# Unit 7.2 Branch and Bound

Algorithms

EE3980

May 14, 2018

---

## 0/1 Knapsack Problem

- Given $n$ objects, each with profit $p_i$ and weight $w_i$, and a sack of maximum weight $m$, select the objects to be placed into the sack such that the profits of the objects in the sack is maximum. (Note that the object must be placed as a whole, no fraction, into the sack.)
- Recall that the greedy algorithm that allows the fraction of an object to be placed into the sack generate the optimal solution (maximal profits).

### Algorithm 4.1.5. Knapsack

```
Algorithm Knapsack($m, n, w, p, x$)
// $n$ objects with $w[i]$ and $p[i]$ find $x[i]$ that maximizes $\sum p_i x_i$ with $\sum w_i x_i \le m$.
{
    $a :=$ Sort$(p/w)$ ; // sort $p[a[i]]/w[a[i]]$ into non-increasing order.
    for $i := 1$ to $n$ do $x[i] := 0$ ;
    $i := 1$ ;
    while ($i \le n$ and $w[a[i]] \le m$) do {
        $x[i] := 1$ ;
        $m := m - w[a[i]]$ ;
        $i := i + 1$ ;
    }
    if ($i \le n$) then $x[i] := m/w[a[i]]$ ;
}
```

# 0/1 Knapsack Problem, Bounds

- Note that on line 12 the last object included might be a fraction which violate the requirement of a whole object.
- Thus, excluding this line the profit $p = \sum_{j=1}^{i} p_j$ is the least one can get for the profit.
    - We can use this $p$ as a lower bound ($lb$)for the profits.
- The profits, $P$, with the fraction object is the maximum and can be used as the upper bound ($ub$).
- Thus, assuming the objects are ordered by $p/w$, the following function generates two bounds for the set of objects

# 0/1 Knapsack Problem, Bounds Algorithm

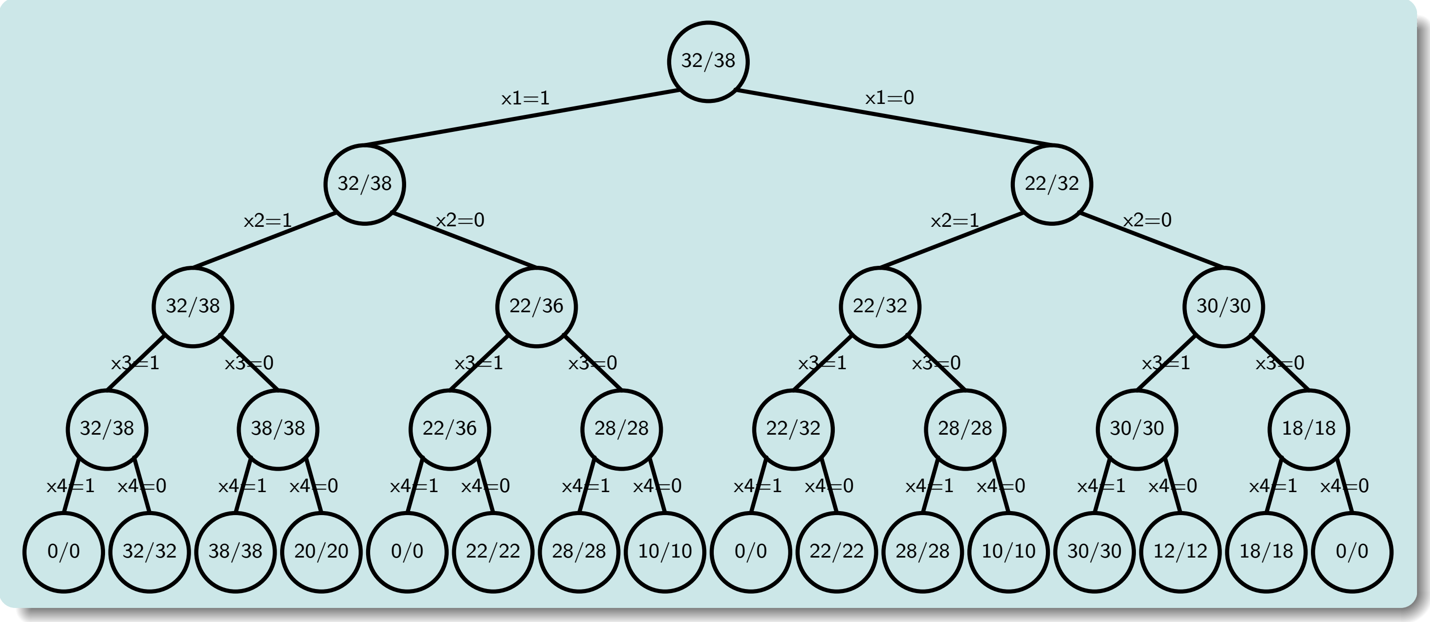
## Algorithm 7.2.1. Bounds

```
Algorithm Bounds($k, cw, cp, lb, ub$)
// Estimate two bounds $lb$ and $ub$ for $n$-object 0/1 knapsack problem
{
    $i := k + 1$ ; $lb := cp$ ;
    while ($i \leq n$ and $cw \leq m$) do {
        $lb := lb + p[i]$ ; $cw := cw + w[i]$ ; $i := i + 1$ ;
    }
    if ($i > n$) $ub := lb$ ;
    else $ub := lb + (1 - (cw - m)/w[i]) * p[i]$ ;
}
```

