COMP 3170 - Analysis of Algorithms & Data Structures

Shahin Kamali

Computational Complexity

CLRS 34.1-34.4

University of Manitoba
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-sort), $\Theta(n^2)$ (e.g., bubble-sort), $\Theta(n^3)$ (e.g., matrix multiplication), etc.

- 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., $O(n), O(n^2), O(n^3), O(n^{2018})$.
  - 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.
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 $E(G)$?
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., $O(n^22^n)$ 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’.
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 there is a polynomial algorithm for $P$ if and only if there is a polynomial algorithm 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 \textbf{NP-Hard} in that case!
  - To show \( P \) is NP-Hard, we reduce another NP-Hard problem to \( P \).
Complexity Classes

- A complexity class is a set of problems that can be solved with a similar amount of time/space/cost resources.
  - E.g., 3Sum-hard problems are those which require at least the same amount of time than 3Sum problem.

- 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 $m\alpha(m, n) \in O(n^2\alpha(n^2, n)) \in O(n^3)$.  
  - Basically, all problems for which you have seen an algorithm belong to class $P$ of problems.
A problem belongs to class $NP$ if a non-deterministic Turning 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?
  - 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, i.e., it belongs to $NP$.
  - Every 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 has been solved: the Poincare Conjecture, solved by Perelman in 2006; he declined the money.
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 $P$ is **NP-hard** if every problem in NP reduces to $P$ in polynomial time.
  - Problem $P$ is as hard as any other problem in NP.

- Stephen Cook, father of complexity:
  
- 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 \( X \) in polynomial time, then all NP problems reduce to \( X \) in polynomial time (\( X \) 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 1971; 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 $A$ 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 $A$, 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 $A$. 
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 $P \neq NP$).

- 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; wait for the next classes).
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
To prove A is NP-hard:

1. Choose a known NP-complete problem B for the reduction. Any NP-complete problem can be used, but some will have simpler reductions.
2. Define a polynomial-time reduction $f$ that transforms any instance $i$ of $B$ into an instance $f(i)$ of A.
3. 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’
4. Show that the reduction can be computed in time $O(n^c)$ (polynomial time).
3-Coloring Problem

- **Instance**: a graph $G = (V, E)$ with $n$ vertices in the set $V(G)$ and $m$ edges in the set $E(G)$
- **Question**: Can each vertex in $V(G)$ be coloured red, blue, or green such that the endpoints of every edge in $E(G)$ are different colours?
- **3-Coloring** is an easy decision variant of graph coloring problem: color a graph using a minimum number of colors.

\[\text{A Yes Instance}\]

\[\text{A No Instance}\]
Graph Coloring Application

- One application is exam-scheduling: What is the minimum number of time slots required to schedule final exams such that no student has two simultaneous exams?
  - Create a graph in which each vertex is a course
  - Two vertices (courses) are connected if a student takes both courses
  - Colour the graph using as few colours as possible: each colour corresponds to a time slot in the exam schedule.
  - The minimum number of colours is exactly the minimum number of time slots necessary for a conflict-free exam schedule.
3-Coloring Complexity

- Task in hand: prove 3-coloring is NP-complete.

- Step I: show 3-coloring is NP.
  - Given a candidate coloring, we can check if it uses 3-colors and endpoints of each edge have different colors. This check can be done in polynomial time.
  - We can check an instance in polynomial time → 3-coloring is in NP.

- Step II: show 3-coloring is NP-hard
  - Go find a suitable NP-hard problem $B$ and reduce it to 3-coloring.
  - Use your instinct, the list of hard problems from Wikipedia, or/and black-magic book by Garey and Johnosn.
  - Here, we reduce 3-Sat to 3-coloring
3-Sat Problem

- Instance: a conjunction of $m$ clauses, each of which is the disjunction of three literals selected from a set of $n$ literals.

- Question: Do there exist truth assignments for the literals that satisfy all of the clauses?

- E.g.,

\[(x \lor y \lor \neg z) \land (w \lor \neg x \lor z) \land (w \lor \neg y \lor z) \land (w \lor y \lor \neg z) \land (\neg w \lor \neg x \lor \neg y)\]

- This is a ‘yes’ instance consisting of 5 clauses from a set of 4 literals. The instance is true when $w = \text{true}$, $x = \text{true}$, $y = \text{false}$, $z = \text{false}$.

  - This is not the only possible truth assignment that satisfies all clauses.
Reduction Summary

Reduction of 3-Sat to 3-coloring in a nutshell:

- Our reduction \( f \) that takes any instance \( i \) of 3-SAT and transforms it into a graph \( f(i) \) 3-COLORABILITY
- The answer to \( i \) is yes iff the answer to \( f(i) \) is yes
  - There exists a truth assignment to the literals of \( i \) that satisfies all its clauses iff there exists a 3-colouring of \( f(i) \) such that no 2 adjacent vertices have the same colour.

How should we define \( f \)?

