LECTURE 5

Greedy Algorithms

- Interval Scheduling
- Interval Partitioning

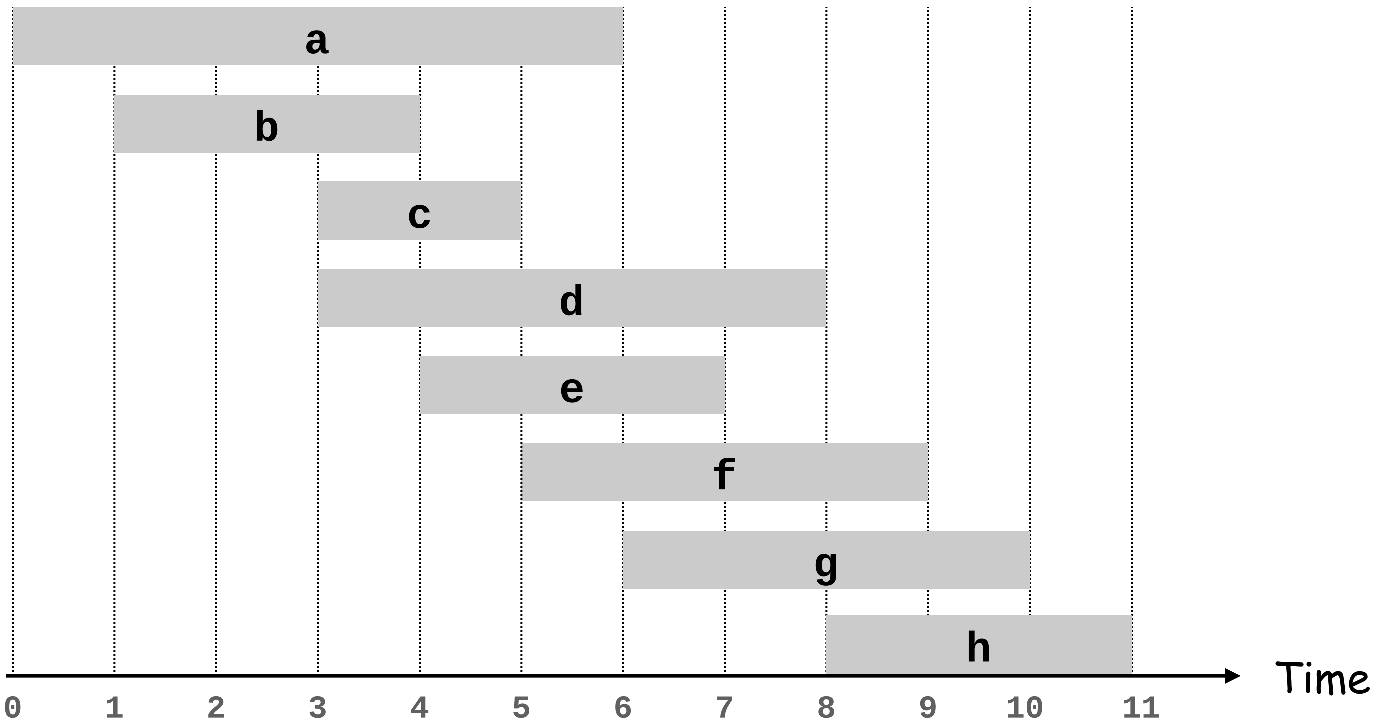
NP Completeness

Greedy Algorithms

- Build up a solution to an optimization problem at each step shortsightedly choosing the option that currently seems the best.
  - Sometimes good
  - Often does not work

Interval Scheduling Problem

- Job j starts at s_j and finishes at f_j.
- Two jobs are **compatible** if they do not overlap.
- **Find**: maximum subset of mutually compatible jobs.



Possible Greedy Strategies

Consider jobs in some natural order. Take next job if it is compatible with the ones already taken.

- **Earliest start time:** ascending order of s_j.
- **Earliest finish time:** ascending order of f_j.
- **Shortest interval:** ascending order of $(f_j – s_j)$.
- **Fewest conflicts:** For each job j, count the number of conflicting jobs c_j. Schedule in ascending order of c_j.

Greedy: Counterexamples

for earliest start time

for shortest interval

for fewest conflicts

Formulating Algorithm

- Arrays of start and finishing times
  - $s_1, s_2, \ldots, s_n$
  - $f_1, f_2, \ldots, f_n$

- Input length?
  - $2n = \Theta(n)$

Greedy Algorithm

- **Earliest finish time:** ascending order of f_j.

```
Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.

A ← ϕ    △ Set of selected jobs
for j = 1 to n {
   if (job j compatible with A)
      A ← A ∪ {j}
}
return A
```

- Implementation. **O(n log n) time**; **O(n) space**.
  - Remember job j^* that was added last to A.
  - Job j is compatible with A if $s_j \geq f_{j^*}$.