- The above algorithm has been generalized such that the decision on the first $k$ objects have been made and $cp$ and $cw$ are the current profits and weights for the first $k$ objects.
- The algorithm estimate the two bounds for the remaining $n - k$ objects.
- Note that arguments $lb$ and $ub$ need to be passed by reference (in `C++`), or passed by pointer (in `C`).

# 0/1 Knapsack Problem Example

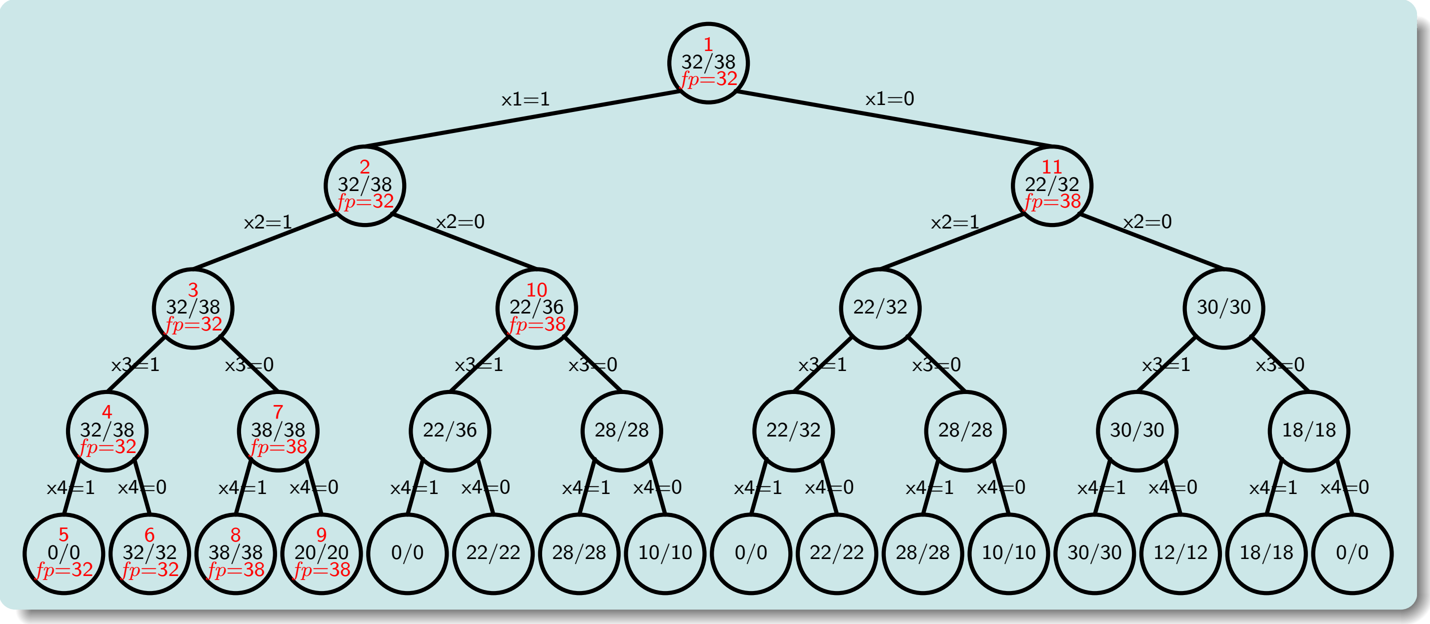
- 0/1 knapsack problem example:
  $n = 4,\ p = (10, 10, 12, 18),\ w = (2, 4, 6, 9),\ m = 15.$
- Complete state space can be shown to be



