,B,C are any three cities of TSP.
  Formally:
  - Instance of the problem:
    - $\{c_1, c_2, \ldots, c_n\}$ … set of n cities
    - $D = (d_{i,j})_{nxn}$ … matrix of distances $d_{i,j}$ from c_i to c_j , where $d_{i,j} = \infty \Leftrightarrow c_i, c_j$ are not connected
    - $d_{i,j} = d_{j,i}$ and $d_{i,k} + d_{k,j} \geq d_{i,j}$, for all i,j,k … symmetry and triangular inequality
    - Feasible solution: any permutation $p = (c_{i1}, c_{i2}, \ldots, c_{in})$ of the cities
  - Quality of a feasible solution p: m(p) = the length of the cycle $c_{i1} \rightarrow c_{i2} \rightarrow \ldots \rightarrow c_{in} \rightarrow c_{i1}$.
  - Goal: Find a permutation p* that minimizes m(p).
- ΔTSP is a subproblem of TSP (if an algorithm solves TSP, then it also solves ΔTSP).
- ΔTSP is known to be NP-hard optimization problem.
- Can we design an approximation algorithm for this subproblem of TSP?
  Yes, we will describe two such algorithms, denoted by B and C.

(cont'd)

Algorithm B

- Idea.
  1. In G(V,E,d), construct a minimum spanning tree T.

     If m(T) := "length of T", then m(T) < m(p*).

     Proof. If we delete an arc from p*, we obtain a spanning tree T', whose length is at least m(T): m(T) ⩽ m(T') < m(p*). □

  2. From a node of T construct a traversal S of T, such that each edge of T is traversed twice.

     If m(S) := "length of S, then m(S) = 2m(T) < 2m(p*).

     (S is not a feasible solution to ΔTSP, not Hamiltonian)

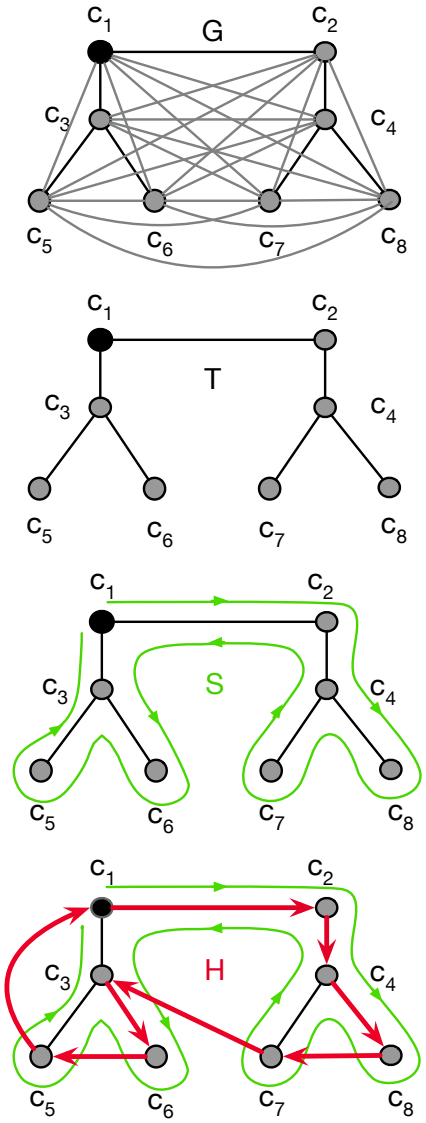
  3. Construct from S a Hamiltonian cycle H in G by passing over every previously visited node.

     (H is a feasible solution to ΔTSP.)

     Due to Δ-inequality: m(H) ⩽ m(S) < 2m(p*).

  - Algorithm B is poly-time algorithm (steps are polynomial).

- **Summary**: Algorithm B is a 2-a.a. for ΔTSP.
- **Question**: Can we do better? Yes, this is algorithm C.



(cont'd)

Algorithm C (Christofides)

- Idea.
  - In G(V,E,d), construct a minimum spanning tree T.

Fact : Any graph has even number of nodes of odd degrees.
Let V_{odd} := set of nodes of T of odd degree. Then $|V_{odd}| = 2k$.

Example: In our T (see figure), $|V_{odd}| = |\{c_3,c_4,c_5,c_6,c_7,c_8.\}| = 6$.

Definiton. A matching in V_{odd} is any partition M of V_{odd} into disjunct pairs $\{c_{i1}, c_{j1}\}, \{c_{i2}, c_{j2}\}, \ldots, \{c_{ik}, c_{jk}\}$. The weight of a matching M is defined to be $w(M) = d_{i1,j1} + d_{i2,j2} + \ldots + d_{ik,jk}$, the sum of the distances associated to the pairs of M.

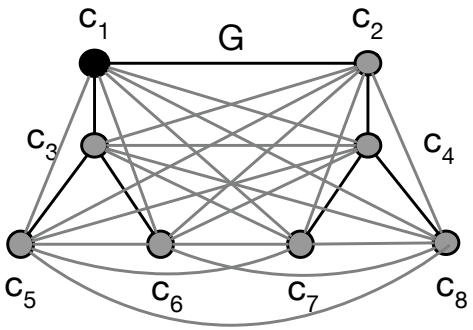
Example: In our T, a matching in V_{odd} is $M = \{\{c_3,c_4\}, \{c_5,c_6\}, \{c_7,c_8\}\}$.
Another matching is, for example, $\{\{c_3,c_7\}, \{c_4,c_5\}, \{c_6,c_8\}\}$.
There are $|V_{odd}|(|V_{odd}|-1)/2$ matchings in V_{odd} . (Prove it.)

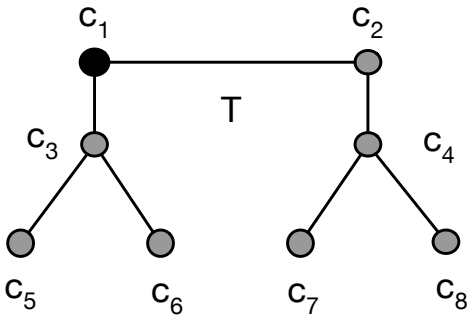
A minimal matching M* in V_{odd} is the one with minimal weight.