- How to create an instance of 3-colorability from a given 3-Sat instance?
Defining Reduction

- Add a vertex for each literal $x_i$ and another for its negation $\neg x_i$.
- Add an edge between the vertices for $x_i$ and $\neg x_i$ to ensure they are assigned different colours.
- Connect all literal vertices with a vertex $B$ (wlog assume $B$'s color is blue).
  - Forming a triangle with $B$ vertex ensures one of $x_i$ and $\neg x_i$ is green and the other is red.
  - In our transformation, green = true and red = false.
Defining Reduction

- We connect $B$ to a second triangle, which will force one vertex to be green (true) and the other to be red (false).
  - These two global truth value vertices will connect to clauses (along with the literals in each clause)
Defining Reduction

For each clause, add a gadget as illustrated below:

- Each gadget has 6 vertices which are connected as depicted.
- The gadget is designed in a way that we prove the reduction is valid, i.e., the answer to 3Sat instance is yes if and only if the answer to the 3coloring is yes.
Validity of Reduction

- Direction 1: assume the answer to 3Sat instance is yes, i.e., there is true/false assignment to literals so that all clauses are satisfied.
  - We show that we can color the graph using three colors
    - Color true/false literals using green/red colors
    - We show for all 7 possibilities, it is possible to color gadget vertices of a clause using 3 colors (there are 7 ways to satisfy a clause).
Validity of Reduction

- Direction 2: assume the answer to 3Coloring instance is yes
  - I.e., it is possible to color the graph using three colors
  - We show we can assign true/false values to literals so that all clauses are satisfied
  - In the 3coloring solution, not all three literal-vertices involved in a clause can be red
Validity of Reduction

- A 3coloring is equivalent to color literal-vertices red/green (they cannot be blue because all connected to B), so that \( x \) and \( \neg x \) have different color and at least one of the three vertices involved in a clause is green \( \rightarrow \) assigning true to green literals satisfy the 3Sat formula.

- We showed ‘answer to 3coloring is yes \( \rightarrow \) answer to 3Sat is yes’.
Validity of Reduction

We showed that the answer to 3Sat instance is yes if and only if the answer to the reduced 3coloring problem is Yes.

We could reduce 3Sat to 3Coloring. What is the reduction time?

- literals: $3 + 2n$ vertices, $3 + 3n$ edges.
- clauses: $6m$ vertices, $13m$ edges.
- $\rightarrow \Theta(n + m)$ vertices and edges.
- $\rightarrow$ Reduction can be done in $\Theta(n + m)$ time which is polynomial.

It is possible to reduce 3Sat to 3Coloring in polynomial time. Since 3Sat is NP-hard, 3Coloring is NP hard as well.

- We showed 3Coloring is NP before; hence, it is NP-Complete, i.e., there is a polynomial time algorithm for it if and only if there is a polynomial algorithm for all thousands of other NP-complete problems.
Lower Bounds Review

- Consider a problem such as 3Sum.
- A lower bound of $\Omega(f(n))$ implies that any algorithm for 3Sum runs in $\Omega(f(n))$, i.e., no algorithm runs in $o(f(n))$.
  - E.g., $\Omega(1)$ is a valid lower bound; any algorithm runs in constant time or more.
  - E.g., $\Omega(\log n)$ is a better lower bound; any algorithm runs in logarithmic time or more (it is better because constant-time algorithms are ruled out).
  - E.g., $\Omega(n)$ is an even better lower bound; any algorithm runs in at least linear time (again, it is better since sublinear algorithms are ruled out).
  - If we can prove $\Omega(n^{2-\epsilon})$, that would be much better lower bound (e.g., algorithms running in $O(n)$ or $O(n \log n)$ are ruled out).
- When discussing lower bounds, we express them using $\Omega$ and $\omega$ notations and we prefer to improve them by providing asymptotically larger lower bounds.
Upper Bounds Review

- Upper bounds for problems (e.g., 3Sum) are associated with algorithms.
- An upper bound of $f(n)$ implies that there is an algorithm for the problem that runs in $O(f(n))$.
  - $O(n^3)$ is a valid upper bound for 3Sum since there are algorithms running in $O(n^3)$.
  - $O(n^2 \log n)$ is a better upper bound since the time complexity of the involved algorithm is asymptotically smaller.
  - $O(n^2)$ is even better upper bound, i.e., a better algorithm.

- When discussing upper bounds, we express them using $O$ and $o$ notations and prefer to improve them by providing asymptotically smaller upper bounds.
Upper and Lower Bound Review

- Given a problem such as 3Sum:
  - Start with a simple, naive lower bound (e.g., $\Omega(n)$ is a simple lower bound for 3Sum since any algorithm has to read the input to given a ‘no’ answer to ‘no’ instances, i.e., it requires $\Omega(n)$ time).
  - Start with a naive algorithm (e.g., try all triplets and see if any of them sums to 0). That would take $O(n^3)$.
  - There is a gap between $\Omega(n)$ and $O(n^3)$, i.e., we know the best algorithm for the problem has a running time in this range (asymptotically no smaller than $n$ and no larger than $n^3$).
  - Try to close the gap by increasing the lower bound and/or decreasing the upper bound.
  - It is ideal to have matching upper and lower bounds, e.g., if we show there is $O(n^2)$ algorithm and no algorithm runs in $O(n^{2-\epsilon})$, the upper and lower bounds are almost-matched (we call them tight upper and lower bounds), i.e., we know the time complexity of the best possible algorithm.
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.
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., \( S = \{0.1, 0.2, 0.2, 0.3, 0.3, 0.4, 0.4, 0.5, 0.5, 0.5, 0.5, 0.6, 0.8, 0.8, 0.9\} \)
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.
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?
- We show this problem is NP-complete even for the easy case of $k = 2$.
- Easy 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.
  - reduction from the partition problem