Running time: O(n log n)

O(n log n)

```
Sort jobs by finish times so that
      f1 ≤ f2 ≤ ... ≤ fn.
```

O(1)

```
A ← (empty)   △ Queue of selected jobs
j* ← 0
```

$n \times O(1)$

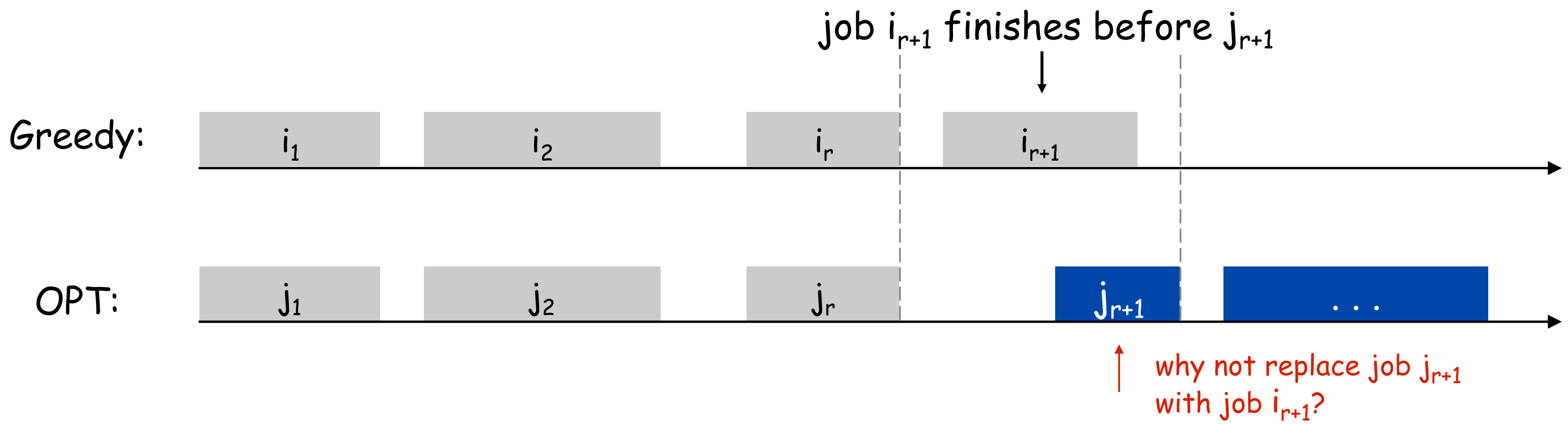
```
for j = 1 to n {
   if (fj* <= sj)
      enqueue(j onto A)
}
return A
```

Analysis: Greedy Stays Ahead

- **Theorem**. Greedy algorithm is optimal.
- **Proof strategy** (by contradiction):
  - –Suppose greedy is not optimal.
  - –Consider an optimal strategy… which one?
    - Consider the optimal strategy that agrees with the greedy strategy for as many initial jobs as possible
  - – Look at first place in list where optimal strategy differs from greedy strategy
  - – Show a new optimal strategy that agrees more w/ greedy

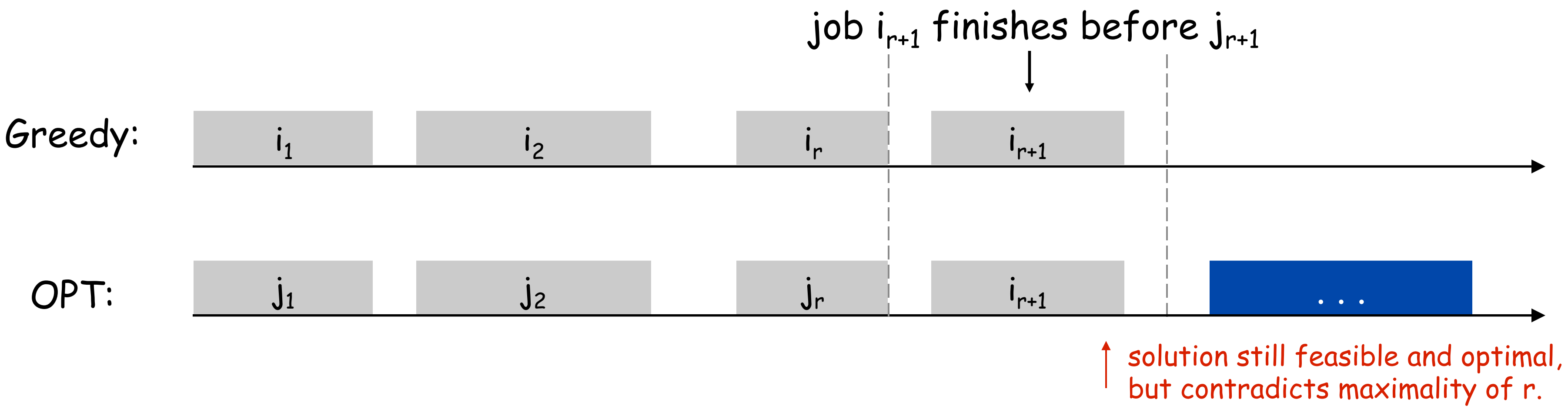
Analysis: Greedy Stays Ahead

- Theorem. Greedy algorithm is optimal.
- Pf (by contradiction): Suppose greedy is not optimal.
  - Let $i_1, i_2, \ldots i_k$ denote set of jobs selected by greedy.
  - Let $j_1, j_2, \ldots j_m$ be set of jobs in the optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ for the largest possible value of r.