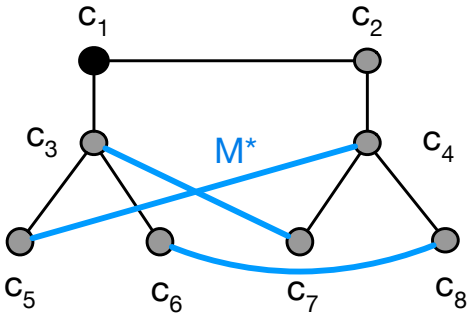
Fact: M* in V_{odd} can be found in polynomial time in $|V_{odd}|$.

- Construct a minimal matching M* in V_{odd} .

Let minimal matching be $M^* = \{\{c_1, c_2\}, \{c_3, c_4\}, \ldots, \{c_{2k-1}, c_{2k}\}\}$.
(We renamed the cities to simplify notation.)







(cont'd)

- Add the pairs (edges) of M* to the tree T.

  Denote the resulting graph by T + M*.
  Note: every node of T + M* has even degree.

  Fact: If all nodes of a graph have even degrees, then the graph is Eulerian (has a cycle traversing each edge exactly once.) Therefore, the graph T + M* is Eulerian.

- Construct an Eulerian cycle S in T U M*.

  But S is not Hamiltonian cycle, not a feasible solution to ΔTSP.

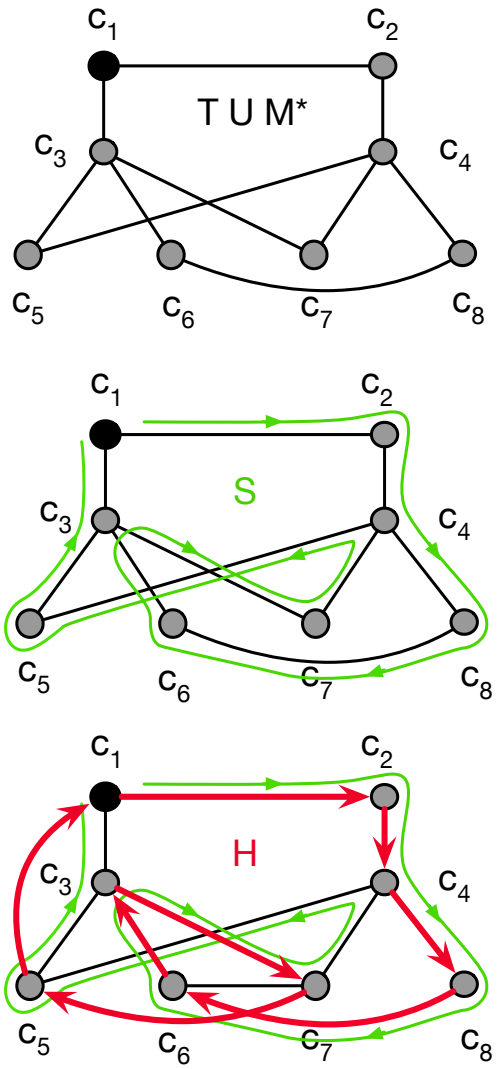
- Construct from S a Hamiltonian cycle H in G by passing over every previously visited node (as in algorithm B).

□

- **Theorem**. m(H) < 1.5 m(p*).

  Algorithm C returns suboptimal solution that is at most 50% larger than the minimal (optimal) one. The algorithm is also poly-time (all steps are polynomial).

- **Summary**: Algorithm C is a 1.5-a.a. for ΔTSP.



Sequential (iterative) Partitioning

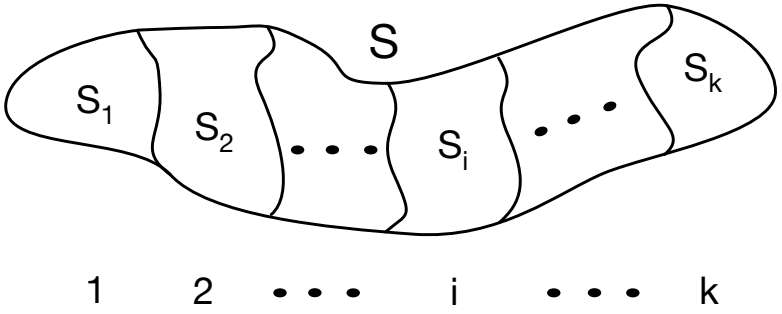
- Some optimization problems have feasible solutions that are partitions of some set. We call such optimization problems partitioning problems.
- There are several methods for solving partitioning problems. Here we are particularly interested in the method called the sequential partitioning.

Partitioning problems

- **Definition**. A partition of a set S= $\{x_1,x_2,\ldots,x_n\}$ is a set P = $\{S_1,S_2,\ldots,S_k\}$, such that
  - $1 \leqslant k \leqslant |S|$
  - $S_i \neq \emptyset$ and $S_i \subseteq S$, for i = 1,2,…,k
  - $S_1 \cup S_2 \cup \ldots \cup S_k = S$
  - $S_i \cap S_j \neq \emptyset$, for i,j = 1,2,…,k.

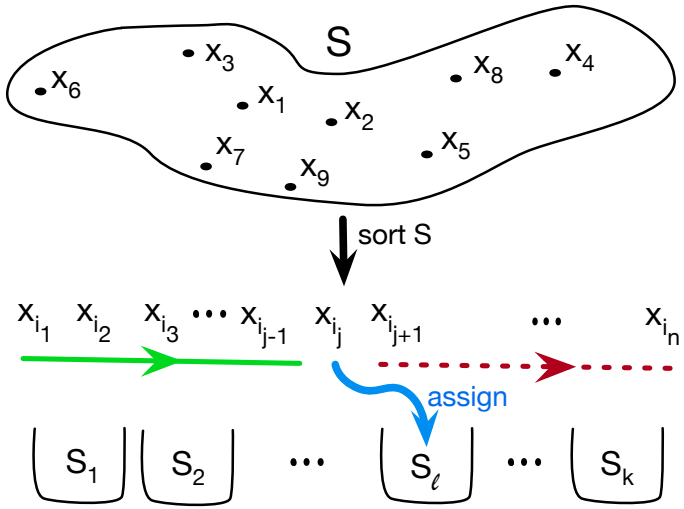
  S_i is called the component of the partition P.

