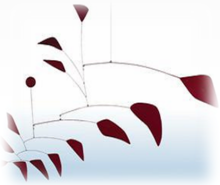


# EECS 3101 - Design and Analysis of Algorithms

**Shahin Kamali**

Topic 7 - Remarks on Complexity

# Overview

- A few remarks about complexity classes
- A brief introduction to NP-completeness
- $P \neq NP$ and other questions for which we do not know the answers!

# Polynomial Algorithms

- Most algorithms you have seen have running times $\Theta(\log n)$ (e.g., binary search), $\Theta(n)$ (e.g., searching in a linked list), $\Theta(n \log n)$ (e.g., merge-sot), $\Theta(n^2)$ (e.g., bubble-sort), $\Theta(n^3)$ (e.g., matrix multiplication), etc.
  - $n$ is the number of words (in word-RAM model) to encode the input.

# Polynomial Algorithms

- Most algorithms you have seen have running times $\Theta(\log n)$ (e.g., binary search), $\Theta(n)$ (e.g., searching in a linked list), $\Theta(n \log n)$ (e.g., merge-sot), $\Theta(n^2)$ (e.g., bubble-sort), $\Theta(n^3)$ (e.g., matrix multiplication), etc.
  - $n$ is the number of words (in word-RAM model) to encode the input.
- The running time of all these algorithms can be bounded by some polynomial function, e.g., $n^5$.

# Polynomial Algorithms

- Most algorithms you have seen have running times $\Theta(\log n)$ (e.g., binary search), $\Theta(n)$ (e.g., searching in a linked list), $\Theta(n \log n)$ (e.g., merge-sot), $\Theta(n^2)$ (e.g., bubble-sort), $\Theta(n^3)$ (e.g., matrix multiplication), etc.
  - $n$ is the number of words (in word-RAM model) to encode the input.
- The running time of all these algorithms can be bounded by some polynomial function, e.g., $n^5$.
- A **Polynomial Algorithm** has running time $O(n^c)$ on input size of $n$, where $c$ is a constant independent of $n$
  - E.g., $O(n), O(n^2), O(n^3), O(n^{2022})$.
  - Also $O(1), O(\alpha(n)), O(\log n), O(n \log n), O(\sqrt{n}), O(n^{3/2})$, etc.

# Polynomial Algorithms

- Most algorithms you have seen have running times $\Theta(\log n)$ (e.g., binary search), $\Theta(n)$ (e.g., searching in a linked list), $\Theta(n \log n)$ (e.g., merge-sot), $\Theta(n^2)$ (e.g., bubble-sort), $\Theta(n^3)$ (e.g., matrix multiplication), etc.
  - $n$ is the number of words (in word-RAM model) to encode the input.
- The running time of all these algorithms can be bounded by some polynomial function, e.g., $n^5$.
- A **Polynomial Algorithm** has running time $O(n^c)$ on input size of $n$, where $c$ is a constant independent of $n$
  - E.g., $O(n), O(n^2), O(n^3), O(n^{2022})$.
  - Also $O(1), O(\alpha(n)), O(\log n), O(n \log n), O(\sqrt{n}), O(n^{3/2})$, etc.
- A function is **super polynomial** if $f(n) \in \omega(n^c)$ for all $c$.

# Polynomial Algorithms

- Most algorithms you have seen have running times $\Theta(\log n)$ (e.g., binary search), $\Theta(n)$ (e.g., searching in a linked list), $\Theta(n \log n)$ (e.g., merge-sot), $\Theta(n^2)$ (e.g., bubble-sort), $\Theta(n^3)$ (e.g., matrix multiplication), etc.
  - $n$ is the number of words (in word-RAM model) to encode the input.
- The running time of all these algorithms can be bounded by some polynomial function, e.g., $n^5$.
- A **Polynomial Algorithm** has running time $O(n^c)$ on input size of $n$, where $c$ is a constant independent of $n$
  - E.g., $O(n), O(n^2), O(n^3), O(n^{2022})$.
  - Also $O(1), O(\alpha(n)), O(\log n), O(n \log n), O(\sqrt{n}), O(n^{3/2})$, etc.
- A function is **super polynomial** if $f(n) \in \omega(n^c)$ for all $c$.
  - E.g., $2^n$, $3^n$, $n!$, $n^n$, etc.

# Primality test

- Problem: given a positive integer $x$, indicate whether $x$ is prime.

# Primality test

- Problem: given a positive integer $x$, indicate whether $x$ is prime.
- Trial division: try to divide $x$ by $i$ for $i \in \{1, 2, \ldots, \sqrt{x}\}$.
  - What is the time complexity?

# Primality test

- Problem: given a positive integer $x$, indicate whether $x$ is prime.
- Trial division: try to divide $x$ by $i$ for $i \in \{1, 2, \dots, \sqrt{x}\}$.
  - What is the time complexity? $\rightarrow \Theta(\sqrt{x})$.

# Primality test

- Problem: given a positive integer $x$, indicate whether $x$ is prime.
- Trial division: try to divide $x$ by $i$ for $i \in \{1, 2, \ldots, \sqrt{x}\}$.
  - What is the time complexity? $\rightarrow \Theta(\sqrt{x})$.
  - Is this algorithm polynomial?

# Primality test

- Problem: given a positive integer $x$, indicate whether $x$ is prime.
- Trial division: try to divide $x$ by $i$ for $i \in \{1, 2, \dots, \sqrt{x}\}$.
  - What is the time complexity? $\rightarrow \Theta(\sqrt{x})$.
  - Is this algorithm polynomial? No (why?)
  - This algorithm is **pseudo-polynomial**; its running time is a polynomial in the numeric **value** of the input (the largest integer present in the input)—but not in the **length** of the input

# Primality test

- Problem: given a positive integer $x$, indicate whether $x$ is prime.
- Trial division: try to divide $x$ by $i$ for $i \in \{1, 2, \ldots, \sqrt{x}\}$.
  - What is the time complexity? $\rightarrow \Theta(\sqrt{x})$.
  - Is this algorithm polynomial? No (why?)
  - This algorithm is **pseudo-polynomial**; its running time is a polynomial in the numeric **value** of the input (the largest integer present in the input)—but not in the **length** of the input
- A rich field of research is formed around this 'simple' problem
  - Whether a polynomial time algorithm exists was not-known until 2002.

# Primality test

- Problem: given a positive integer $x$, indicate whether $x$ is prime.
- Trial division: try to divide $x$ by $i$ for $i \in \{1, 2, \ldots, \sqrt{x}\}$.
  - What is the time complexity? $\rightarrow \Theta(\sqrt{x})$.
  - Is this algorithm polynomial? No (why?)
  - This algorithm is **pseudo-polynomial**; its running time is a polynomial in the numeric **value** of the input (the largest integer present in the input)—but not in the **length** of the input
- A rich field of research is formed around this 'simple' problem
  - Whether a polynomial time algorithm exists was not-known until 2002.
  - Three scientists from India discovered an algorithm (later named AKS primality test) that runs in $\tilde{O}((\log x)^6) \in O((\log x)^7)$.
  - Their paper won the 2006 Godel prize among other things.


Manindra Agrawal


Neeraj Kayal


Nitin Saxena

# 3SUM problem

- the **3SUM problem** asks if a given set of $n$ real numbers contains three elements that sum to zero, e.g., for $S = \{-25, -10, -7, -3, 2, 4, 8, 10\}$, the answer is "true".
  - A naive solutions solves the problem in $\Theta(n^3)$.

# 3SUM problem

- the **3SUM problem** asks if a given set of $n$ real numbers contains three elements that sum to zero, e.g., for $S = \{-25, -10, -7, -3, 2, 4, 8, 10\}$, the answer is "true".
  - A naive solutions solves the problem in $\Theta(n^3)$.
  - A smart (but not complicated) algorithm exists that answers 3SUM in $\Theta(n^2)$.
- **3Sum-conjecture:** 3-Sum requires $\Omega(n^2)$ time, any algorithm for 3Sum runs in $\Omega(n^2)$.