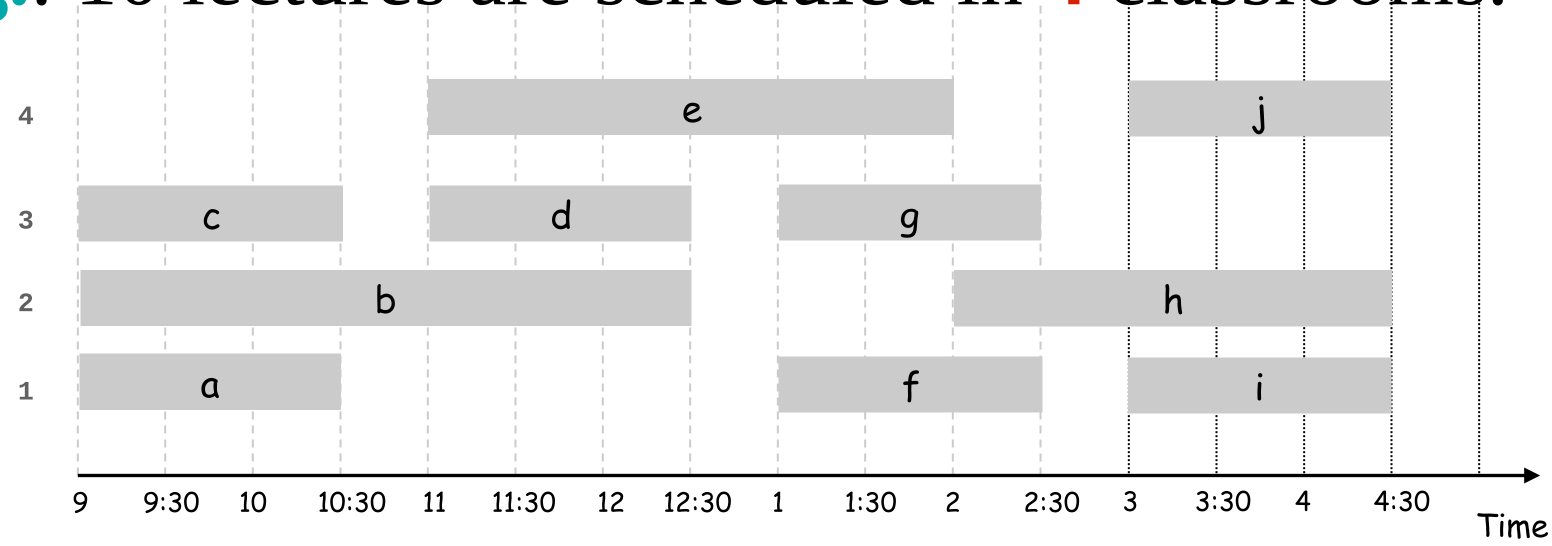
Analysis: Greedy Stays Ahead

- Theorem. Greedy algorithm is optimal.
- Pf (by contradiction): Suppose greedy is not optimal.
  - Let i_1, i_2, ... i_k denote set of jobs selected by greedy.
  - Let j_1, j_2, ... j_m be set of jobs in the optimal solution with $i_1 = j_1$, $i_2 = j_2$, ..., $i_r = j_r$ for the largest possible value of r.