---

# 0/1 Knapsack Problem Example — Depth First

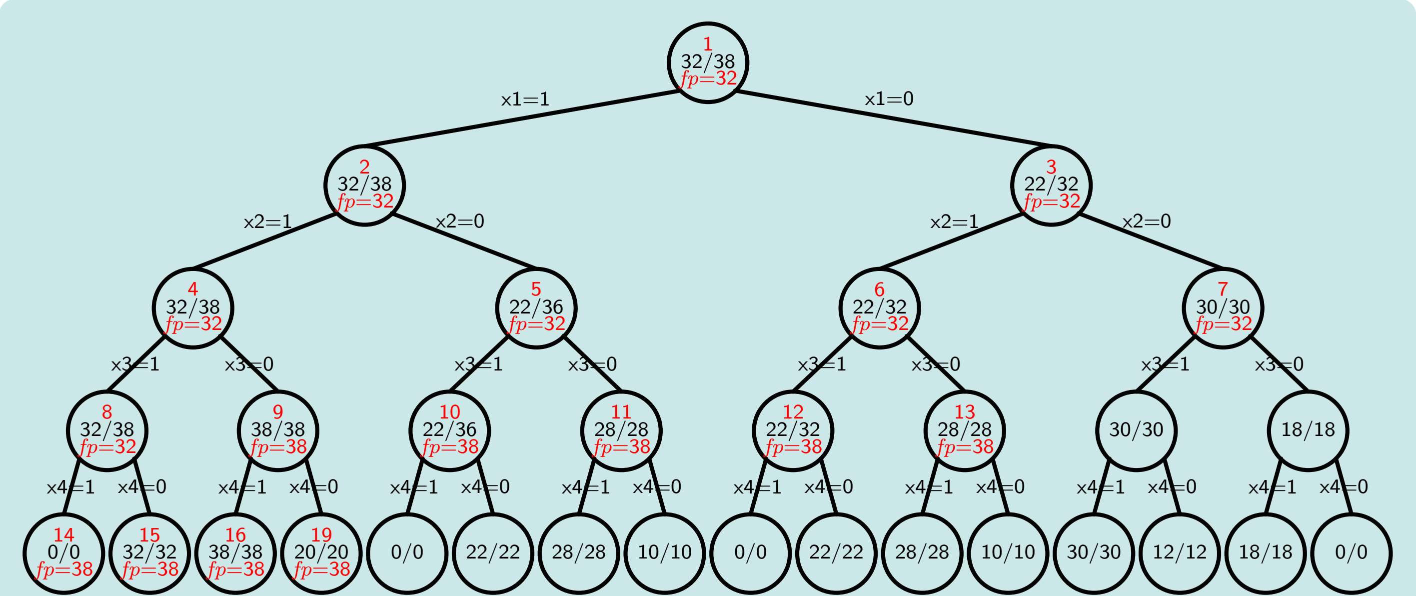
- 0/1 knapsack problem example:
  $n = 4,\ p = (10, 10, 12, 18),\ w = (2, 4, 6, 9),\ m = 15.$
- Using depth-first traversal branch and bound approach, we have



- Branch and bound stops after 11 steps
- The solution is $x = (1, 1, 0, 1),\ fp = 38,\ fw = 15.$