# 3SUM problem

- the **3SUM problem** asks if a given set of $n$ real numbers contains three elements that sum to zero, e.g., for $S = \{-25, -10, -7, -3, 2, 4, 8, 10\}$, the answer is "true".
  - A naive solutions solves the problem in $\Theta(n^3)$.
  - A smart (but not complicated) algorithm exists that answers 3SUM in $\Theta(n^2)$.
- **3Sum-conjecture:** 3-Sum requires $\Omega(n^2)$ time, any algorithm for 3Sum runs in $\Omega(n^2)$.
  - This conjecture was open for a long time, until it was refuted in 2014 by an algorithm which runs in $O(n^2/(\log n \log\log n)^{2/3})$. [Gronlund and Pettie paper on "Threesomes, Degenerates, and Love Triangles"]

# 3SUM problem

- the **3SUM problem** asks if a given set of $n$ real numbers contains three elements that sum to zero, e.g., for $S = \{-25, -10, -7, -3, 2, 4, 8, 10\}$, the answer is "true".
  - A naive solutions solves the problem in $\Theta(n^3)$.
  - A smart (but not complicated) algorithm exists that answers 3SUM in $\Theta(n^2)$.
- **3Sum-conjecture:** 3-Sum requires $\Omega(n^2)$ time, any algorithm for 3Sum runs in $\Omega(n^2)$.
  - This conjecture was open for a long time, until it was refuted in 2014 by an algorithm which runs in $O(n^2/(\log n \log\log n)^{2/3})$. [Gronlund and Pettie paper on "Threesomes, Degenerates, and Love Triangles"]
  - Modern 3Sum-conjecture: 3-Sum requires $\Omega(n^{2-\epsilon})$ time for any constant $\epsilon > 0$.

# 3SUM problem

- the **3SUM problem** asks if a given set of $n$ real numbers contains three elements that sum to zero, e.g., for $S = \{-25, -10, -7, -3, 2, 4, 8, 10\}$, the answer is “true".
  - A naive solutions solves the problem in $\Theta(n^3)$.
  - A smart (but not complicated) algorithm exists that answers 3SUM in $\Theta(n^2)$.
- **3Sum-conjecture:** 3-Sum requires $\Omega(n^2)$ time, any algorithm for 3Sum runs in $\Omega(n^2)$.
  - This conjecture was open for a long time, until it was refuted in 2014 by an algorithm which runs in $O(n^2/(\log n \log\log n)^{2/3})$. [Gronlund and Pettie paper on “Threesomes, Degenerates, and Love Triangles”]
  - Modern 3Sum-conjecture: 3-Sum requires $\Omega(n^{2-\epsilon})$ time for any constant $\epsilon > 0$.
  - If this conjecture is true, many other **3SUM-hard** problems also requires $\Omega(n^{2-\epsilon})$.

# Exhaustive Search

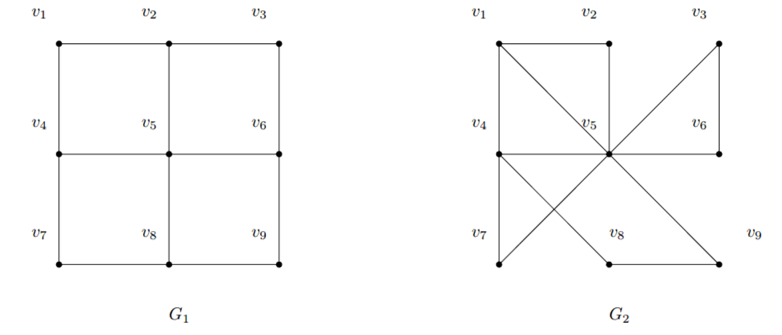
- Many problems have an exponential number of possible solutions.
- An algorithm which applies an exhaustive search on the solution space will eventually find a solution
- The time will be proportional to the size of solution space in the worst case, i.e., it will be super-polynomial.
  - This is not good!

## Exhaustive Search

- Many problems have an exponential number of possible solutions.
- An algorithm which applies an exhaustive search on the solution space will eventually find a solution
- The time will be proportional to the size of solution space in the worst case, i.e., it will be super-polynomial.
  - This is not good!
  - For many problems, we have failed to do much better.

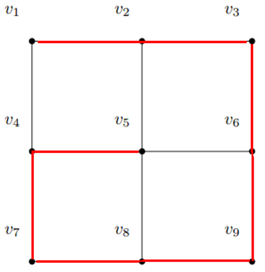
# Hamiltonian Path

- Instance: a graph $G$ with vertex set $V$ and edge set $E$.
- Question: Does there exist a path in $G$ that visits every vertex in $V(G)$ exactly once along a sequence of edges in $E(G)$?

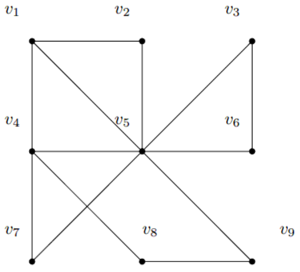


# Hamiltonian Path

- Instance: a graph $G$ with vertex set $V$ and edge set $E$.
- Question: Does there exist a path in $G$ that visits every vertex in $V(G)$ exactly once along a sequence of edges in $E(G)$?




$G_1$




$G_2$

# Exhaustive Search for HP

- Try all paths and check whether the sequence of edges exist in $G$
- In other words, try all permutations of vertices
  - $v_1, v_2, v_3, v_4, \ldots, v_n$
  - $v_2, v_1, v_3, v_4, \ldots, v_n$
  - $\ldots$

# Exhaustive Search for HP

- Try all paths and check whether the sequence of edges exist in $G$
- In other words, try all permutations of vertices
  - $v_1, v_2, v_3, v_4, \ldots, v_n$
  - $v_2, v_1, v_3, v_4, \ldots, v_n$
  - $\ldots$
- There are $n!$ different paths
  - Some paths are redundant, e.g., $v_1, v_2, \ldots, v_n$ is the same as $v_n, v_{n-1}, \ldots, v_1$.
  - Regardless, the number of distinct paths is still $\Theta(n!)$.

# Exhaustive Search for HP

- Try all paths and check whether the sequence of edges exist in $G$
- In other words, try all permutations of vertices
  - $v_1, v_2, v_3, v_4, \ldots, v_n$
  - $v_2, v_1, v_3, v_4, \ldots, v_n$
  - $\ldots$
- There are $n!$ different paths
  - Some paths are redundant, e.g., $v_1, v_2, \ldots, v_n$ is the same as $v_n, v_{n-1}, \ldots, v_1$.
  - Regardless, the number of distinct paths is still $\Theta(n!)$.
- $\rightarrow$ exhaustive search requires $\Omega(n!)$ in the worst case

# Complexity of HP

- There are 'faster' algorithms, e.g., $O(n^2 2^n)$ deterministic and $O(1.415^n)$ randomized algorithms.

# Complexity of HP

- There are 'faster' algorithms, e.g., $O(n^2 2^n)$ deterministic and $O(1.415^n)$ randomized algorithms.
- Is there a polynomial algorithm for Hamiltonian Path?
    - We don't know, but no such algorithm is discovered yet, and it is unlikely that we can find one!
    - This relates to $P \neq NP$ conjecture that we see in a minute.

# Complexity of HP

- There are 'faster' algorithms, e.g., $O(n^2 2^n)$ deterministic and $O(1.415^n)$ randomized algorithms.
- Is there a polynomial algorithm for Hamiltonian Path?
  - We don't know, but no such algorithm is discovered yet, and it is unlikely that we can find one!
  - This relates to $P \neq NP$ conjecture that we see in a minute.
- There are many **'Hard'** problems like Hamiltonian path problem for which we do not know whether a polynomial algorithm exists; they form a complexity class.

# Complexity of HP

- There are 'faster' algorithms, e.g., $O(n^2 2^n)$ deterministic and $O(1.415^n)$ randomized algorithms.
- Is there a polynomial algorithm for Hamiltonian Path?
  - We don't know, but no such algorithm is discovered yet, and it is unlikely that we can find one!
  - This relates to $P \neq NP$ conjecture that we see in a minute.
- There are many **'Hard'** problems like Hamiltonian path problem for which we do not know whether a polynomial algorithm exists; they form a complexity class.
  - If there is a polynomial algorithm for any of these problems, there will be polynomial algorithms for all of them.
  - When you fail to come up with a polynomial algorithm for a problem, investigate whether it is 'Hard'.

# Application of Reductions

- Assume you have a problem $P$ for which you look for an efficient, polynomial algorithm, and you fail after trying a bit.