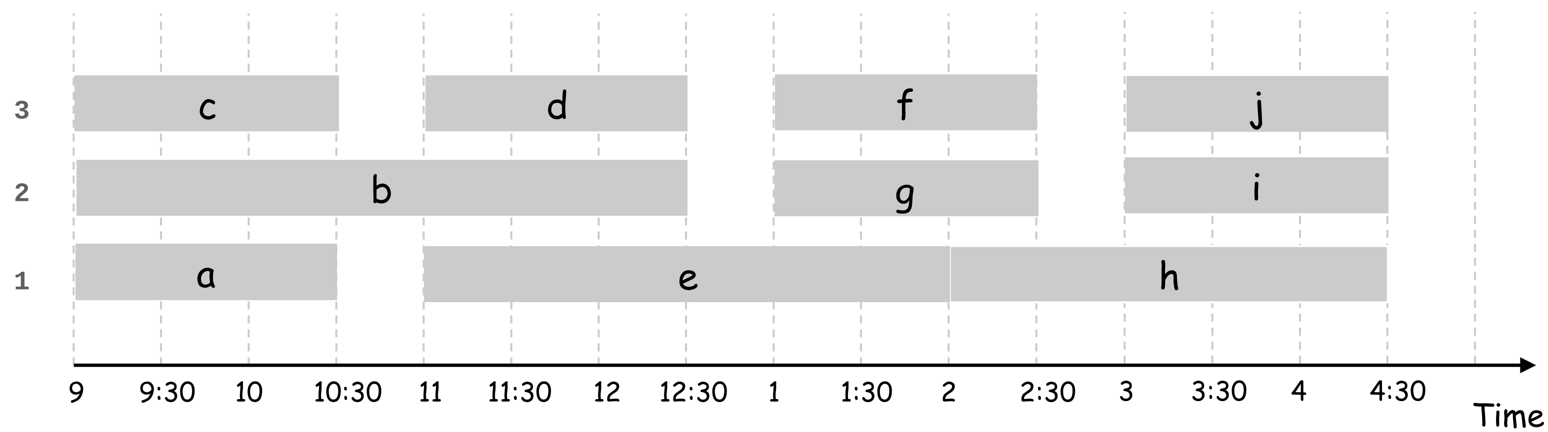
Interval Partitioning

- Lecture j starts at s_j and finishes at f_j.
- **Find**: minimum number of classrooms to schedule all lectures so that no two occur at the same time in the same room.
- **E.g.**: 10 lectures are scheduled in **4** classrooms.



Interval Partitioning

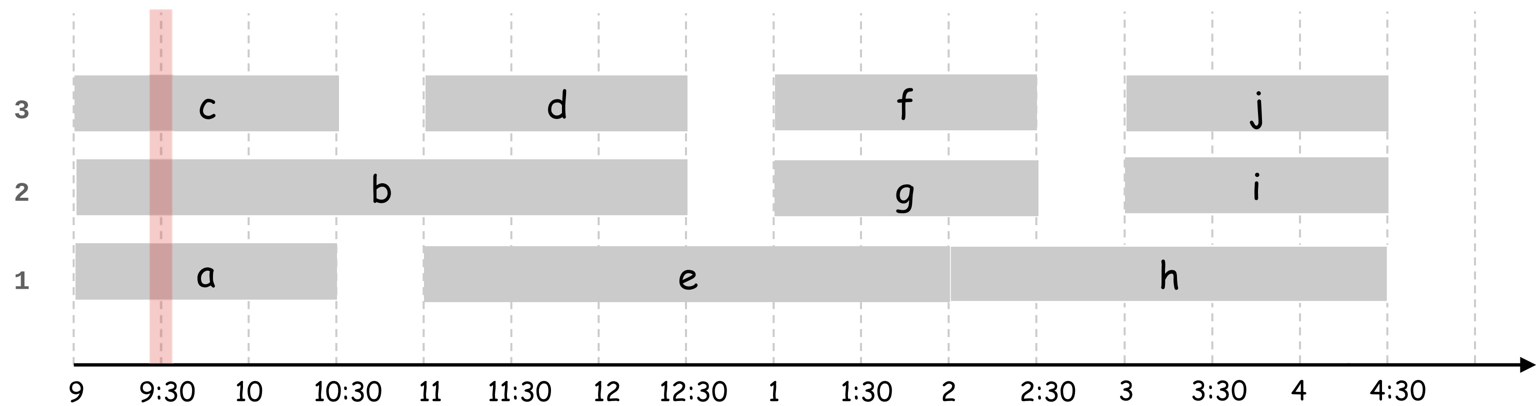
- Lecture j starts at s_j and finishes at f_j.
- **Find**: minimum number of classrooms to schedule all lectures so that no two occur at the same time in the same room.
- **E.g.**: Same lectures are scheduled in **3** classrooms.



Lower Bound

- **Definition.** The **depth** of a set of open intervals is the maximum number that contain any given time.
- **Key observation.** Number of classrooms needed $\geq$ depth.
- **E.g.:** Depth of this schedule = 3 $\Rightarrow$ this schedule is optimal.

a, b, c all contain 9:30



- **Q:** Is it always sufficient to have number of classrooms = depth?

Greedy Algorithm

- Consider lectures in increasing order of start time: assign lecture to any compatible classroom.