# 0/1 Knapsack Problem Example — Breadth First

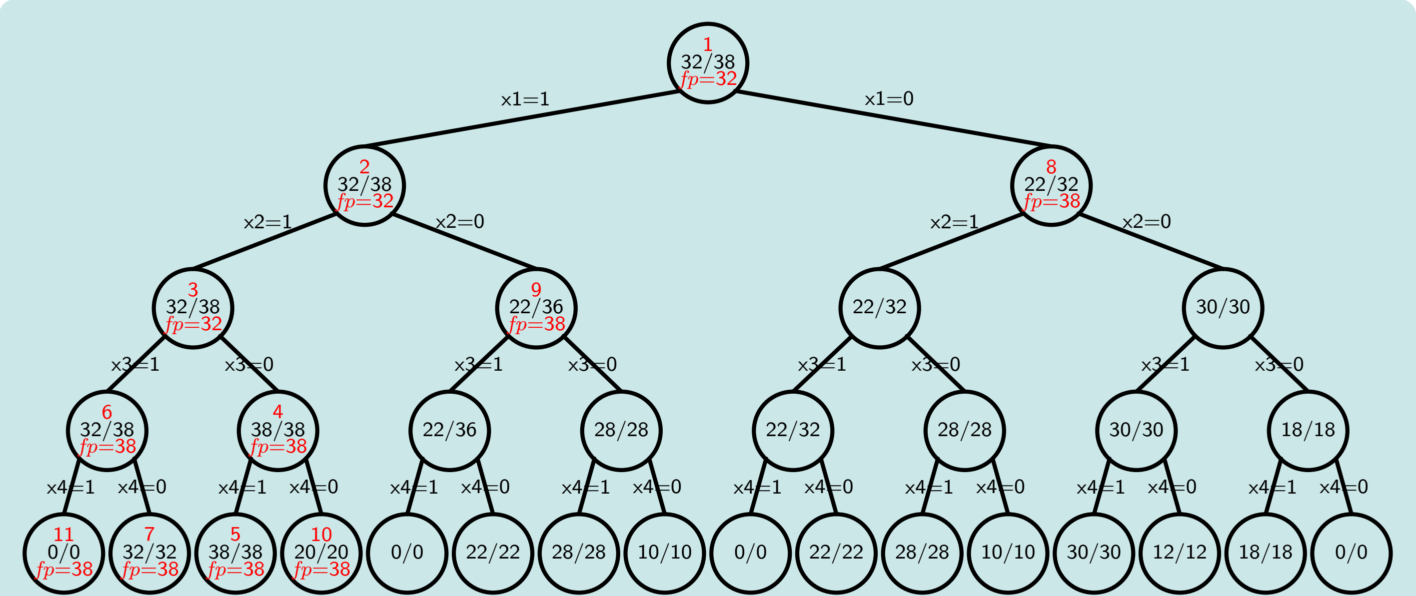
- 0/1 knapsack problem example:
  $n = 4,\ p = (10, 10, 12, 18),\ w = (2, 4, 6, 9),\ m = 15.$
- Using breadth-first traversal branch and bound approach, we have



- Branch and bound stops after 19 steps
- The solution is $x = (1, 1, 0, 1),\ fp = 38,\ fw = 15.$

# 0/1 Knapsack Problem Example — Least Cost

- 0/1 knapsack problem example:
  $n = 4,\ p = (10, 10, 12, 18),\ w = (2, 4, 6, 9),\ m = 15.$
- Using Least-cost branch and bound approach, we have



- Branch and bound stops after 11 steps
- The solution is $x = (1, 1, 0, 1),\ fp = 38,\ fw = 15.$

## Branch and Bound Algorithms

- Branch and bound method is applicable to all state space search methods.
  - All children of a search node are generated before any other live node is explored.
  - Bounding functions are used to help reducing the number of subtrees to be explored.
- Two tree traversal algorithms are applicable to explore the state space.
  - Breadth-first search: also known as first-in-first-out (FIFO) strategy.
    - Need a stack to keep the live nodes.
  - Depth-first search: also known as the last-in-first-out (LIFO) strategy.
- An additional strategy least cost search has been introduced.
  - Each node is associated with a cost that estimates the solution cost.
  - To select the next node to explore, select one with the least cost.
- The following algorithm is a high level description of the LC-search approach.
- The LC-search algorithm uses the following `structure`.

```
struct listnode {
      double cost , lb , ub ; // cost and estimated lower and upper bounds
      struct listnode *next, *parent;
}
```