# Application of Reductions

- Assume you have a problem $P$ for which you look for an efficient, polynomial algorithm, and you fail after trying a bit.
- How can you determine whether you should keep searching for an efficient algorithm or whether it's unlikely that any efficient algorithm for problem $P$ exists?

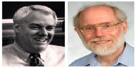
# Application of Reductions

- Assume you have a problem $P$ for which you look for an efficient, polynomial algorithm, and you fail after trying a bit.
- How can you determine whether you should keep searching for an efficient algorithm or whether it's unlikely that any efficient algorithm for problem $P$ exists?
- If you can reduce one of those Hard problems to $P$ in polynomial time, then a polynomial algorithm for $P$ gives polynomial algorithms for all those hard problems.

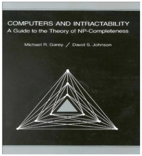
# Application of Reductions

- Since none of those Hard problems have any known polynomial algorithm, it is unlikely that you can come up with a polynomial algorithm for $P$.
    - Informally, to give up searching for a polynomial algorithm for $P$, it suffices to reduce a 'Hard' problem to $P$ in polynomial time.
    - We say the problem is **NP-Hard** in that case!
    - To show $P$ is NP-Hard, we reduce another NP-Hard problem to $P$

"I can't find an efficient algorithm, but neither can all these famous people."



Michael Garey David Johnson



# Complexity Classes

- A complexity class is a set of problems that can be solved with a similar amount of time/space/cost resources.

# Complexity Classes

- A complexity class is a set of problems that can be solved with a similar amount of time/space/cost resources.
- Important complexity classes: P, NP, EXP, R, etc.

# Complexity Classes

- A complexity class is a set of problems that can be solved with a similar amount of time/space/cost resources.
- Important complexity classes: P, NP, EXP, R, etc.
- P = problems that can be solved in polynomial time, i.e., $O(n^c)$ for some fixed $c$
  - E.g., given a graph on $n$ vertices and $m$ edges, find its MST; it can be done in $O(n^3)$.
  - Basically, all problems for which you have seen an algorithm belong to class $P$ of problems.

# Class NP

- A problem belongs to class *NP* if a non-deterministic Turing machine can solve it in polynomial time.

# Class NP

- A problem belongs to class *NP* if a non-deterministic Turing machine can solve it in polynomial time.
- These are problems whose solutions can be verified in polynomial time.
  - For decision problems, instances with a yes answer can be verified.

# Class NP

- A problem belongs to class *NP* if a non-deterministic Turing machine can solve it in polynomial time.
- These are problems whose solutions can be verified in polynomial time.
    - For decision problems, instances with a **yes** answer can be verified.
- E.g., Hamiltonian Path is an NP problem: given an instance of the problem we can verify if a solution gives a 'yes' answer in polynomial time.
    - Given a solution path, we can verify whether it is a Hamiltonian path, i.e., check whether it visits every vertex exactly once, in polynomial time (in $O(n \log n)$ exactly).

# Class NP

- Is Hamiltonian Path in P?

# Class NP

- Is Hamiltonian Path in P?
    - We don't know but it is unlikely!

# Class NP

- Is Hamiltonian Path in P?
  - We don't know but it is unlikely!
- Is Hamiltonian Path in NP?

# Class NP

- Is Hamiltonian Path in P?
  - We don't know but it is unlikely!
- Is Hamiltonian Path in NP?
  - Yes, we just showed given a solution (a candidate path), we can check in polynomial time whether it is Hamiltonian.

# Class NP

- Is Hamiltonian Path in P?
  - We don't know but it is unlikely!
- Is Hamiltonian Path in NP?
  - Yes, we just showed given a solution (a candidate path), we can check in polynomial time whether it is Hamiltonian.
- Is 3SUM in $P$?

# Class NP

- Is Hamiltonian Path in P?
  - We don't know but it is unlikely!
- Is Hamiltonian Path in NP?
  - Yes, we just showed given a solution (a candidate path), we can check in polynomial time whether it is Hamiltonian.
- Is 3SUM in $P$?
  - Yes, because it can be solved in $O(n^2)$.

# Class NP

- Is Hamiltonian Path in P?
  - We don't know but it is unlikely!
- Is Hamiltonian Path in NP?
  - Yes, we just showed given a solution (a candidate path), we can check in polynomial time whether it is Hamiltonian.
- Is 3SUM in $P$?
  - Yes, because it can be solved in $O(n^2)$.
- Is 3SUM in $NP$?

# Class NP

- Is Hamiltonian Path in P?
  - We don't know but it is unlikely!
- Is Hamiltonian Path in NP?
  - Yes, we just showed given a solution (a candidate path), we can check in polynomial time whether it is Hamiltonian.
- Is 3SUM in $P$?
  - Yes, because it can be solved in $O(n^2)$.
- Is 3SUM in $NP$?
  - Yes, given a solution (3 numbers from the set), we can verify in polynomial time whether they sum to 0.

# P vs NP

- If a problem can be solved in polynomial time (belongs to $P$), a solution to that can be checked in polynomial time (belongs to $NP$) $\rightarrow$ Eevery problem in $P$ also belongs to $NP$.

# P vs NP

- If a problem can be solved in polynomial time (belongs to $P$), a solution to that can be checked in polynomial time (belongs to $NP$) $\rightarrow$ Eevery problem in $P$ also belongs to $NP$.
- Does the other direction hold?

# P vs NP

- If a problem can be solved in polynomial time (belongs to $P$), a solution to that can be checked in polynomial time (belongs to $NP$) $\rightarrow$ Eevery problem in $P$ also belongs to $NP$.
- Does the other direction hold?
  - If a solution to a problem can be checked in polynomial time (e.g., Hamiltonian path), is it true that a polynomial-time algorithm exists for the problem?

# P vs NP

- If a problem can be solved in polynomial time (belongs to $P$), a solution to that can be checked in polynomial time (belongs to $NP$) $\rightarrow$ Eevery problem in $P$ also belongs to $NP$.
- Does the other direction hold?
  - If a solution to a problem can be checked in polynomial time (e.g., Hamiltonian path), is it true that a polynomial-time algorithm exists for the problem?
  - We do not know the answer.

## P vs NP

- If a problem can be solved in polynomial time (belongs to $P$), a solution to that can be checked in polynomial time (belongs to $NP$) $\rightarrow$ Eevery problem in $P$ also belongs to $NP$.
- Does the other direction hold?
  - If a solution to a problem can be checked in polynomial time (e.g., Hamiltonian path), is it true that a polynomial-time algorithm exists for the problem?
  - We do not know the answer.
- Question: Does any problem in NP belong to P? Is it that P=NP?

# P vs NP

- If a problem can be solved in polynomial time (belongs to $P$), a solution to that can be checked in polynomial time (belongs to $NP$) $\rightarrow$ Eevery problem in $P$ also belongs to $NP$.
- Does the other direction hold?
  - If a solution to a problem can be checked in polynomial time (e.g., Hamiltonian path), is it true that a polynomial-time algorithm exists for the problem?
  - We do not know the answer.
- Question: Does any problem in NP belong to P? Is it that P=NP?
  - It is One of seven Millennium Prize problems in mathematics announced in 2000 by Clay Mathematics Institute with a prize of \$1M for solving any of the problems.

# P vs NP

- If a problem can be solved in polynomial time (belongs to $P$), a solution to that can be checked in polynomial time (belongs to $NP$) $\rightarrow$ Eevery problem in $P$ also belongs to $NP$.
- Does the other direction hold?
  - If a solution to a problem can be checked in polynomial time (e.g., Hamiltonian path), is it true that a polynomial-time algorithm exists for the problem?
  - We do not know the answer.
- Question: Does any problem in NP belong to P? Is it that P=NP?
  - It is One of seven Millennium Prize problems in mathematics announced in 2000 by Clay Mathematics Institute with a prize of $1M for solving any of the problems.
  - To date only one of the Millennium has been solved, the Poincare Conjecture, solved by Perelman in 2006; he declined the money. He was also awarded Fields medal and rejected it: *"I'm not interested in money or fame; I don't want to be on display like an animal in a zoo"*.

# P & NP review

- P: class of problems which can be solved in polynomial time, e.g., Minimum Spanning Tree, 3Sum.

# P & NP review

- P: class of problems which can be solved in polynomial time, e.g., Minimum Spanning Tree, 3Sum.
- NP: class of problems for which a solution can be verified in polynomial time.
  - Hamiltonian Path: we can check in $O(n \log n)$ if a given solution (path) is Hamiltonian or not.