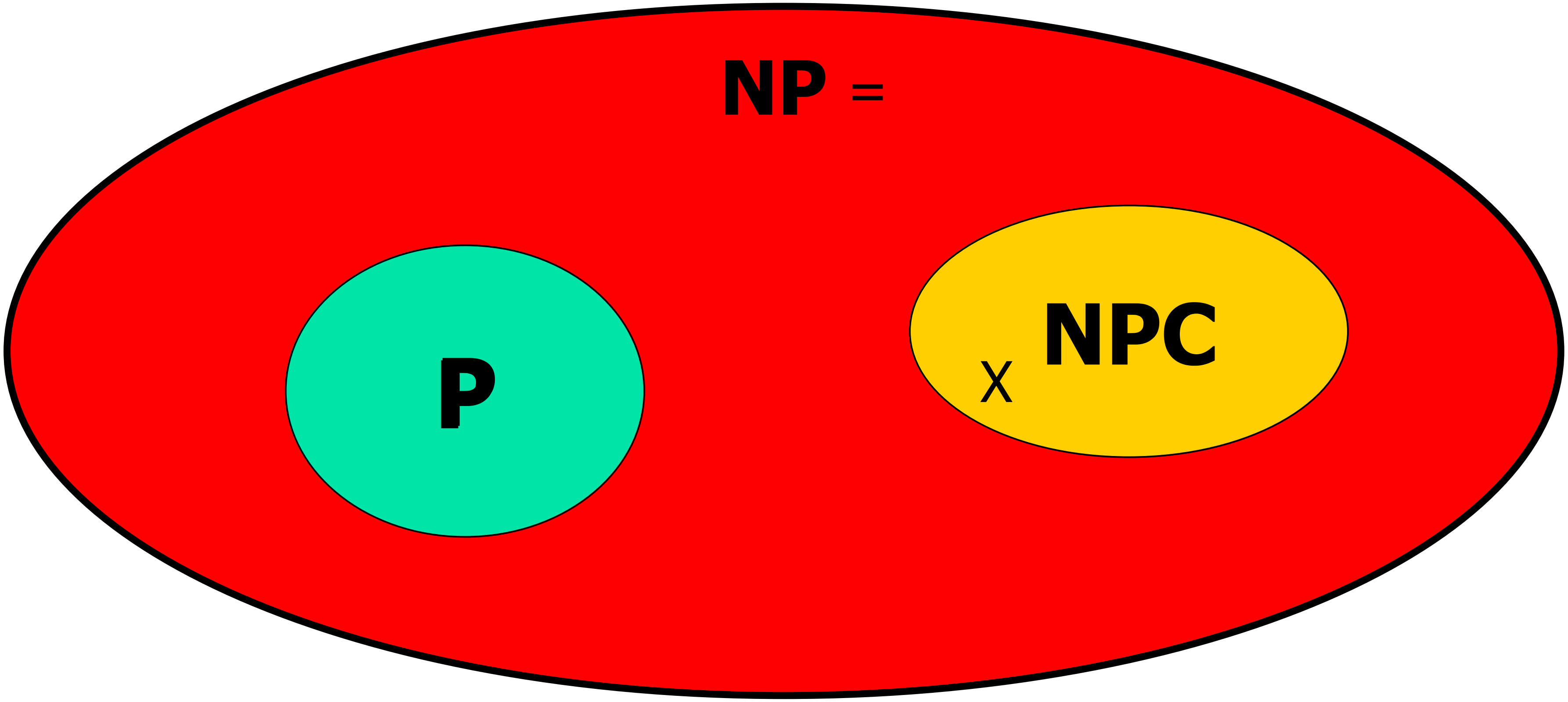
```
Sort intervals by starting time so that s1 ≤ s2 ≤ ... ≤ sn.
d ← 0   △ Number of allocated classrooms
for j = 1 to n {
   if (lecture j is compatible with some classroom k)
      schedule lecture j in classroom k
   else
      allocate a new classroom d + 1
      schedule lecture j in classroom d + 1
      d ← d + 1
}
```

- Implementation. $O(n \log n)$ time; $O(n)$ space.
  – For each classroom, maintain the finish time of the last job added.
  – Keep the classrooms in a priority queue (main loop $n \log(d)$ time)

Analysis: Structural Argument

- **Observation**. Greedy algorithm never schedules two incompatible lectures in the same classroom.
- **Theorem**. Greedy algorithm is optimal.
- **Proof:** Let d = number of classrooms allocated by greedy.
  - Classroom d is opened because we needed to schedule a lecture, say j, that is incompatible with all d-1 last lectures in other classrooms.
  - These d lectures each end after s_j.
  - Since we sorted by start time, they start no later than s_j.
  - Thus, we have d lectures overlapping at time $s_j + \varepsilon$.
  - Key observation $\Rightarrow$ all schedules use $\geq$ d classrooms. ▪

NP-Completeness

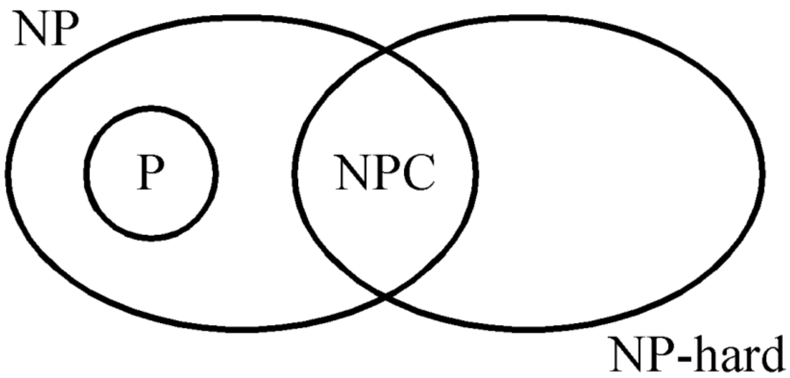


P=?NP

NP: Non-deterministic Polynomial

P: Polynomial

NPC: Non-deterministic Polynomial Complete



- **P**: the class of problems which can be solved by a deterministic polynomial algorithm.
- **NP** : the class of decision problem which can be solved by a non-deterministic polynomial algorithm.
- **NP-hard**: the class of problems to which every NP problem reduces.
- **NP-complete (NPC)**: the class of problems which are NP-hard and belong to NP.