- A partition P of S is completely defined by specifying a surjective function $f: S \rightarrow \{1,2,\ldots,k\}$, such that $f^{-1}(i) = S_i$, i = 1,2,…,k; that is, *f* maps each element of S to the index of a component of P.



The Method of Sequential Partitioning

- The Method
  1. Sort S // Let $(x_{i1},x_{i2},\ldots,x_{in})$ be sorted S
  2. Run through S assigning each x_i to some component of P by using some criterion.



- **Algorithm** Seq_Part (S)

  instance: $S = \{x_1, x_2, \ldots, x_n\}$

  output: partition function $f : S \rightarrow \{1,2,\ldots,k\}$

  begin

  for i = 1 **to** n **do** $f(x_i) := 0$;

  Sort S; // sort S into the order $(x_{i1}, x_{i2}, \ldots, x_{in})$ depending on the problem

  for $i := i_1, i_2, \ldots, i_n$ **do**

  if x_i belongs to S_ℓ // x_i should be in S_ℓ according to the criterion that depends on the problem

  then $f(x_i) := \ell$

  endfor;

  return *f*

  end.

Clearly, S must be known in advance to sort it. We say that the above algorithm is an offline. If S were not known in advance (if x_i are arriving one by one in succession), S couldn't be preprocessed, and the algorithm would lack useful information about S as a whole. We say that the algorithm would be online.

PROBLEM: TASK SCHEDULING

- **Definition**. There are n tasks, each of known length (duration), to be completed, and p identical processors. Allocate the tasks to the processors so that all tasks will complete as soon as possible. Formally:
  - Instance of the problem:
    - $T = \{x_1, x_2, \ldots, x_n\}$ … set of tasks
    - ℓ_i … length (duration, processing time) of task i
    - p … number of processors
    - Feasible solution: any function $\pi : T \rightarrow \{1,2,\ldots,p\}$ … which allocates task x to processor $\pi(x)$
  - Quality of a feasible solution π: $m(\pi) = \max_{1 \leqslant i \leqslant p} \sum_{\pi(x_j)=i} \ell_j$ … maximal allocated load to a processor
  - Goal: Find a function π^* that minimizes $m(\pi)$.

- TASK SCHEDULING is NP-hard optimization problem.

- With the method of sequential partitioning we design two polytime a.a.'s for the problem: the LS (list scheduling) algorithm and the LPT (largest processing time) algorithm.

Algorithm LS (List Scheduling)

- Idea.
  1. Algorithm will be online (there will be no preprocessing (e.g. sorting) of the whole T)
  2. Each arrived task is allocated to the processor with the currently smallest load
- Algorithm

Algorithm LS.
instance: $T = \{x_1, x_2, \ldots, x_n\}$; $\ell_i \in \mathbb{R}^+, i = 1, 2, \ldots, n$; $p \in \mathbb{N}, p \leq n$
output: function $\pi_{LS} : T \to \{1, 2, \ldots, p\}$
begin
  for $i := 1, 2, \ldots, n$ **do** $\pi_{LS}(x_i) := 0$;
  for $i := 1, 2, \ldots, n$ **do**
    $k :=$ processor with currently smallest load;
    $\pi_{LS}(x_i) := k$
  endfor;
  return π_{LS}
end.

- Time complexity of LS: LS is a poly-time algorithm (loop executes n times; body takes $\mathcal{O}$(n) time,).
- Quality of suboptimal solutions π_{LS}

Theorem. $m(\pi_{LS}) \leqslant (2 - \frac{1}{p})m(\pi^*_{LS})$ (Proof. See booklet. □)

Intuitively. For p>1, LS returns π_{LS} such that T completes in time which is ⩽100% larger than m*.
In other words, LS is a polytime $(2 - \frac{1}{p})$-a.a. for the TASK SCHEDULING problem.

Can we do better? Yes.

Algorithm LPT (Longest Processing Time First)

- Idea.
  1. Sort all tasks in T by their decreasing lengths (the algorithm will be offline)
  2. Prevent the situation possible in LS where x_n of large ℓ_n could spoil a good allocation

- Algorithm

Algorithm LPT.
instance: $T = \{x_1, x_2, \ldots, x_n\}$; $\ell_i \in \mathbb{R}^+, i = 1, 2, \ldots, n$; $p \in \mathbb{N}, p \leq n$
output: function $\pi_{LPT} : T \to \{1, 2, \ldots, p\}$
begin
  for $i := 1, 2, \ldots, n$ **do** $\pi_{LPT}(x_i) := 0$;
  sort T into $(x_{i_1}, x_{i_2}, \ldots, x_{i_n})$ so that $\ell_{i_1} \geq \ell_{i_2} \geq \ldots \geq \ell_{i_n}$;
  for $i := i_1, i_2, \ldots, i_n$ **do**
    $k :=$ processor with currently smallest load;
    $\pi_{LPT}(x_i) := k$
  endfor;
  return π_{LPT}
end.

- Time complexity of LPT: LPT is a poly-time algorithm with time complexity $\mathcal{O}(n^2)$.
- Quality of suboptimal solutions π_{LPT}

  Theorem. $m(\pi_{LPT}) \leqslant (\frac{4}{3} - \frac{1}{3p}) m(\pi^*_{LPT})$ (Proof. See booklet. □)

  Intuitively. If p>1, T allocated by π_{LPT} requires ⩽ 33% more time to complete than necessary.
  LPT is a polytime $(\frac{4}{3} - \frac{1}{3p})$-a.a. for the TASK SCHEDULING problem.