# P & NP review

- P: class of problems which can be solved in polynomial time, e.g., Minimum Spanning Tree, 3Sum.
- NP: class of problems for which a solution can be verified in polynomial time.
  - Hamiltonian Path: we can check in $O(n \log n)$ if a given solution (path) is Hamiltonian or not.
  - If a problem can be solved in polynomial time, its solutions can be checked in polynomial time as well, i.e., $P$ is a subset of NP.

# P & NP review

- P: class of problems which can be solved in polynomial time, e.g., Minimum Spanning Tree, 3Sum.
- NP: class of problems for which a solution can be verified in polynomial time.
  - Hamiltonian Path: we can check in $O(n \log n)$ if a given solution (path) is Hamiltonian or not.
  - If a problem can be solved in polynomial time, its solutions can be checked in polynomial time as well, i.e., $P$ is a subset of NP.
  - The other direction is conjectured to be false, i.e., it is conjectured that there are problems which are in NP but not P, i.e., no polynomial algorithm exists for them.
  - Recall this problem ($NP \in P$) which is equal to $P = NP$ is open.

# NP-hard problems

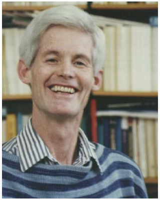
- A problem $Q$ is **NP-hard** if **every** problem in NP **reduces** to $Q$ in polynomial time.

# NP-hard problems

- A problem $Q$ is **NP-hard** if **every** problem in NP **reduces** to $Q$ in polynomial time.
  - Problem $Q$ is as hard as any other problem in NP.

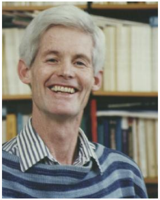
# NP-hard problems

- A problem $Q$ is **NP-hard** if **every** problem in NP **reduces** to $Q$ in polynomial time.
  - Problem $Q$ is as hard as any other problem in NP.
- Stephen Cook, father of complexity:



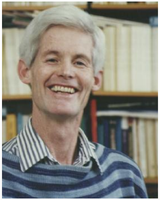
# NP-hard problems

- A problem $Q$ is **NP-hard** if **every** problem in NP **reduces** to $Q$ in polynomial time.
  - Problem $Q$ is as hard as any other problem in NP.
- Stephen Cook, father of complexity:
  - joined UC Berkeley in 1966, denied a tenure in 1970, had to leave Berkeley for U. of Toronto.



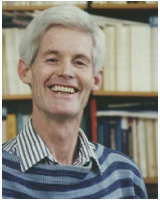
# NP-hard problems

- A problem $Q$ is **NP-hard** if **every** problem in NP **reduces** to $Q$ in polynomial time.
  - Problem $Q$ is as hard as any other problem in NP.
- Stephen Cook, father of complexity:
  - joined UC Berkeley in 1966, denied a tenure in 1970, had to leave Berkeley for U. of Toronto.
- in 1971, Cook published a seminal paper which shaped theory of complexity:



# NP-hard problems

- A problem $Q$ is **NP-hard** if **every** problem in NP **reduces** to $Q$ in polynomial time.
  - Problem $Q$ is as hard as any other problem in NP.
- Stephen Cook, father of complexity:
  - joined UC Berkeley in 1966, denied a tenure in 1970, had to leave Berkeley for U. of Toronto.
- in 1971, Cook published a seminal paper which shaped theory of complexity:
  - defined the concepts of reduction, NP-hardness, and NP-completeness
  - showed that every problem in NP reduces to boolean satisfiability problem (SAT) $\rightarrow$ SAT is NP-hard.



# NP-hard problems

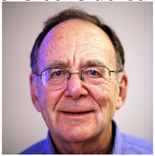
- Reduction is transitive: if problem A reduces to B in time $f(x)$ and B reduces to C in time $g(x)$, then A reduces to C in time $(f(x) + g(x))$.
  - If all NP problems reduce to SAT in polynomial time and SAT reduces to problem $Q$ in polynomial time, then all NP problems reduce to Q in polynomial time (Q is NP-hard).

# NP-hard problems

- Reduction is transitive: if problem A reduces to B in time $f(x)$ and B reduces to C in time $g(x)$, then A reduces to C in time $(f(x) + g(x))$.
  - If all NP problems reduce to SAT in polynomial time and SAT reduces to problem $Q$ in polynomial time, then all NP problems reduce to Q in polynomial time (Q is NP-hard).
- In 1972, Richard Karp from Berkeley showed
  - 21 problems for which no polynomial algorithm exists for years were NP-hard (SAT reduces to them directly or via transition).

# NP-hard problems

- Reduction is transitive: if problem A reduces to B in time $f(x)$ and B reduces to C in time $g(x)$, then A reduces to C in time $(f(x) + g(x))$.
  - If all NP problems reduce to SAT in polynomial time and SAT reduces to problem $Q$ in polynomial time, then all NP problems reduce to Q in polynomial time (Q is NP-hard).
- In 1972, Richard Karp from Berkeley showed
  - 21 problems for which no polynomial algorithm exists for years were NP-hard (SAT reduces to them directly or via transition).
  - Cook got his Turing award in 1982; his departure is considered one of the biggest failures for UC Berkeley.
  - Karp got his Turing award in 1986; partially because his contribution to complexity theory.



# NP-hard problem Consequences

- If a problem $Q$ is NP-hard:
  - All NP-problems reduce to A in polynomial time, i.e., it is at least as hard as any NP problem.
  - Upper bound consequence: if we have a polynomial algorithm that solves $Q$, then there will be polynomial algorithms for all NP problems.
  - Lower bound consequence: if we show there is no polynomial algorithm for any NP problem, then there is no polynomial algorithm for $Q$.

## NP-Complete Problems

- A problem is NP-complete if it belongs to both NP and NP-hard family of problems.

# NP-Complete Problems

- A problem is NP-complete if it belongs to both NP and NP-hard family of problems.
- For a pair of NP-complete problems A and B, A reduces to B in polynomial time and B reduces to A as well.
  - Since A is in NP and B is NP-hard, all NP problems (particularly A) reduce to B.
  - Since B is in NP and A is NP-hard, all NP problems (particularly B) reduce to A.

# NP-Complete Problems

- A problem is NP-complete if it belongs to both NP and NP-hard family of problems.
- For a pair of NP-complete problems A and B, A reduces to B in polynomial time and B reduces to A as well.
  - Since A is in NP and B is NP-hard, all NP problems (particularly A) reduce to B.
  - Since B is in NP and A is NP-hard, all NP problems (particularly B) reduce to A.
- Either both A, B are solvable in polynomial time (the case if P=NP) or neither A,B are solvable in polynomial time (in the more likely case of P$\neq$ NP).

# NP-Complete Problems

- A problem is NP-complete if it belongs to both NP and NP-hard family of problems.
- For a pair of NP-complete problems A and B, A reduces to B in polynomial time and B reduces to A as well.
  - Since A is in NP and B is NP-hard, all NP problems (particularly A) reduce to B.
  - Since B is in NP and A is NP-hard, all NP problems (particularly B) reduce to A.
- Either both A, B are solvable in polynomial time (the case if P=NP) or neither A,B are solvable in polynomial time (in the more likely case of P$\neq$ NP).
- Note that there are NP-problems which are not NP-complete (e.g., 3Sum or MST) and there are NP-hard problems that we do not know whether they belong to NP (EXP-complete problems).

# NP-complete Problems

- If we show a problem is NP-complete, we often stop any effort for designing any polynomial algorithm or devising a polynomial time lower bound (just give up on finding exact solutions for the problem).
  - You might try; but your effort for providing an algorithm/lower bound will be equivalent to trying to solve $P \neq NP$ conjecture.

# NP-complete Problems

- If we show a problem is NP-complete, we often stop any effort for designing any polynomial algorithm or devising a polynomial time lower bound (just give up on finding exact solutions for the problem).
  - You might try; but your effort for providing an algorithm/lower bound will be equivalent to trying to solve $P \neq NP$ conjecture.
- Steps for showing NP-completeness of a problem $A$:
  - Show $A$ is in NP, i.e., show that a yes instance of size $n$ can be verified in polynomial time (i.e., $O(n^c)$).
  - Show that $A$ is NP-hard, i.e., prove that all NP problem reduce to A in polynomial time

# NP-hardness proof

- To prove A is NP-hard:
  - Choose a known NP-complete problem B for the reduction. Any NP-complete problem can be used, but some will have simpler reductions.