Partition: decide whether a multiset $P$ of positive integers can be partitioned into two subsets $S$ and $P - S$ s.t. sum of the numbers in $S = \text{sum of the numbers in } P - S$

- $P = \{3, 1, 3, 2, 3, 2, 3, 3, 4, 1\} \rightarrow S = \{3, 2, 3, 3\} \quad P - S = \{1, 3, 2, 4, 1\}$

Partition is NP-complete, i.e., assuming $P \neq NP$ there is no algorithm that runs in $O(n^c)$ for an input of length $n$.

- 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.
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 bin 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.

- 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., $S = \bigcup_{q_i \in R} \{p_i\}$.
- 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.
  - It is possible to partition $P$ into two groups, each of total sum $t/2$ if and only if it is possible to partition $Q$ into two groups, each of total sum 1.
  - 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.**
Recall that bin packing is an optimization problem (it asks for \textbf{minimizing} the number of bins for a given multi-set of items).

Since it is NP-complete, there is no polynomial algorithm for find the smallest number of bins assuming $P \neq NP$.

We can \textbf{approximate} the solution!

The solution provided by an approximation algorithm is not necessarily optimal (e.g., the best possible packing) but an approximation of that (e.g., a packing which opens 1.7 times bins that an optimal algorithm does).

We review approximation algorithms next week!
Complexity classes

- class P: decision problems which can be answered in $O(n^c)$.
- class NP: decision problems for which a candidate answer (also known as a certificate) can be checked in $O(n^c)$.
- We have $P \subseteq NP$: if you can answer a problem in polynomial time, you can check a candidate solution (certificate) in polynomial time as well.
- class EXP: decision problems which can be answered in $O(2^{n^c})$.
- Is it that $P \subseteq EXP$?
  - Yes, if you can answer a problem in $O(n^c)$, you have answered it in $O(2^{n^c})$. 
Complexity classes

- Is it that $NP \subseteq EXP$?
  - Consider a problem $P$ which belongs to NP.
  - Since we could check a certificate (candidate solution) in polynomial time, the length of any candidate solution is $O(n^c)$ for some constant $c$ (otherwise, we could not even read the certificate in polynomial time).
  - The number of candidate solutions is hence $O(2^{n^c})$.
    - E.g., for Hamiltonian path, there are
      $$n! = n \cdot n \ldots n < 2^n \cdot 2^n \ldots 2^n = 2^{n^2}$$
      candidate solutions.
  - We can exhaustively check all candidate solutions; there are $O(2^{n^c})$ of them, and checking each requires $O(n^{c'})$ (by definition of NP).
  - Total time to answer an NP problem can be bounded as
    $$2^{n^c} \cdot O(n^{c'}) \in O(2^{n^c} \cdot 2^{n^{c'}}) \in O(2^{n^{\max\{c,c'\}}}) \rightarrow \text{an NP problem belongs to EXP class of problems.}$$

- Is it that $EXP \subseteq NP$ (i.e., is it that $EXP = NP$).
  - No one knows yet; but it seems likely that $EXP \neq NP$. 
Complexity classes

- **EXP-Complete Problems**: Decision problems which belong to EXP (can be solved in exponential time) and all EXP problems reduce to them in polynomial time.
  - These are the hardest EXP problems
  - EXP-Complete problems cannot be solved in polynomial time (It is a consequence of ‘time hierarchy theorem’)
  - Example: Bounded Halting decision problem: does a program running on a computer (precisely a Turing machine) halts (finishes) after $k$ steps?
    - A simulation requires time linear to $k$ while the input (value of $k$) can be encoded in $\Theta(\log k)$ bits.
  - Examples: generalized chess on $n \times n$ board, checkers, and Go (with Japanese rules).
    - Decision problem for these games is whether a given configuration (position of pieces of the board), there is a strategy for white s.t. it always wins regardless of how black plays.
    - The decision problem can be solved in constant time for classic chess (everything is constant; there is no $n$). But the constant is so huge that our hardware cannot approach it yet.
Complexity Classes Review

- What we know: $P \subseteq NP$ and $NP \subseteq EXP$.
- What we don’t know: $NP \subseteq P$? and $EXP \subseteq NP$?.
- We also know that $EXP \not\subseteq P$
  - EXP-complete problems cannot be solved in polynomial time (time hierarchy theorem)
- We can conclude at least one of the following statements are correct: $NP \not\subseteq P$, $EXP \not\subseteq NP$
  - The general belief is that both statements are correct!