PROBLEM: BIN PACKING

- **Definition**. Allocate given objects of known sizes to minimum number of bins of equal capacities. Formally:
  - Instance of the problem:
    - T = $\{x_1, x_2, \ldots, x_n\}$ … set of objects
    - $\ell_i \in (0,1]$ … normalized sizes of objects
    - 1 … capacity of each bin
  - Feasible solution: $\pi : T \rightarrow \mathbb{N}$ such that $\mathrm{load}(j) \stackrel{\mathrm{def}}{=} \sum_{\pi(x_i)=j} \ell_i \leqslant 1$, for j = 1,2, ...
  - Quality of a feasible solution π: $m(\pi) = \mu \stackrel{\mathrm{def}}{=} \max_{x_i \in T} \pi(x_i)$ … the number of bins used by π
  - Goal: Find a function π^* that minimizes m(π).
- BIN PACKING is NP-hard optimization problem.
- Using the method sequential partitioning we will design several polytime a.a.'s for the BIN PACKING problem: the NF, FF, FFD, and BFD algorithm.

Algorithm NF (Next Fit)

- Idea.
  1. Algorithm will be online (there will be no preprocessing (e.g. sorting) of T)
  2. Allocate x_1 to 1st bin; each next object allocate to the last used bin if there is enough space for the object, otherwise allocate the object to a new, empty bin.
- Algorithm

Algorithm NF.
instance: $T = \{x_1, x_2, \ldots, x_n\}$; $\ell_i \in (0, 1], i = 1, 2, \ldots, n$
output: $\pi_{NF} : T \to \{1, 2, \ldots, \mu_{NF}\}$, where $\text{load}(j) = \sum_{\pi_{NF}(x_i)=j} \ell_i \leq 1$ for $j = 1, \ldots, \mu_{NF}$

```
begin
  μ_NF := 1;                                  // at least one bin is needed
  for i := 1 to n do
    if load(μ_NF) + ℓ_i ≤ 1                   // currently last bin μ_NF can accept x_i
      then π_NF(x_i) := μ_NF                  // allocate x_i to it
      else
        begin                                 // introduce a new empty bin
          μ_NF := μ_NF + 1;                   // make it currently last
          π_NF(x_i) := μ_NF                   // allocate x_i to it
        end
  endfor;
  return π_NF
end.
```

(cont'd)

- Time complexity of NF

  NF is a poly-time algorithm (loop executes n times; loop-body takes $\mathcal{O}(1)$ time).

- Quality of suboptimal solutions π_{NF}

  Theorem. $m(\pi_{NF}) \leqslant 2m(\pi^*)$ (Proof. See booklet. □)

  Intuitively. NF returns allocation π_{NF} which uses $\leqslant$100% more bins than it would be necessary.

  Comment. The upper bound is tight: there are instances for which $m(\pi_{NF}) = 2m(\pi^*)$.

- A **drawback** of algorithm NF

  NF tries to allocate only to the last bin. Why not try to allocate to some previous bin?

.

Algorithm FF (First Fit)

- Idea.
  1. This algorithm too will be online (no preprocessing of the whole T).
  2. Allocate x_1 to 1st bin; allocate each next object to the first used bin that can accept it. If there is no such bin, allocate the object to a new bin. (Eliminates the drawback of NF.)

- Algorithm

Algorithm FF.
instance: $T = \{x_1, x_2, \ldots, x_n\}$; $\ell_i \in (0,1], i = 1, 2, \ldots, n$
output: $\pi_{FF} : T \to \{1, 2, \ldots, \mu_{FF}\}$, where $\text{load}(j) = \sum_{\pi_{FF}(x_i)=j} \ell_i \leq 1$ for $j = 1, \ldots, \mu_{FF}$

```
begin
  μ_FF := 1;                                   // at least one bin is needed
  for i := 1 to n do
    if ∃j ∈ {1, 2, ..., μ_FF} : load(j) + ℓ_i ≤ 1   // some used bin j can take x_i
      then π_FF(x_i) := smallest such j          // allocate x_i to first such bin
      else
        begin                                    // introduce a new empty bin
          μ_FF := μ_FF + 1;                      // remember it is currently last
          π_FF(x_i) := μ_FF                      // allocate x_i to it
        end
  endfor;
  return π_FF
end.
```

(cont'd)

- Time complexity of FF

  FF is a poly-time algorithm (the loop executes n times; loop-body takes $\mathcal{O}(n)$ time).

- Quality of suboptimal solutions π_{FF}

  Theorem. $m(\pi_{FF}) \leqslant 1.7\, m(\pi^*) + 2$ (Proof. See booklet. □)

  Intuitively. FF returns allocation π_{FF} that uses $\leqslant$70% more bins than necessary.
  Comment. The upper bound is tight: there are instances for which $m(\pi_{FF}) = 1.7\, m(\pi^*)$.

- A **drawback** of algorithm FF

  If the largest objects arrive at the end, it may happen that they cannot be allocated to the used bins (but they could be allocated if the allocated objects were reallocated).

Algorithm FFD (First Fit Decreasing)

- Idea.
  1. This algorithm will be offline (the whole T can be preprocessed, e.g. sorted).
  2. Sort T = {x_1 ,x_2 , …, x_n } by decreasing sizes ℓ_i .
  3. Apply algorithm FF on the sorted T. (This eliminates the drawback of FF.)

- Algorithm

Algorithm FFD.
instance: $T = \{x_1, x_2, \ldots, x_n\}$; $\ell_i \in (0,1], i = 1, 2, \ldots, n$
output: $\pi_{FFD} : T \to \{1, 2, \ldots, \mu_{FFD}\}$, where $\text{load}(j) = \sum_{\pi_{FFD}(x_i)=j} \ell_i \leq 1$ for $j = 1, \ldots, \mu_{FFD}$

```
begin
  sort T into (x_{i_1}, x_{i_2}, ..., x_{i_n}) such that ℓ_{i_1} ⩾ ℓ_{i_2} ⩾ ... ⩾ ℓ_{i_n};
  μ_FFD := 1;                                   // at least one bin is needed
  for i := i_1, i_2, ..., i_n do
    if ∃j ∈ {1, 2, ..., μ_FFD} : load(j) + ℓ_i ≤ 1   // some used bin j can take x_i
      then π_FFD(x_i) := smallest such j          // allocate x_i to first such bin
      else
        begin                                     // introduce a new empty bin
          μ_FFD := μ_FFD + 1;                     // remember it is currently last
          π_FFD(x_i) := μ_FFD                     // allocate x_i to it
        end
  endfor;
  return π_FFD
end.
```

(cont'd)

- Time complexity of FFD

  FFD is a poly-time algorithm (sorting takes $\mathcal{O}(n \log n)$ time and FF takes $\mathcal{O}(n^2)$ time).