---

## Branch and Bound Algorithms — LC Search

### Algorithm 7.2.2. LC Search

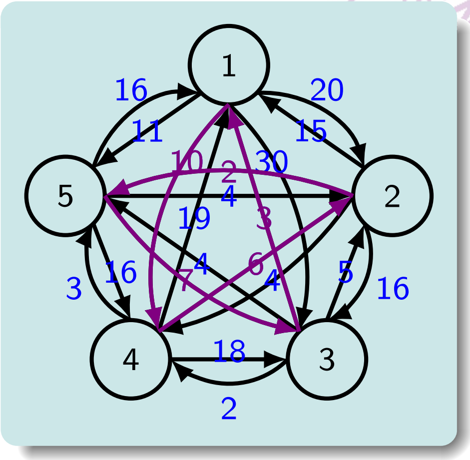
```
Algorithm LCSearch($t$)
// General framework for least cost search.
{
      if $t$ is an answer node then { write ($t$); return ; }
      $E := t$; // Current search node.
      Initialize the list of live nodes to be empty ;
      while ($E \neq \emptyset$) do {
            for each child $x$ of $E$ do {
                  if $x$ is an answer node then { write ( path from $x$ to $t$); return ; }
                  Add($x$); // $x$ is a new live node.
                  $x \rightarrow parent := E$;
            }
            if there are no live nodes then { write ("No answer."); return ; }
            $E :=$ Least();
      }
}
```

## Branch and Bound Algorithms — General

- In the above algorithm, two functions are used
  - `Add`: add a new live node to the list.
  - `Least`: find the minimum cost node from the live node list and remove it from the list.
- The `list` data structure is used for `LCS` for searching of least cost node is needed. In contrast,
  - DFS uses `stack` (LIFO),
  - BFS uses `queue` (FIFO).
  - Selecting the next live node is more consuming in `LCS` approach.
- All three search approaches can be used in branch-and-bound method.
- For each $E$-node, in addition to the cost $c$ two more estimates are calculated: a lower bound $lb$ and an upper bound $ub$.
- In exploring each node, the best cost $fc$ is also tracked.
- Thus, when exploring node $E$ if $lb > fc$ then there is no need to traverse the subtree of $E$.
  - And, in selecting $E$ node, the one with the minimum $lb$ should be selected.
- By reducing the number of subtrees to be explored, the branch-and-bound algorithm can be fast.

## Traveling Salesperson Problem

- Let $G = (V, E)$ be a directed graph, with $|V| = n$ and $c_{ij}$ be the cost of edge $\langle i, j\rangle \in E$, $c_{ij} = \infty$ if $\langle i, j\rangle \notin E$.
- Without loss of generality, we can assume every tour start from vertex 1. So, the solution space is $S = \{1, \pi, 1 | \pi$ is a permutation of $(2, 3, \cdots, n)$.
- Of course, for any solution $(1, i_1, i_2, \cdots, i_{n-1}, 1) \in S$, $\langle i_j, i_{j+1}\rangle \in E$, $0 \le j \le n-1$ and $i_0 = i_n = 1$.
- The objective is to find a path with the minimum cost.
- Traveling salesperson problem example



- Cost matrix

$$\begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix}$$

# Traveling Salesperson Problem — Reduced Cost Matrix

- Given a cost matrix, it can reduced as following.
- Note that $c_{i,j}$ is the cost from vertex $i$ to vertex $j$
  - Thus, if $c_{i,k} = \min_{j=1}^{n} c_{i,j}$, then $c_{i,k}$ is the minimum cost leaving vertex $i$.
  - And, if $c_{k,j} = \min_{i=1}^{n} c_{i,j}$, then $c_{k,j}$ is the minimum cost entering vertex $j$.

- Original cost matrix

$$\begin{array}{cl} \begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix} & \begin{array}{l} \text{row } 1 - 10 \\ \text{row } 2 - 2 \\ \text{row } 3 - 2 \\ \text{row } 4 - 3 \\ \text{row } 5 - 4 \end{array} \end{array}$$

- Row- and Column-reduced cost matrix

$$\begin{bmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix}$$

- Row-reduced cost matrix

$$\begin{bmatrix} \infty & 10 & 20 & 0 & 1 \\ 13 & \infty & 14 & 2 & 0 \\ 1 & 3 & \infty & 0 & 2 \\ 16 & 3 & 15 & \infty & 0 \\ 12 & 0 & 3 & 12 & \infty \end{bmatrix}$$

- Column 1 is reduced by 1, and column 3 reduced by 3 are performed.
- The total reduction $R = 10+2+2+3+4+1+3 = 25$ is the lower bound for for the salesperson traveling problem.