# NP-hardness proof

- To prove A is NP-hard:
  - Choose a known NP-complete problem B for the reduction. Any NP-complete problem can be used, but some will have simpler reductions.
  - Define a polynomial-time reduction $f$ that transforms any instance $i$ of $B$ into an instance $f(i)$ of A.

# NP-hardness proof

- To prove A is NP-hard:
  - Choose a known NP-complete problem B for the reduction. Any NP-complete problem can be used, but some will have simpler reductions.
  - Define a polynomial-time reduction $f$ that transforms any instance $i$ of $B$ into an instance $f(i)$ of A.
  - Prove the correctness of the reduction. Show:
    - answer to $i$ is 'yes' $\rightarrow$ answer to $f(i)$ is 'yes'
    - answer to $f(i)$ is 'yes' $\rightarrow$ answer to $i$ is 'yes'

# NP-hardness proof

- To prove A is NP-hard:
  - Choose a known NP-complete problem B for the reduction. Any NP-complete problem can be used, but some will have simpler reductions.
  - Define a polynomial-time reduction $f$ that transforms any instance $i$ of $B$ into an instance $f(i)$ of A.
  - Prove the correctness of the reduction. Show:
    - answer to $i$ is 'yes' $\rightarrow$ answer to $f(i)$ is 'yes'
    - answer to $f(i)$ is 'yes' $\rightarrow$ answer to $i$ is 'yes'
  - Show that the reduction can be computed in time $O(n^c)$ (polynomial time).

# Reduction & Bounds

- Assume we reduce a problem $E$ to problem $H$ (e.g., reduce 3Sum to collinearity).
- Intuitively, $H$ is as hard as $E$

# Reduction & Bounds

- Assume we reduce a problem $E$ to problem $H$ (e.g., reduce 3Sum to collinearity).
- Intuitively, $H$ is as hard as $E$
  - A lower bound $\Omega(f(n))$ for $E$ also applies to $H$, assuming $f(n)$ is not dominated by the reduction time.
    - E.g., lower bound $\Omega(n^{2-\epsilon})$ of 3Sum applies to collinearity, i.e., there is no collinearity algorithm that runs in $\Omega(n^{2-\epsilon})$ (assuming the modern 3Sum conjecture is true).

# Reduction & Bounds

- Assume we reduce a problem $E$ to problem $H$ (e.g., reduce 3Sum to collinearity).
- Intuitively, $H$ is as hard as $E$
  - A lower bound $\Omega(f(n))$ for $E$ also applies to $H$, assuming $f(n)$ is not dominated by the reduction time.
    - E.g., lower bound $\Omega(n^{2-\epsilon})$ of 3Sum applies to collinearity, i.e., there is no collinearity algorithm that runs in $\Omega(n^{2-\epsilon})$ (assuming the modern 3Sum conjecture is true).
  - An upper bound $O(f'(n))$ for $H$ applies to $E$, assuming $f'(n)$ is not dominated by the reduction time.
    - E.g., a Collinearity algorithm that runs in $O(n^2)$ implies that there is an algorithm that runs in $O(n^2)$ for 3Sum .

# Reductions & hardness

- Assume we reduce an NP-hard problem $X$ to problem $Y$ in polynomial time.

# Reductions & hardness

- Assume we reduce an NP-hard problem $X$ to problem $Y$ in polynomial time.
- A lower bound for $X$ also applies to $Y$

# Reductions & hardness

- Assume we reduce an NP-hard problem $X$ to problem $Y$ in polynomial time.
- A lower bound for $X$ also applies to $Y$
  - In particular if we know any algorithm for $X$ runs in $\omega(n^c)$ (i.e., no algorithm for $X$ runs in polynomial time), we can make the same statement for $Y$, i.e., no algorithm for $Y$ runs in polynomial time.

# Reductions & hardness

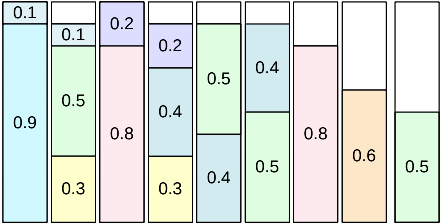
- Assume we reduce an NP-hard problem $X$ to problem $Y$ in polynomial time.
- A lower bound for $X$ also applies to $Y$
  - In particular if we know any algorithm for $X$ runs in $\omega(n^c)$ (i.e., no algorithm for $X$ runs in polynomial time), we can make the same statement for $Y$, i.e., no algorithm for $Y$ runs in polynomial time.
  - If $P \neq NP$, then there is an NP problem $Q$ which has no polynomial time algorithm; such problem reduce to $X$ (by definition of NP-hardness), and $X$ reduces to $Y$. Since $\omega(n^c)$ is a lower bound for $Q$, that would be a lower bound for $Y$, i.e., no algorithm for $Y$ runs in polynomial time.

# Reductions & hardness

- Assume we reduce an NP-hard problem $X$ to problem $Y$ in polynomial time.
- A lower bound for $X$ also applies to $Y$
  - In particular if we know any algorithm for $X$ runs in $\omega(n^c)$ (i.e., no algorithm for $X$ runs in polynomial time), we can make the same statement for $Y$, i.e., no algorithm for $Y$ runs in polynomial time.
  - If $P \neq NP$, then there is an NP problem $Q$ which has no polynomial time algorithm; such problem reduce to $X$ (by definition of NP-hardness), and $X$ reduces to $Y$. Since $\omega(n^c)$ is a lower bound for $Q$, that would be a lower bound for $Y$, i.e., no algorithm for $Y$ runs in polynomial time.
- An upper bound for $Y$ also applies to $X$, i.e., in particular if there is a polynomial time algorithm for $Y$, then that algorithm can be used to answer $X$ (and all NP problems which reduce to $X$) in polynomial time. This implies that $P = NP$.

# Bin Packing Problem

- The input is a **multi-set** of items of various sizes in range (0,1].
- The goal is to pack these items into a minimum number of bins of uniform capacity.
  - E.g., $S = \{0.1, 0.2, 0.2, 0.3, 0.3, 0.4, 0.4, 0.4, 0.5, 0.5, 0.5, 0.5, 0.6, 0.8, 0.8, 0.9\}$



# First Fit Algorithm

- First Fit: process items one by one in arbitrary order. Place each item in the first bin which has enough space for the item.
  - Open a new bin if such bin does not exist.

# First Fit Algorithm

- First Fit: process items one by one in arbitrary order. Place each item in the first bin which has enough space for the item.
  - Open a new bin if such bin does not exist.

< 0.9 0.3 0.8 0.5 0.1 0.1 0.3 0.2 0.4 0.2 0.4 0.5 0.5 0.8 0.6 0.4 0.5 ... >

# First Fit Algorithm

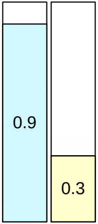
- First Fit: process items one by one in arbitrary order. Place each item in the first bin which has enough space for the item.
  - Open a new bin if such bin does not exist.

< 0.9 0.3 0.8 0.5 0.1 0.1 0.3 0.2 0.4 0.2 0.4 0.5 0.5 0.8 0.6 0.4 0.5 ... >

# First Fit Algorithm

- First Fit: process items one by one in arbitrary order. Place each item in the first bin which has enough space for the item.
  - Open a new bin if such bin does not exist.

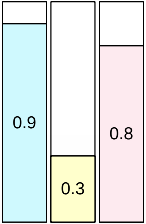
< 0.9 0.3 0.8 0.5 0.1 0.1 0.3 0.2 0.4 0.2 0.4 0.5 0.5 0.8 0.6 0.4 0.5 ... >



# First Fit Algorithm

- First Fit: process items one by one in arbitrary order. Place each item in the first bin which has enough space for the item.
  - Open a new bin if such bin does not exist.

< 0.9 0.3 0.8 0.5 0.1 0.1 0.3 0.2 0.4 0.2 0.4 0.5 0.5 0.8 0.6 0.4 0.5 ... >



# First Fit Algorithm

- First Fit: process items one by one in arbitrary order. Place each item in the first bin which has enough space for the item.
  - Open a new bin if such bin does not exist.