- Quality of suboptimal solutions π_{FFD}

  Theorem. $m(\pi_{FFD}) \leqslant 1.5\, m(\pi^*) + 1$ (Proof. See booklet. □)

  Intuitively. FFD returns allocation π_{FFD} that uses $\leqslant 50\%$ more bins than necessary.

  Comment. The upper bound is not tight. Actually, we know that π_{FFD} is better that:

  Theorem. $m(\pi_{FFD}) \leqslant \frac{11}{9} m(\pi^*) + 4$ (Proof. See booklet. □)

  Comment. This upper bound is tight. So, allocation π_{FFD} uses $\leqslant 22\%$ more bins than necessary.

Algorithm BFD (Best Fit Decreasing)

- Idea.
  1. This algorithm is offline (the whole T will be sorted).
  2. Sort T = {x_1 ,x_2 , …, x_n } by decreasing sizes ℓ_i .
  3. Put each new object into the used bin where the remaining capacity will be minimal, if there is such bin; otherwise, use a new bin.
- Algorithm

Algorithm BFD.
instance: $T = \{x_1, x_2, \ldots, x_n\}$; $\ell_i \in (0,1], i = 1, 2, \ldots, n$
output: $\pi_{BFD} : T \to \{1, 2, \ldots, \mu_{BFD}\}$, where $\text{load}(j) = \sum_{\pi_{BFD}(x_i)=j} \ell_i \leq 1$ for $j = 1, \ldots, \mu_{BFD}$

begin
  sort T into $(x_{i_1}, x_{i_2}, \ldots, x_{i_n})$ such that $\ell_{i_1} \geqslant \ell_{i_2} \geqslant \ldots \geqslant \ell_{i_n}$;
  $\mu_{BFD} := 1$; // at least one bin is needed
  for $i := i_1, i_2, \ldots, i_n$ **do**
    if $\exists j \in \{1, 2, \ldots, \mu_{BFD}\} : \text{load}(j) + \ell_i \leq 1$ // some used bin j can take x_i
      then $\pi_{BFD}(x_i) := j$ which minimizes $1 - \text{load}(j) - \ell_i$ // allocate x_i to a fullest one
      else
        begin // introduce a new empty bin
          $\mu_{BFD} := \mu_{BFD} + 1$; // remember it is currently last
          $\pi_{BFD}(x_i) := \mu_{BFD}$ // allocate x_i to it
        end
  endfor;
  return π_{BFD}
end.

(cont'd)

- Time complexity of BFD

  BFD is a poly-time algorithm (sorting takes $\mathcal{O}(n \log n)$ time and the rest takes $\mathcal{O}(n^2)$ time).

- Quality of suboptimal solutions π_{BFD}

  Theorem. $m(\pi_{BFD}) \leqslant \frac{11}{9} m(\pi^*) + 4$ (Proof. See booklet. □)

  Intuitively. BFD returns allocation π_{BFD} that uses $\leqslant$ 22% more bins than necessary.

  Comment. The upper bound is tight.

  Theorem. $m(\pi_{BFD}) \leqslant m(\pi_{FFD})$ (Proof. See booklet. □)

  Comment. Therefore, BFD is not worse than FFD. However, we know that there exist instances for which $m(\pi_{BFD}) < m(\pi_{FFD})$.

Dynamic Programming

- Sometimes we can compute optimal solution of an instance of a given computational problem if we know optimal solutions of smaller instances.
- The method of dynamic programming starts with solutions to trivial instances and then progressively computes optimal solutions of ever larger instances from optimal solutions of smaller instances.

PROBLEM: KNAPSACK (revisited)

We dealt with the KNAPSACK problem when we used the greedy method.
Now we will use the method of dynamic programming to solve this problem.

- **Definition**. Fill a knapsack of given load-capacity with most valuable subset of a given set of objects. Formally:
  - Instance of the problem:
    - $X = \{x_1,\ldots,x_n\}$ … set of n objects
    - $a_i \in \mathbf{N}$ … weights of objects i=1,2,…,n
    - $p_i \in \mathbf{N}$ … values of objects i=1,2,…,n
    - $b \in \mathbf{N}$ … load-capacity
  - Feasible solution: every $Y \subseteq X$ such that $\sum_{x_i \in Y} a_i \leqslant b$ … every subset Y of X not heavier than b
  - Quality of a feasible solution Y: $m(Y) = \sum_{x_i \in Y} p_i$ … value of Y
  - Goal: Find a feasible solution Y* which maximizes m(Y).

- **Our design strategy**.
  1. Design an exact exp-time algorithm E for KNAPSACK using dynamic programming.
  2. Transform the algorithm E into an approximation algorithm A for KNAPSACK.

Algorithm E (Exact algorithm)

First, we design an exact, pseudo-polynomial algorithm for the KNAPSACK problem.

- Idea.
  - Let p ∈ $[0, \sum_{1}^{n} p_i]$ … a possible value (i.e. neither negative nor too large)
  - Define X_i := $\{x_1, x_2, \ldots x_i\}$ … the set of the first i objects
  - Every subset of X_i has some weight and some value. Let us focus on the subsets of X_i whose weights are $\leqslant b$ and values are =p (subsets of Xi that are not too heavy and value exactly p). Sometimes there are no such subsets in X_i . But, if there are, then (at least) one of them has the smallest weight (is the lightest); in this case, let us denote this set by S(i,p). In short, we define:

$$S(i,p) = \begin{cases} \text{the lightest subset of } X_i, \\ \text{which weights} \leqslant b \text{ and values} = p, & \textit{if such a subset exists;} \\ \\ \uparrow \text{(undefined)}, & \textit{otherwise.} \end{cases}$$

  - Use of S(i,p). Suppose that we computed, in succession, the sets

    S(n, $\sum_{1}^{n} p_i$), S(n, $\sum_{1}^{n} p_i$ - 1), S(n, $\sum_{1}^{n} p_i$ - 2),… until we found the first S(n,p*) which is defined.