Decision problems

- The solution is simply "Yes" or "No".
- Optimization problem : more difficult

  Decision problem
- E.g. the traveling salesperson problem
  - Optimization version:

    Find the shortest tour
  - Decision version:

    Is there a tour whose total length is less than or equal to a constant C ?

Nondeterministic algorithms

- **A nondeterministic algorithm is an algorithm consisting of two phases: guessing and checking.**
- **Furthermore, it is assumed that a nondeterministic algorithm always makes a correct guessing.**

Nondeterministic algorithms

- **They do not exist and they would never exist in reality.**
- **They are useful only because they will help us define a class of problems: NP problems**

NP algorithm

- If the checking stage of a nondeterministic algorithm is of polynomial time-complexity, then this algorithm is called an NP (nondeterministic polynomial) algorithm.

NP problem

- If a decision problem can be solved by a NP algorithm, this problem is called an NP (nondeterministic polynomial) problem.
- NP problems : (must be decision problems)

To express Nondeterministic Algorithm

- Choice(S) : arbitrarily chooses one of the elements in set S
- Failure : an unsuccessful completion
- Success : a successful completion

Nondeterministic searching Algorithm :

$j \leftarrow$ choice(1 : n) /* guess

if A(j) = x then success /* check

else failure

- A nondeterministic algorithm terminates unsuccessfully iff there exist no set of choices leading to a success signal.
- The time required for choice(1 : n) is O(1).
- A deterministic interpretation of a non-deterministic algorithm can be made by allowing unbounded parallelism in computation.

Problem Reduction

- Problem A reduces to problem B ($A \propto B$)
  - iff A can be solved by using any algorithm which solves B.
  - If $A \propto B$, B is more difficult (B is at least as hard as A)

$$\begin{tikzcd} \text{instance of A} \arrow[r, "\text{transformation}", "T(tr_1)"'] \arrow[d, "T(A)"'] & \text{instance of B} \arrow[d, "T(B)"', "\text{solver of B}"] \\ \text{answer of A} & \text{answer of B} \arrow[l, "T(tr_2)", "\text{transformation}"'] \end{tikzcd}$$

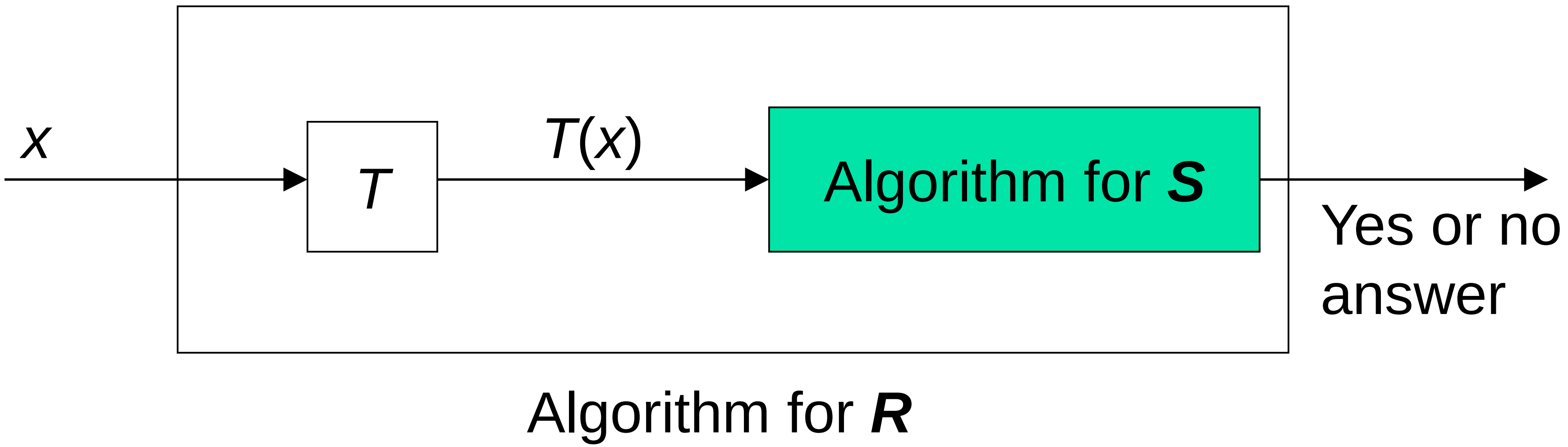
- Note: $T(tr_1) + T(tr_2) < T(B)$
- $T(A) \leq T(tr_1) + T(tr_2) + T(B) \sim O(T(B))$

Polynomial-time Reductions

- We want to solve a problem ***R***; we already have an algorithm for ***S***
- We have a transformation function T
  - Correct answer for ***R*** on x is "yes", iff the correct answer for ***S*** on $T(x)$ is "yes"
- Problem ***R*** is *polynomially reducible* to ***S*** if such a transformation T can be computed in polynomial time
- The point of reducibility: ***S*** is at least as hard to solve as ***R***

Polynomial-time Reductions

- We use *reductions* (or *transformations)* to prove that a problem is **NP**-complete



- *x* is an input for ***R***; *T*(*x*) is an input for ***S***
- (R $\propto$ S)

NPC and NP-hard

- A problem A is NP-hard if every NP problem reduces to A.
- A problem A is NP-complete (NPC) if $A \in NP$ and every NP problem reduces to A.
  - Or we can say a problem A is NPC if $A \in NP$ and A is NP-hard.