# Traveling Salesperson Problem — Reduced Cost Matrix, II

- The technique of reduced cost matrix to estimate the lower bound of the traveling salesperson problem can be extended to estimating path selection.
- Suppose an edge $\langle i, j \rangle$ is selected, the cost of the path is increased by $c_{i,j}$
  - All other edges $\langle i, k \rangle$, $k \neq j$ cannot be selected. Thus, set $c_{i,k} = \infty$, $1 \le k \le n$. (Row $i$)
  - All edges $\langle k, j \rangle$, $k \neq i$, cannot be selected. Thus, set $c_{k,j} = \infty$, $1 \le k \le n$. (Column $j$)
  - The edge $\langle j, 1 \rangle$ cannot be selected (unless $j$ is the only vertex not selected). Thus, set $c_{j,1} = \infty$.
  - Perform reduced matrix technique to the resulting matrix to get the lower bound, $r$.
  - Then the lower bound of path cost of selecting edge $\langle i, j \rangle$ is $R + c_{i,j} + r$, where $R$ is the lower bound before selecting edge $\langle i, j \rangle$.

# Traveling Salesperson Problem — Reduced Cost Matrix, III

- Example

- Original cost matrix

$$\begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix}$$

- Selecting edge $\langle 1, 3 \rangle$

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 4 & 3 & \infty & \infty & 0 \\ 0 & 0 & \infty & 12 & \infty \end{bmatrix}$$

- Cost-reduced cost matrix, $R = 25$.

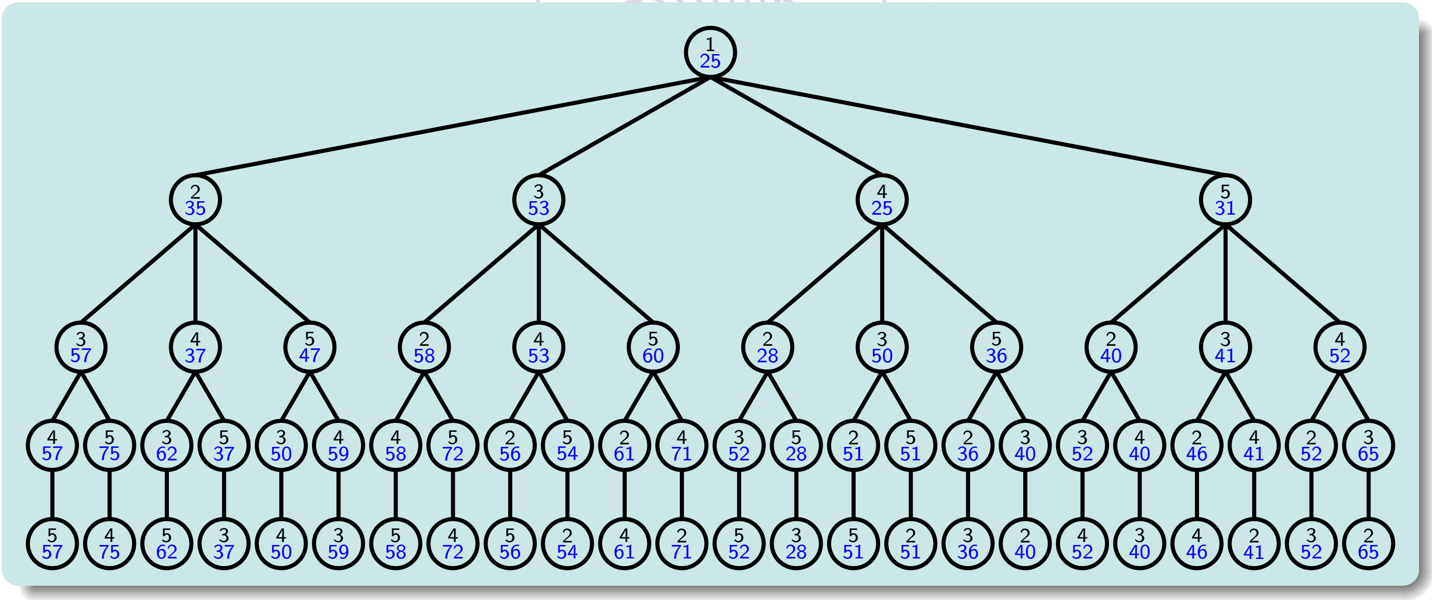
$$\begin{bmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix}$$