    So, p* would be the largest value for which there exists a subset of X_n (all objects) that is not heavier than b. Thus, S(n,p*) would be the solution to the KNAPSACK problem!

(cont'd)

- **Question**. How to compute the sets S(i,p)?
- **Answer**. We can use the method of dynamic programming:
  - For all p ∈ $[0, \sum_{1}^{n} p_i]$, compute the (trivial) sets S(1,p) and their weights A (1,p)
  - Find out how to compute "larger" sets S from "smaller" ones.
  - Using this, compute "largest" sets S(n, p) and return a defined set with maximal p.
- Details.
  - **i = 1**. Now $X_1 = \{x_1\}$. The subsets of X_1 are ∅ and $\{x_1\}$.
    The weight and value of ∅ are 0. The weight of $\{x_1\}$ is a_1 and its value is p_1. Therefore, we obtain:

    $S(1,0) = \emptyset$ and $A(1,0) = 0$

    $S(1,p_1) = \{x_1\}$ and $A(1, p_1) = a_1$

    $S(1,p) = \uparrow$ and $A(1, p) = \uparrow$, for all $p \in [0, \sum_{1}^{n} p_i]$ except 0 and p_1
  - **i ⩾ 2** Consider object x_i. There are two possibilities: either $x_i \in S(i,p)$ or $x_i \notin S(i,p)$. We analyze each.
    - If $x_i \in S(i,p)$ then $S(i,p) = \{x_i\} \cup S(i-1,p-p_i)$ and $A(i,p) = a_i + A(i-1,p-p_i)$
      assuming that $p-p_i \geqslant 0 \wedge S(i-1,p-p_i)\downarrow \wedge a_i + A(i-1,p-p_i) \leqslant b$
    - If $x_i \notin S(i,p)$ then $S(i,p) = S(i-1,p)$ and $A(i,p) = A(i-1,p)$
    - By definition, S(i,p) must be the lighter of the two possibilities; therefore, the weight of S(i,p) is

      $A(i,p) = \min \{ A(i-1,p), a_i + A(i-1,p-p_i) \}$
- The computation of the sets S(i,p) will run in the bottom-up fashion (increasing i, increasing p).

(cont'd)

- Algorithm

Algorithm E.
instance: X, n, p_i, a_i, b
output: Y^*
begin
  for $p := 0$ **to** $\sum_{k=1}^{n} p_k$ **do** // if $i = 1$
    $S(1,p) := \uparrow$;
    $A(1,p) := 1 + \sum_{j=1}^{n} a_j$
  endfor;
  $S(1,0) := \emptyset$; $A(1,0) := 0$;
  $S(1,p_1) := \{x_1\}$; $A(1,p_1) := a_1$;
  for $i := 2$ **to** n **do** // if $i = 2, 3, \ldots, n$
    for $p := 0$ **to** $\sum_{j=1}^{n} p_j$ **do**
      if $p - p_i \geq 0 \wedge S(i-1, p-p_i)\downarrow \wedge$
        $\wedge\, a_i + A(i-1,p-p_i) \leq b \wedge a_i + A(i-1,p-p_i) \leq A(i-1,p)$
        then
          $S(i,p) := \{x_i\} \cup S(i-1,p-p_i)$; $A(i,p) := a_i + A(i-1,p-p_i)$
        else
          $S(i,p) := S(i-1,p)$; $A(i,p) := A(i-1,p)$
    endfor
  endfor;
  $p^* :=$ largest p for which $S(n,p)\downarrow$;
  $Y^* := S(n,p^*)$;
  return Y^*
end.

(cont'd)

- Computational complexity of algorithm E

  It seems that the time complexity of the algorithm E is $\Theta(n\sum_{1}^{n} p_i)$ (due to the double loop). But recall that the complexity is (by definition) a function of the sizes of the arguments, (space required by the arguments), and not a function of the magnitudes of the arguments.

  In $\Theta(n\sum_{1}^{n} p_i)$, p_i represents the *magnitude* of the value of the object x_i, *not* the space required to store the object's value of this magnitude. The space required to store the object's value of magnitude p_i is $\log(p_i)$. Similarly for the sum $\sum_{1}^{n} p_i$ of magnitudes.

  So, $\Theta(n\sum_{1}^{n} p_i)$ is a polynomial function in the magnitude of the arguments p_i . However, it is an exponential function in the size of the arguments p_i .

  We say that algorithm E has a pseudo-polynomial time complexity.

(cont'd)

Algorithm A (Approximation algorithm)

Based on the algorithm E, we now design an approximation algorithm for the KNAPSACK..

- Idea.
  - Exponential complexity of the algorithm E stems in the loop **for** p:=0 **to** $\sum_1^n p_i$ **do** because $\sum_1^n p_i$ is exponential function of the size of the input data p_1, p_2,…, p_n .
  - Let us replace each p_i by substantially smaller value p_i' (i.e., $p_i' \ll p_i$). Then also $\sum_1^n p_i$ turns into a substantially smaller $\sum_1^n p_i'$ (i.e., $\sum_1^n p_i' \ll \sum_1^n p_i$), thus speeding up E.
  - In particular, replacing each p_i by $p_i' = p_i/2^c$, for some constant $c \in \mathbf{N}$, the execution time of algorithm E will reduce by factor 2^c.
- Note that instead of exact solving the instances of the form $I = (X,n,p_1,p_2,\ldots,p_n,a_1,a_2,\ldots,a_n,b)$,
  algorithm E now computes exact solutions of some other instances, the instances of the form $I' = (X,n,p'_1,p'_2,\ldots,p'_n,a_1,a_2,\ldots,a_n,b)$.
- Are the solutions of instances I and I' related?
  Yes, the exact solution Y*(I') of I' is at the same time approximate solution I, because
  - $Y^*(I') \subseteq X$
  - $\sum_{x_j \in Y^*(I')} a_j \leq b.$

(cont'd)

- We can now write the approximation algorithm A for solving instances I of KNAPSACK.