NP-Completeness

- "***NP**-complete problems*": the hardest problems in **NP**
- Interesting property
  - If any *one* **NP**-complete problem can be solved in polynomial time, then *every* problem in **NP** can also be solved similarly (i.e., P=NP)
- Many believe **P≠NP**

Importance of NP-Completeness

- NP-complete problems: considered "intractable"
- Important for algorithm designers & engineers
- Suppose you have a problem to solve
  - Your colleagues have spent a lot of time to solve it exactly but in vain
  - See whether you can prove that it is NP-complete
  - If yes, then spend your time developing an *approximation* (*heuristic*) *algorithm*
- Many natural problems can be NP-complete

Relationship Between **NP** and **P**

- It is not known whether **P=NP** or whether **P** is a proper subset of **NP**
- It is believed **NP** is much larger than **P**
  - But no problem in **NP** has been proved as not in **P**
  - No known deterministic algorithms that are polynomially bounded for many problems in **NP**
  - So, "does **P** = **NP**?" is still an open question!

Cook's theorem (1971)

NP = P iff SAT $\in$ P

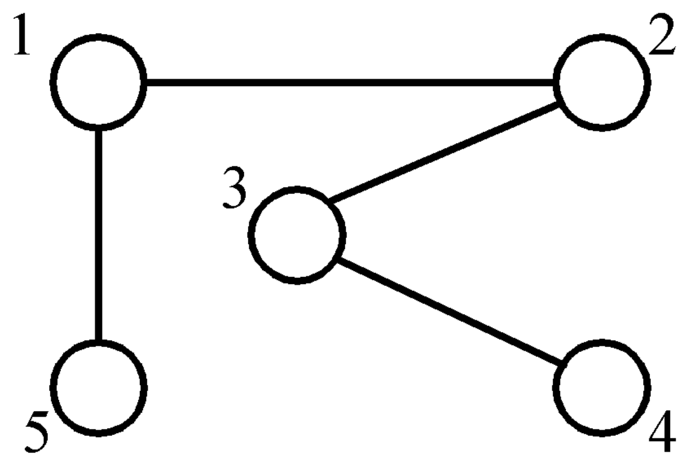
- SAT (the satisfiability problem) is NP-complete
- It is the first NP-complete problem
- Every NP problem reduces to SAT

SAT is NP-complete

- Every NP problem can be solved by an NP algorithm
- Every NP algorithm can be transformed in polynomial time to an SAT problem (a Boolean formula C)
- Such that the SAT problem is satisfiable iff the answer for the original NP problem is “yes”
- That is, every NP problem $\propto$ SAT
- SAT is NP-complete

The Node Cover Problem

- **Def**: Given a graph G = (V, E), S is the node cover of G if $S \subseteq V$ and for ever edge $(u, v) \in E$, (u,v) is incident to a node in S.



node cover:
{1, 3}
{5, 2, 4}

- Decision problem: $\exists\, S \ni |S| \le K$?

How to Prove a Problem *S* is NP-Complete?

1. Show ***S*** is in NP
2. Select a known NP-complete problem ***R***
   - Since ***R*** is NP-complete, all problems in NP are reducible to ***R***
3. Show how ***R*** can be poly. reducible to ***S***
   - Then all problems in NP can be poly. reducible to ***S*** (because polynomial reduction is transitive)
4. Therefore ***S*** is NP-complete

NPC Problems

- CLIQUE(k): Does G=(V,E) contain a clique of size $\geq k$?

Definition:

- A clique in a graph is a set of vertices such that any pair of vertices are joined by en edge.

NPC Problems

- Vertex Cover(k): Given a graph G=(V, E) and an integer *k*, does G have a vertex cover with $\leq k$ vertices?

Definition:

- A vertex cover of G=(V, E) is $V' \subseteq V$ such that every edge in E is incident to some $v \in V'$.

NPC Problems

- Dominating Set(k): Given an graph $G=(V, E)$ and an integer k, does G have a dominating set of size $\leq k$?

Definition:

- A dominating set D of $G=(V, E)$ is $D \subseteq V$ such that every $v \in V$ is either in D or adjacent to at least one vertex of D.

NPC Problems

- SAT: Give a Boolean expression (formula) in DNF (conjunctive normal form), determine if it is satisfiable.
- 3SAT: Give a Boolean expression in DNF such that each clause has *exactly* 3 variables (literals), determine if it is satisfiable.

NPC Problems

- Chromatic Coloring(k): Given a graph G=(V, E) and an integer *k*, does G have a coloring for *k*

Definition

- A coloring of a graph G=(V, E) is a function $f : V \rightarrow \{ 1, 2, 3, \ldots, k \} \ni$ if $(u, v) \in E$, then $f(u) \neq f(v)$.