< 0.9 0.3 0.8 0.5 0.1 0.1 0.3 0.2 0.4 0.2 0.4 0.5 0.5 0.8 0.6 0.4 0.5 ... >

# First Fit Algorithm

- First Fit: process items one by one in arbitrary order. Place each item in the first bin which has enough space for the item.
  - Open a new bin if such bin does not exist.

< 0.9 0.3 0.8 0.5 0.1 0.1 0.3 0.2 0.4 0.2 0.4 0.5 0.5 0.8 0.6 0.4 0.5 ... >

# First Fit Algorithm

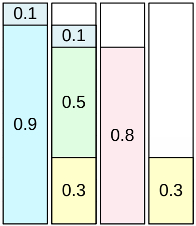
- First Fit: process items one by one in arbitrary order. Place each item in the first bin which has enough space for the item.
  - Open a new bin if such bin does not exist.

< 0.9 0.3 0.8 0.5 0.1 0.1 0.3 0.2 0.4 0.2 0.4 0.5 0.5 0.8 0.6 0.4 0.5 ... >

# First Fit Algorithm

- First Fit: process items one by one in arbitrary order. Place each item in the first bin which has enough space for the item.
    - Open a new bin if such bin does not exist.

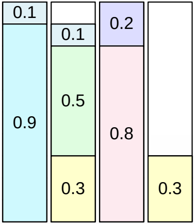
< 0.9 0.3 0.8 0.5 0.1 0.1 0.3 0.2 0.4 0.2 0.4 0.5 0.5 0.8 0.6 0.4 0.5 ... >



# First Fit Algorithm

- First Fit: process items one by one in arbitrary order. Place each item in the first bin which has enough space for the item.
  - Open a new bin if such bin does not exist.

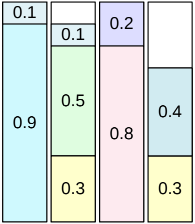
< 0.9 0.3 0.8 0.5 0.1 0.1 0.3 0.2 0.4 0.2 0.4 0.5 0.5 0.8 0.6 0.4 0.5 ... >



# First Fit Algorithm

- First Fit: process items one by one in arbitrary order. Place each item in the first bin which has enough space for the item.
  - Open a new bin if such bin does not exist.

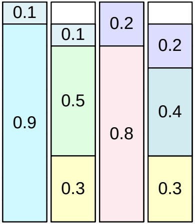
< 0.9 0.3 0.8 0.5 0.1 0.1 0.3 0.2 0.4 0.2 0.4 0.5 0.5 0.8 0.6 0.4 0.5 ... >



# First Fit Algorithm

- First Fit: process items one by one in arbitrary order. Place each item in the first bin which has enough space for the item.
  - Open a new bin if such bin does not exist.

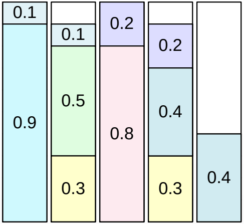
< 0.9 0.3 0.8 0.5 0.1 0.1 0.3 0.2 0.4 0.2 0.4 0.5 0.5 0.8 0.6 0.4 0.5 ... >



# First Fit Algorithm

- First Fit: process items one by one in arbitrary order. Place each item in the first bin which has enough space for the item.
  - Open a new bin if such bin does not exist.

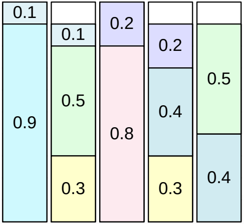
< 0.9 0.3 0.8 0.5 0.1 0.1 0.3 0.2 0.4 0.2 0.4 0.5 0.5 0.8 0.6 0.4 0.5 ... >



# First Fit Algorithm

- First Fit: process items one by one in arbitrary order. Place each item in the first bin which has enough space for the item.
  - Open a new bin if such bin does not exist.

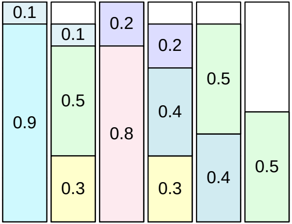
< 0.9 0.3 0.8 0.5 0.1 0.1 0.3 0.2 0.4 0.2 0.4 0.5 0.5 0.8 0.6 0.4 0.5 … >



# First Fit Algorithm

- First Fit: process items one by one in arbitrary order. Place each item in the first bin which has enough space for the item.
  - Open a new bin if such bin does not exist.

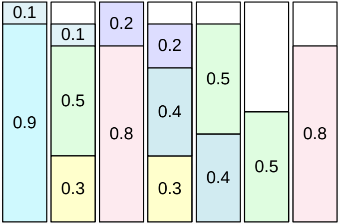
< 0.9 0.3 0.8 0.5 0.1 0.1 0.3 0.2 0.4 0.2 0.4 0.5 0.5 0.8 0.6 0.4 0.5 ... >



# First Fit Algorithm

- First Fit: process items one by one in arbitrary order. Place each item in the first bin which has enough space for the item.
  - Open a new bin if such bin does not exist.

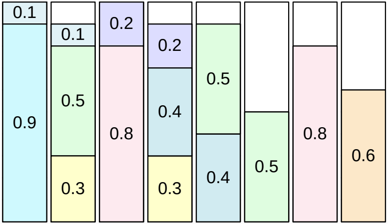
< 0.9 0.3 0.8 0.5 0.1 0.1 0.3 0.2 0.4 0.2 0.4 0.5 0.5 0.8 0.6 0.4 0.5 ... >



# First Fit Algorithm

- First Fit: process items one by one in arbitrary order. Place each item in the first bin which has enough space for the item.
  - Open a new bin if such bin does not exist.

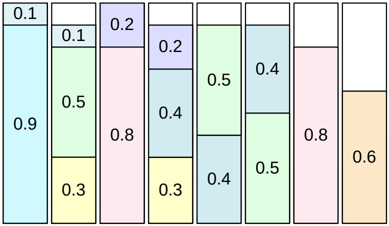
< 0.9 0.3 0.8 0.5 0.1 0.1 0.3 0.2 0.4 0.2 0.4 0.5 0.5 0.8 0.6 0.4 0.5 ... >



# First Fit Algorithm

- First Fit: process items one by one in arbitrary order. Place each item in the first bin which has enough space for the item.
  - Open a new bin if such bin does not exist.

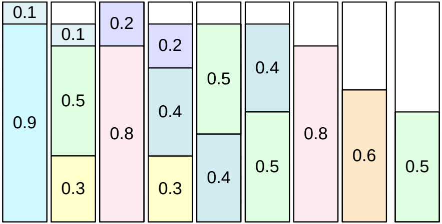
< 0.9 0.3 0.8 0.5 0.1 0.1 0.3 0.2 0.4 0.2 0.4 0.5 0.5 0.8 0.6 0.4 0.5 ... >



# First Fit Algorithm

- First Fit: process items one by one in arbitrary order. Place each item in the first bin which has enough space for the item.
    - Open a new bin if such bin does not exist.

< 0.9 0.3 0.8 0.5 0.1 0.1 0.3 0.2 0.4 0.2 0.4 0.5 0.5 0.8 0.6 0.4 0.5 ... >



# Applications of Bin Packing

- Loading trucks (e.g., trucks moving between Toronto and Montreal)
    - Truck have uniform weight capacity.

# Applications of Bin Packing

- Loading trucks (e.g., trucks moving between Toronto and Montreal)
  - Truck have uniform weight capacity.
- Stock cutting, e.g., cutting standard-sized wood material (bins) into pieces of specified sizes (items).

# Applications of Bin Packing

- Loading trucks (e.g., trucks moving between Toronto and Montreal)
  - Truck have uniform weight capacity.
- Stock cutting, e.g., cutting standard-sized wood material (bins) into pieces of specified sizes (items).
- Server consolidation (e.g., in cloud)
  - Servers are bins and items are clients (e.g., cloud tenants) and you want to minimize the number of active servers.

# Complexity of Bin Packing

- Decision Variant of Bin Packing: given a multi-set of items, is it possible to pack them into $k$ bins?

# Complexity of Bin Packing

- Decision Variant of Bin Packing: given a multi-set of items, is it possible to pack them into $k$ bins?
- We show this problem is NP-complete even for the easy case of $k = 2$.
- Decision problem: given a multi-set of items, is it possible to pack them into 2 bins?