Algorithm

Algorithm A.
instance: $I = \{X, n, p_i, a_i, b\}$, constant $c \in \mathbb{N}$;
output: suboptimal solution $Y(I)$ of the instance I
begin
for $i := 1$ **to** n **do** $p'_i := \lfloor p_i / 2^c \rfloor$;
//let $I' = \{X, n, p'_i, a_i, b\}$ be the instance associated to I
$Y^*(I') :=$ exact solution of I' obtained by algorithm E;
$Y(I) := Y^*(I')$;
return $Y(I)$
end.

(cont'd)

- Quality of suboptimal solutions Y(I)

  What is the performance ratio?

  Theorem. $\frac{m(Y^*(I))}{m(Y(I))} \leq r$, where $r = \frac{p_{max}}{p_{max} - n2^c}$ and $p_{max} = \max_i p_i$.

  Intuitively. Algorithm A returns a solution Y(I) of I which is at most r-times smaller than the optimal solution Y*(I). Note also that r depends on the instance I and constant c.

- When we fix the constant c, the choice of c also effects the ratio r (see above), and hence the quality of the approximate solution Y(I).
- Alternatively, we can choose the desired r (performance ratio), and then determine the associated constant c. We find (see booklet) that
- $$c = \left\lfloor \log\left(\frac{r-1}{r}\,\frac{p_{max}}{n}\right) \right\rfloor .$$
- Computational complexity of algorithm A

  Theorem. Algorithm A has time complexity $\mathcal{O}(n^3 \frac{r}{r-1})$.

- In fact, A is fully polynomial approximation scheme FPAS for the KNAPSACK problem.

RANDOMIZED SOLVING OF COMPUTATIONAL PROBLEMS

When we face an NP-complete or NP-hard computational problem we usually give up searching for an exact polynomial-time algorithm for the problem.

Instead of searching for an approximation algorithm, we may search for a heuristic algorithm that trades certainty of a solution for execution time. When such an algorithm exploits random numbers and returns, for any instance of the problem, a solution whose probability of error is bounded, we say that the algorithm is randomized.

PROBLEM: PRIMES

- **Definition**. Given a natural number n, decide whether or not n is prime. Formally:
  - Instance of the problem:
    - n … natural number
  - Solution: YES (if n is prime), NO (if n is composite)
- Clearly, PRIMES is a decision problem.

The Naïve algorithm

There is an obvious and intuitively appealing algorithm for the PRIMES problem.

- Idea. Check whether or not the number n is divisible by any of the numbers 2, 3, …, $\sqrt[2]{n}$.
  If it is, then n is composite; otherwise n is prime.
- Algorithm.

```
procedure NaivePRIMES(n) return YES/NO;
begin
  prime := YES;          //Provisional answer
  for i:=2 to ⌊√n⌋ do
    if i|n then
      prime := NO; exit
    endif
  endfor;
  return(prime)
end.
```

- Computational complexity of the naïve algorithm
  As the test i|n is done at most $\mathcal{O}(\sqrt[2]{n})$-times, it seems that time complexity is $\mathcal{O}(\sqrt[2]{n})$.
  However, the size of the input n is size(n) = $\lceil \log_2 n \rceil$ and not n (magnitude). So, the seeming time complexity $\mathcal{O}(\sqrt[2]{n})$, when expressed by size(n), is in fact exponential:

$$\sqrt[2]{n} = \sqrt[2]{2^{\log_2 n}} \leqslant \sqrt[2]{2^{\text{size}(n)}} = 2^{\frac{1}{2}\text{size}(n)} = \Theta(2^{\frac{1}{2}\text{size}(n)})$$

The AKS algorithm

Question. Can the problem PRIMES be deterministically solved in polynomial time? After several decades of fruitless attempts of many scientists, in 2002, Agrawal, Kayal, and Saxena answered the question positively; they discovered a deterministic polynomial time algorithm, called the AKS algorithm, and proved its time complexity to be $\widetilde{\mathcal{O}}(\log(n)^{12})$. Other researchers improved this loose upper bound to $\widetilde{\mathcal{O}}(\log(n)^{10.5})$ and then to $\widetilde{\mathcal{O}}(\log(n)^{7.5})$ A 2005 variant of AKS algorithm has time complexity $\widetilde{\mathcal{O}}(\log(n)^{6})$. Other attempts to find even better (tighter) upper bound for AKS algorithm and its versions are well under way.

The algorithm is of immense theoretical importance, but it is not useful in practice.

Until the discovery of the AKS algorithm, the PRIMES was believed to be unsolvable in polynomial time. For this reason, scientists developed other approaches to primality testing that are still of great practical use. We will describe the Rabin's algorithm, ana algorithm that is founded on random number selection.

The Rabin's algorithm

- We begin with a well-known theorem.

  Theorem. (Fermat little theorem) If *n* is prime number, then for any integer *w*, the number $w^n - w$ is an integer multiple of *n*. (w^n-w=0(mod n), i.e. w^n=w(mod n), i.e. $w^{n-1} = 1 \pmod n$).

  Focusing on $w \in \{1,2,\ldots,n-1\}$, the theorem reads as follows:

  $$n \text{ is prime} \Rightarrow \forall w \in \{1,2,\ldots,n-1\}: w^{n-1} = 1 \pmod n$$

- The equivalent statement is (as $A \Rightarrow B$ is equivalent to $\neg A \Leftarrow \neg B$, and $\neg\forall x P(x)$ to $\exists x \neg P(x)$)

  $$n \text{ is composite} \Leftarrow \exists w \in \{1,2,\ldots,n-1\}: w^{n-1} \not\equiv 1 \pmod n$$

- **Definition**. Let $F(w,n) := w^{n-1} \not\equiv 1 \pmod n$. A $w \in \mathbf{N}$, such that F(w,n) is true, is called a Fermat's witness to the compositeness of n.