Traveling salesperson problem

- Given: A set of n planar points

  Find: A closed tour which includes all points exactly once such that its total length is minimized.
- This problem is NP-complete.

Partition problem

- Given: A set of positive integers S
  Find: S_1 and S_2 such that $S_1 \cap S_2 = \varnothing$, $S_1 \cup S_2 = S$, $\sum_{i \in S1} i = \sum_{i \in S2} i$
  (partition into S_1 and S_2 such that the sum of S_1 is equal to S_2)
- e.g. S={1, 7, 10, 9, 5, 8, 3, 13}
  - S_1={1, 10, 9, 8}
  - S_2={7, 5, 3, 13}
- This problem is NP-complete.

Partition Problem

- **Def**: Given a set of positive numbers A = $\{ a_1, a_2, \ldots, a_n \}$, determine if $\exists$ a partition P, $\ni \sum_{i \in p} a_i = \sum_{i \notin p} a_i$
- e.g. A = {3, 6, 1, 9, 4, 11}
  partition: {3, 1, 9, 4} and {6, 11}

<Theorem> sum of subsets $\propto$ partition

Bin Packing Problem

- **Def**: n items, each of size c_i , $c_i > 0$, a positive number k and bin capacity C,

  determine if we can assign the items into k bins such that the sum of c_i's assigned to each bin does not exceed C.

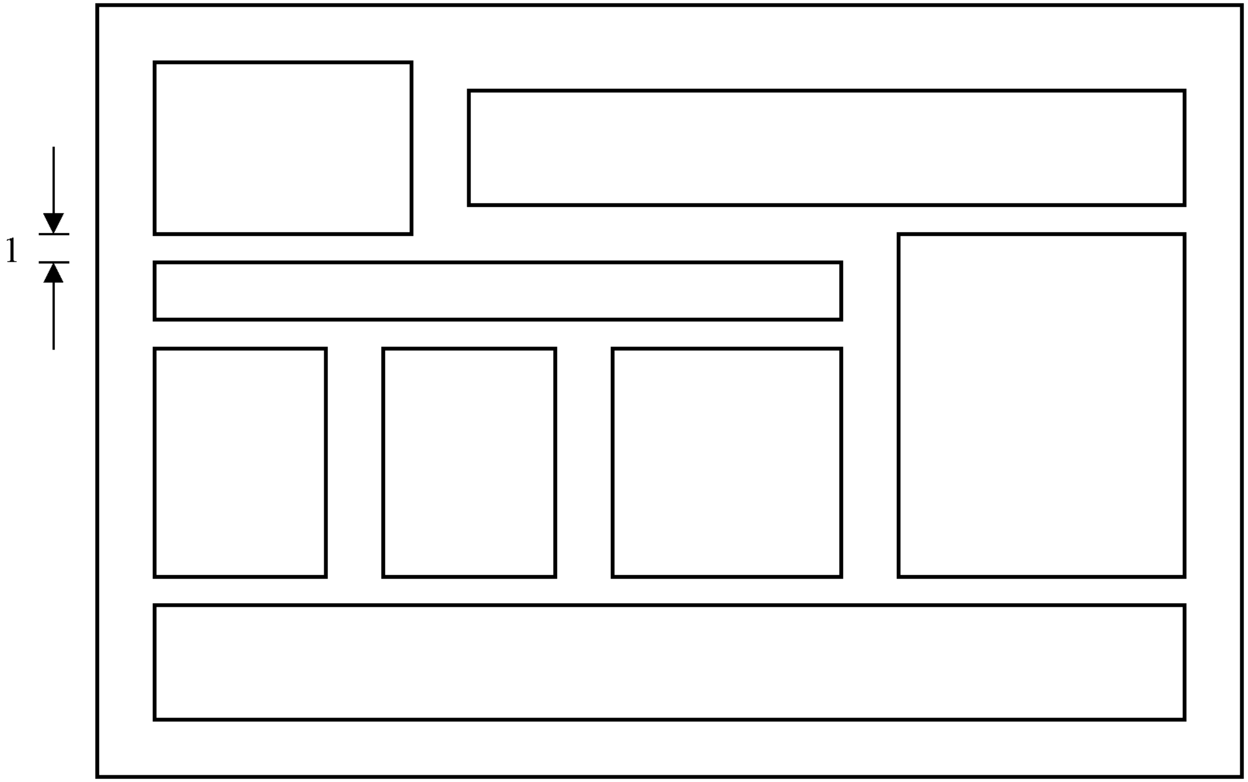
<Theorem> partition $\propto$ bin packing.

VLSI Discrete Layout Problem

- Given: n rectangles, each with height h_i (integer) width w_i and an area A, determine if there is a placement of the n rectangles within A according to the following rules:

1. Boundaries of rectangles are parallel to x axis or y axis.
2. Corners of rectangles lie on integer points.
3. No two rectangles overlap.
4. Two rectangles are separated by at least a unit distance.

(See the figure on the next page.)



A Successful Placement

<Theorem> bin packing $\propto$ VLSI discrete layout.