# Complexity of Bin Packing

- Decision Variant of Bin Packing: given a multi-set of items, is it possible to pack them into $k$ bins?
- We show this problem is NP-complete even for the easy case of $k = 2$.
- Decision problem: given a multi-set of items, is it possible to pack them into 2 bins?
- The problem is in NP: given a solution (e.g., assignment of items to 2 bins), we can check in linear (i.e., polynomial) time whether the total size of items in each bin is at most 1.

# Bin Packing Hardness

- Prove that it is NP-Hard to decide whether a multi-set of items can be packed in 2 bins.

# Bin Packing Hardness

- Prove that it is NP-Hard to decide whether a multi-set of items can be packed in 2 bins.
  - reduction from the **partition problem**

# Bin Packing Hardness

- Prove that it is NP-Hard to decide whether a multi-set of items can be packed in 2 bins.
  - reduction from the **partition problem**
- **Partition**: decide whether a multiset $P$ of positive integers can be partitioned into two subsets $S$ and $P - S$ .t.
  sum of the numbers in $S$ = sum of the numbers in $P - S$
  - $P = \{3, 1, 3, 2, 3, 2, 3, 3, 4, 1\} \rightarrow S = \{3, 2, 3, 3\} \quad P - S = \{1, 3, 2, 4, 1\}$

## Bin Packing Hardness

- Prove that it is NP-Hard to decide whether a multi-set of items can be packed in 2 bins.
  - reduction from the **partition problem**
- **Partition**: decide whether a multiset $P$ of positive integers can be partitioned into two subsets $S$ and $P - S$ .t.
  sum of the numbers in $S$ = sum of the numbers in $P - S$
  - $P = \{3, 1, 3, 2, 3, 2, 3, 3, 4, 1\} \rightarrow S = \{3, 2, 3, 3\}$ $P - S = \{1, 3, 2, 4, 1\}$
- Partition is NP-complete, i.e., assuming $P \neq NP$ there is no algorithm that runs in $O(n^c)$ for an input of emphlength $n$.

# Bin Packing Hardness

- Prove that it is NP-Hard to decide whether a multi-set of items can be packed in 2 bins.
  - reduction from the **partition problem**
- **Partition**: decide whether a multiset $P$ of positive integers can be partitioned into two subsets $S$ and $P - S$ .t.
  sum of the numbers in $S$ = sum of the numbers in $P - S$
  - $P = \{3, 1, 3, 2, 3, 2, 3, 3, 4, 1\} \rightarrow S = \{3, 2, 3, 3\}$ $P - S = \{1, 3, 2, 4, 1\}$
- Partition is NP-complete, i.e., assuming $P \neq NP$ there is no algorithm that runs in $O(n^c)$ for an input of emphlength $n$.
  - An algorithm runs in polynomial if it is polynomial in the **length** of the input

# Bin Packing Hardness

- Prove that it is NP-Hard to decide whether a multi-set of items can be packed in 2 bins.
  - reduction from the **partition problem**
- **Partition**: decide whether a multiset $P$ of positive integers can be partitioned into two subsets $S$ and $P - S$ .t.
  sum of the numbers in $S$ = sum of the numbers in $P - S$
  - $P = \{3, 1, 3, 2, 3, 2, 3, 3, 4, 1\} \rightarrow S = \{3, 2, 3, 3\}$ $P - S = \{1, 3, 2, 4, 1\}$
- Partition is NP-complete, i.e., assuming $P \neq NP$ there is no algorithm that runs in $O(n^c)$ for an input of emphlength $n$.
  - An algorithm runs in polynomial if it is polynomial in the **length** of the input
  - E.g., a polynomial time algorithm should run in polynomial time even if there is an integer $2^n$ in the input (the input length will be still polynomial and that number's length is $n$ in the input).

# Bin Packing Hardness

- Prove that it is NP-Hard to decide whether a multi-set of items can be packed in 2 bins.
  - reduction from the **partition problem**
- **Partition**: decide whether a multiset $P$ of positive integers can be partitioned into two subsets $S$ and $P - S$ .t.
  sum of the numbers in $S$ = sum of the numbers in $P - S$
  - $P = \{3, 1, 3, 2, 3, 2, 3, 3, 4, 1\} \rightarrow S = \{3, 2, 3, 3\}$ $P - S = \{1, 3, 2, 4, 1\}$
- Partition is NP-complete, i.e., assuming $P \neq NP$ there is no algorithm that runs in $O(n^c)$ for an input of emphlength $n$.
  - An algorithm runs in polynomial if it is polynomial in the **length** of the input
  - E.g., a polynomial time algorithm should run in polynomial time even if there is an integer $2^n$ in the input (the input length will be still polynomial and that number's length is $n$ in the input).
  - If all numbers are $O(n^c)$, there is an algorithm that runs in polynomial time; that is called a **pseudo-polynomial** algorithm.

# Reduction from Partition to Bin Packing

- Assume we have an instance $P = \{p_1, p_2, \ldots p_n\}$ of Partition problem.

# Reduction from Partition to Bin Packing

- Assume we have an instance $P = \{p_1, p_2, \ldots p_n\}$ of Partition problem.
- Create an instance of bin packing as follows:
  - Let $t = \sum\limits_{p_i \in P} p_i$.
  - Define a multi-set of item sizes $Q = \{q_1, \ldots, q_n\}$ such that $q_i = p_i \cdot \frac{2}{t}$.

# Reduction from Partition to Bin Packing

- Assume we have an instance $P = \{p_1, p_2, \ldots p_n\}$ of Partition problem.
- Create an instance of bin packing as follows:
  - Let $t = \sum_{p_i \in P} p_i$.
  - Define a multi-set of item sizes $Q = \{q_1, \ldots, q_n\}$ such that $q_i = p_i \cdot \frac{2}{t}$.
  - Note that we have $\sum_{q_i \in Q} q_i = \frac{2}{t} \cdot \sum_{p_i \in P} p_i = \frac{2}{t} \cdot t = 2$.

# Validity of Reduction

- **Show that the answer to the partition instance $P = \{p_1, p_2, \ldots p_n\}$ is yes if and only the answer to in packing instance $Q = \{q_1, \ldots, q_n\}$ is yes (i.e., items can be packed in 2 bins)**.
    - Recall that $q_i = p_i \cdot \frac{2}{t}$.

## Validity of Reduction

- **Show that the answer to the partition instance $P = \{p_1, p_2, \ldots p_n\}$ is yes if and only the answer to in packing instance $Q = \{q_1, \ldots, q_n\}$ is yes (i.e., items can be packed in 2 bins)**.
  - Recall that $q_i = p_i \cdot \frac{2}{t}$.
- Assume the answer to the partition instance is yes
  - I.e., there is $S \in P$ so that $\sum_{p_i \in S} p_i = \sum_{p_i \in P-S} p_i = t/2$

## Validity of Reduction

- **Show that the answer to the partition instance $P = \{p_1, p_2, \ldots p_n\}$ is yes if and only the answer to in packing instance $Q = \{q_1, \ldots, q_n\}$ is yes (i.e., items can be packed in 2 bins)**.
  - Recall that $q_i = p_i \cdot \frac{2}{t}$.
- Assume the answer to the partition instance is yes
  - I.e., there is $S \in P$ so that $\sum_{p_i \in S} p_i = \sum_{p_i \in P-S} p_i = t/2$
- We show that the bin packing instance can be packed into 2 bins.
  - Since $\sum_{p_i \in S} p_i = \sum_{p_i \in P-S} p_i = t/2$, we have $\sum_{p_i \in S} q_i = \sum_{p_i \in P-S} q_i = \frac{t}{2} \cdot \frac{2}{t} = 1$.
  - We can pack the items associated with set $S$ (i.e., set of $q_i$'s s.t. $p_i \in S$) in one bin and the rest in another.
  - The total size in each bin will not be more than 1 (hence a valid packing).

# Validity of Reduction

- Next, we show if the answer to the bin packing instance $Q = \{q_1, \ldots, q_n\}$ is yes, then the answer to the partition problem is yes.