- If a Fermat witness (to the compositeness of n) exists then n is surely composite. The following algorithm (next slide) checks all potential witnesses w = 2,3,… n-1.

(cont'd)

```
procedure Composite(n) return YES/NO;
begin
  composite := NO; //Provisional ans
  for w:=2 to n-1 do
    if F(w,n) then
      composite := YES; exit
    endif
  endfor;
  return(composite)
end.
```

- The procedure Composite(n) returns YES if it has found a Fermat's witness for n, thus we can be sure that YES is a reliable answer.
- What if the procedure returns NO? This happens if it didn't find any Fermat witness among the values 2,3,…,n-1. Does that mean that n is composite? No!
- Why? The reason is that there exist composite numbers which have no Fermat's witnesses! Such numbers are called the Carmichael's numbers.

  Examples. The numbers 561, 1105, 1729, 2465 are Carmichael's numbers. Since 1994 we know there are infinitely many Carmichael's numbers. In fact, it was proved that there are about $n^{2/7}$ Carmichael's numbers between 1 and large n.

- **Summary.** The answer YES (n is composite) can be trusted, i.e. YES is always correct. The answer NO (n isn't composite, it is prime) cannot be trusted , i.e. NO may be false (as n may be prime or Carmichael's number). Composite(n) returns a true$_{YES}$, true$_{NO}$ or false$_{NO}$.

(cont'd)

- Idea.
  - Replace F(w,n) with some sharper predicate R(w,n), so that the following will hold:

$$n \text{ is composite} \iff \exists w \in \{1,2,\ldots,n-1\}: R(w,n)$$

  - If we found such an R(w,n) then Carmichael's numbers would no longer trouble us. If we replaced F(w,n) in the procedure Composite(n) with R(w,n), both YES and NO would be reliable answers. The corrected procedure NewComposite(n) would be

```
procedure NewComposite(n) return YES/NO;
begin
  composite := NO; //Provisional ans
  for w:=2 to n-1 do
    if R(w,n) then
      composite := YES; exit
    endif
  endfor;
  return(composite)
end.
```

(cont'd)

- The hypothetical predicate R(w,n) should have two properties:
  - It should be efficiently computable. (Its value true/false should be computable in polynomial time)
  - If n is composite, then there should be "sufficiently many" R-witnesses to n's compositeness. (Why? How many? We will answer this shortly.)

- But, does such an R(w,n) exist? Yes!

Theorem. Let k,m ∈ **N**, where m is odd, satisfying the equation n-1 = $m2^k$. Then

$$R(w,n) = w^m \not\equiv \pm 1 \pmod{n} \ \wedge \ \forall i \in \{1,2,\ldots,k-1\}: w^{m2^i} \not\equiv -1 \pmod{n}$$

Definition. A w ∈ **N**, such that R(w,n) is true, is called a Riemann's witness to the compositeness of n.

- It was found that
  - R(w,n) is computable in polynomial time.
  - If n is composite, then there exist at least $\frac{1}{2}n$ (F V R)-witnesses for the compositeness of n.
  - Miller proved, that there is a predicate W(w,n) (= F(w,n) V R(w,a) V A(w,n), for some predicate A(w,n)) such that if n is composite, then there are at least $\frac{3}{4}n$ W-witnesses for the compositeness of n. He found that

$$W(w,n) = w^{n-1} \not\equiv 1 (\mathrm{mod}\ n) \ \vee \ \exists i \left(2^i | (n-1) \ \wedge \ 1 < \gcd(w^{(n-1)/2^i} - 1, n) < n\right)$$

(cont'd)

- How did Rabin use these ideas to design a primality test? :
  - If he applied the procedure NewComposite(n) --- with R(w,n) replaced by F(w,r) V R(w,n) to have at least $\frac{1}{2}n$ witnesses for a composite n, --- both YES and NO would be correct answers.
  - But in case of a prime n, the procedure would check in succession all values w = 2,3,…,n-1, and require n-2 = Θ(n) time, which is in fact exponential (recall: $n = 2^{\log_2 n} = \mathcal{O}(2^{\text{size}(n)})$).

  - Instead of systematic checking all w = 2,3,…,n-1, Rabin applied checking randomly picked w's from {2,3,…,n-1}. A single random selection and checking is performed by the following procedure:

```
procedure RandomlyPick&Check(n) return YES/NO;
begin
  composite := NO;              //Provisional answer
  w := Randomly_select(1,n);    //Randomly pick w from {1,2,...,n}
  if R(w,n) V F(w,n)            //Check the value of predicate R V F
    then composite := YES       //w is (R V F)-witness
  endif;
  return(composite)
end.
```

  - The answer YES is reliable (it is trueYES), since a witness has been picked.
    The answer NO is not reliable (it is trueNO or falseNO), since
    - either witnesses exist but random selection has missed all of them (so n is composite)
    - or there are no witnesses (so n is prime)

(cont'd)

- Now suppose that we run RandomlyPick&Check(n) r times (e.g. r = 100) as follows:
- In each run, w is randomly picked and checked and an answer YES/NO is returned.
- **Question**. What is the probability that falseNO is obtained in each of r runs?
  - A falseNO is obtained when there are witnesses but random selection of w missed all of them. Since there are at least $\frac{1}{2}n$ (FVR)-witnesses, the probability of missing all of them is < $\frac{1}{2}$.
  - So the probability of missing the witnesses r-times in succession is $P(r) < (\frac{1}{2})^r$.

```
procedure RabinPrimalityTest(n,r) return (YES/NO, real);
begin
  i:=1;
  repeat
    composite := NO;
    w := RandomlyPick&Check(1,n);
    if R(w,n) V F(w,n)
      then composite := YES
    endif;
    i++
  until (composite=YES or i=r);
  if composite=YES
    then return (YES, 1)     //n is composite with probability 1
    else return (NO, 1-P(r)  //n is prime with probability 1-P(r)
  endif
end.
```

END