# Validity of Reduction

- Next, we show if the answer to the bin packing instance $Q = \{q_1, \ldots, q_n\}$ is yes, then the answer to the partition problem is yes.
  - Recall that $\sum\limits_{q_i \in Q} q_i = \frac{2}{t} \cdot \sum\limits_{p_i \in P} p_i = \frac{2}{t} \cdot t = 2.$

# Validity of Reduction

- Next, we show if the answer to the bin packing instance $Q = \{q_1, \ldots, q_n\}$ is yes, then the answer to the partition problem is yes.
  - Recall that $\sum_{q_i \in Q} q_i = \frac{2}{t} \cdot \sum_{p_i \in P} p_i = \frac{2}{t} \cdot t = 2$.
  - Our assumption implies that a set of items of total size 2 have been packed into 2 bins $\rightarrow$ each bin is completely full.

# Validity of Reduction

- Next, we show if the answer to the bin packing instance $Q = \{q_1, \ldots, q_n\}$ is yes, then the answer to the partition problem is yes.
  - Recall that $\sum\limits_{q_i \in Q} q_i = \frac{2}{t} \cdot \sum\limits_{p_i \in P} p_i = \frac{2}{t} \cdot t = 2$.
  - Our assumption implies that a set of items of total size 2 have been packed into 2 bins $\rightarrow$ each bin is completely full.
  - Our packing is equivalent to partitioning $Q$ into two subsets $R$, $Q - R$ each of total size 1, i.e., $\sum\limits_{q_i \in R} q_i = \sum\limits_{q_i \in Q-R} q_i = 1$.

## Validity of Reduction

- Next, we show if the answer to the bin packing instance $Q = \{q_1, \ldots, q_n\}$ is yes, then the answer to the partition problem is yes.
  - Recall that $\sum\limits_{q_i \in Q} q_i = \frac{2}{t} \cdot \sum\limits_{p_i \in P} p_i = \frac{2}{t} \cdot t = 2$.
  - Our assumption implies that a set of items of total size 2 have been packed into 2 bins $\rightarrow$ each bin is completely full.
  - Our packing is equivalent to partitioning $Q$ into two subsets $R$, $Q - R$ each of total size 1, i.e., $\sum\limits_{q_i \in R} q_i = \sum\limits_{q_i \in Q - R} q_i = 1$.
  - Let $S$ be the multiset associated with items of $R$ in the partition instance, i.e., $S = \cup_{q_i \in R}\{p_i\}$.

# Validity of Reduction

- Next, we show if the answer to the bin packing instance $Q = \{q_1, \ldots, q_n\}$ is yes, then the answer to the partition problem is yes.
  - Recall that $\sum_{q_i \in Q} q_i = \frac{2}{t} \cdot \sum_{p_i \in P} p_i = \frac{2}{t} \cdot t = 2$.
  - Our assumption implies that a set of items of total size 2 have been packed into 2 bins $\rightarrow$ each bin is completely full.
  - Our packing is equivalent to partitioning $Q$ into two subsets $R$, $Q - R$ each of total size 1, i.e., $\sum_{q_i \in R} q_i = \sum_{q_i \in Q-R} q_i = 1$.
  - Let $S$ be the multiset associated with items of $R$ in the partition instance, i.e., $S = \cup_{q_i \in R}\{p_i\}$.
  - We have $\sum_{p_i \in S} p_i = \sum_{p_i \in S} q_i \cdot \frac{t}{2} = \frac{t}{2}$.
  - So, $S$ and $P - S$ will be two subsets of the partition instance each with total sum of $t/2 \rightarrow$ the answer to partition instance is yes.

# Bin Packing NP-completeness

- Given any instance of the partition problem $P$ with total sum $t$, we created an instance $Q$ of bin packing problem by scaling down the size of numbers in $P$ by a factor of $t/2$ so that their total sum is 2.

# Bin Packing NP-completeness

- Given any instance of the partition problem $P$ with total sum $t$, we created an instance $Q$ of bin packing problem by scaling down the size of numbers in $P$ by a factor of $t/2$ so that their total sum is 2.
  - It is possible to partition $P$ into two groups, each of total sum $t/2$ $\leftrightarrow$
    It is possible to partition $Q$ into two groups, each of total sum 1 $\leftrightarrow$
    It is possible to pack items into two bins.

# Bin Packing NP-completeness

- Given any instance of the partition problem $P$ with total sum $t$, we created an instance $Q$ of bin packing problem by scaling down the size of numbers in $P$ by a factor of $t/2$ so that their total sum is 2.
  - It is possible to partition $P$ into two groups, each of total sum $t/2$ $\leftrightarrow$
    It is possible to partition $Q$ into two groups, each of total sum 1 $\leftrightarrow$
    It is possible to pack items into two bins.
  - The answer to partition instance is yes if and only if the packing instance can be packed into 2 bins.
  - This means answering the decision problem "can a multiset of items be packed into 2 bins" is NP-hard.

# Bin Packing NP-completeness

- Given any instance of the partition problem $P$ with total sum $t$, we created an instance $Q$ of bin packing problem by scaling down the size of numbers in $P$ by a factor of $t/2$ so that their total sum is 2.
  - It is possible to partition $P$ into two groups, each of total sum $t/2$
    $\leftrightarrow$
    It is possible to partition $Q$ into two groups, each of total sum 1 $\leftrightarrow$
    It is possible to pack items into two bins.
  - The answer to partition instance is yes if and only if the packing instance can be packed into 2 bins.
  - This means answering the decision problem "can a multiset of items be packed into 2 bins" is NP-hard.
- We showed the decision variant of bin packing is NP, i.e., we can check whether a given solution to bin packing is valid (total size of items i each bin is at most 1) or not in polynomial time.

## Bin Packing NP-completeness

- Given any instance of the partition problem $P$ with total sum $t$, we created an instance $Q$ of bin packing problem by scaling down the size of numbers in $P$ by a factor of $t/2$ so that their total sum is 2.
  - It is possible to partition $P$ into two groups, each of total sum $t/2$ $\leftrightarrow$
    It is possible to partition $Q$ into two groups, each of total sum 1 $\leftrightarrow$
    It is possible to pack items into two bins.
  - The answer to partition instance is yes if and only if the packing instance can be packed into 2 bins.
  - This means answering the decision problem "can a multiset of items be packed into 2 bins" is NP-hard.
- We showed the decision variant of bin packing is NP, i.e., we can check whether a given solution to bin packing is valid (total size of items i each bin is at most 1) or not in polynomial time.

**Bin Packing is an NP-complete problem.**

# Approximation Algorithms

- Given an NP-complete optimization problem, there is no optimal polynomial algorithm, assuming $P \neq NP$.

# Approximation Algorithms

- Given an NP-complete optimization problem, there is no optimal polynomial algorithm, assuming $P \neq NP$.
- We can **approximate** the solution!
- The solution provided by an approximation algorithm is not necessarily optimal but an approximation of that.

# Approximation Algorithms

- Given an NP-complete optimization problem, there is no optimal polynomial algorithm, assuming $P \neq NP$.
- We can **approximate** the solution!
- The solution provided by an approximation algorithm is not necessarily optimal but an approximation of that.

# The Ending

**Observation**

*You should aim for the stars - and hopefully avoid ending up in the clouds!* Roxanne McKee

- We covered some materials about algorithms & complexity; the goal was not to cover everything; but prepare you to get interested and discover yourself in your future career.
- When dealing with a problem, we are interested in:
  - designing algorithms for them (using tools such as data structures)
  - analyzing algorithms (based on time complexity, memory requirement, approximation ratio, etc.) to provide guarantees.
  - understanding the restrictions of algorithms (lower bounds and complexity classes).
- 99 percent of people who talk about algorithms (e.g., in media, news, etc.) don't understand them. Hopefully you are not one of them any more.

## The Ending

**Observation**

*You should aim for the stars - and hopefully avoid ending up in the clouds!* Roxanne McKee

- Template for final is posted. If any thing in the slides is not clear, ask me to explain it on Piazza.

## The Ending

**Observation**

*You should aim for the stars - and hopefully avoid ending up in the clouds!* Roxanne McKee

- Template for final is posted. If any thing in the slides is not clear, ask me to explain it on Piazza.
- Your feedback is appreciated; if something can be improved (which is 100 percent the case), let me know.

# The Ending

**Observation**

*You should aim for the stars - and hopefully avoid ending up in the clouds!* Roxanne McKee

- Template for final is posted. If any thing in the slides is not clear, ask me to explain it on Piazza.
- Your feedback is appreciated; if something can be improved (which is 100 percent the case), let me know.
- I hope to see you in future courses.