- $c_{1,3} = 17$.
- Row 1 is set to $\infty$
- Column 3 is set to $\infty$
- $c_{3,1}$ is set to $\infty$
- Then column 1 can be reduced by 11. $(r = 11)$
- The lower bound for selecting edge $\langle 1, 3 \rangle$ is $R + c_{1,3} + r = 25 + 17 + 11 = 53$.

---

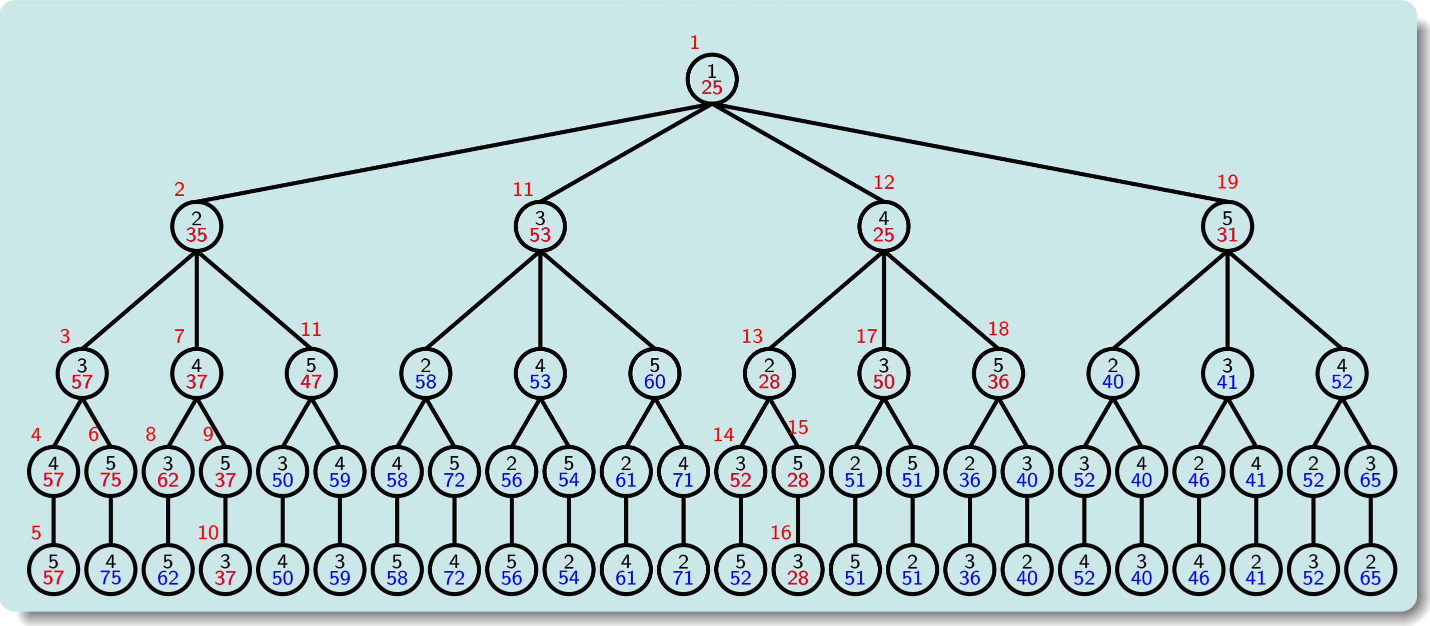
# Traveling Salesperson Problem

- The full state space is shown below



# Traveling Salesperson Problem — Depth-First Search BB

- Using depth-first search with BB, we have

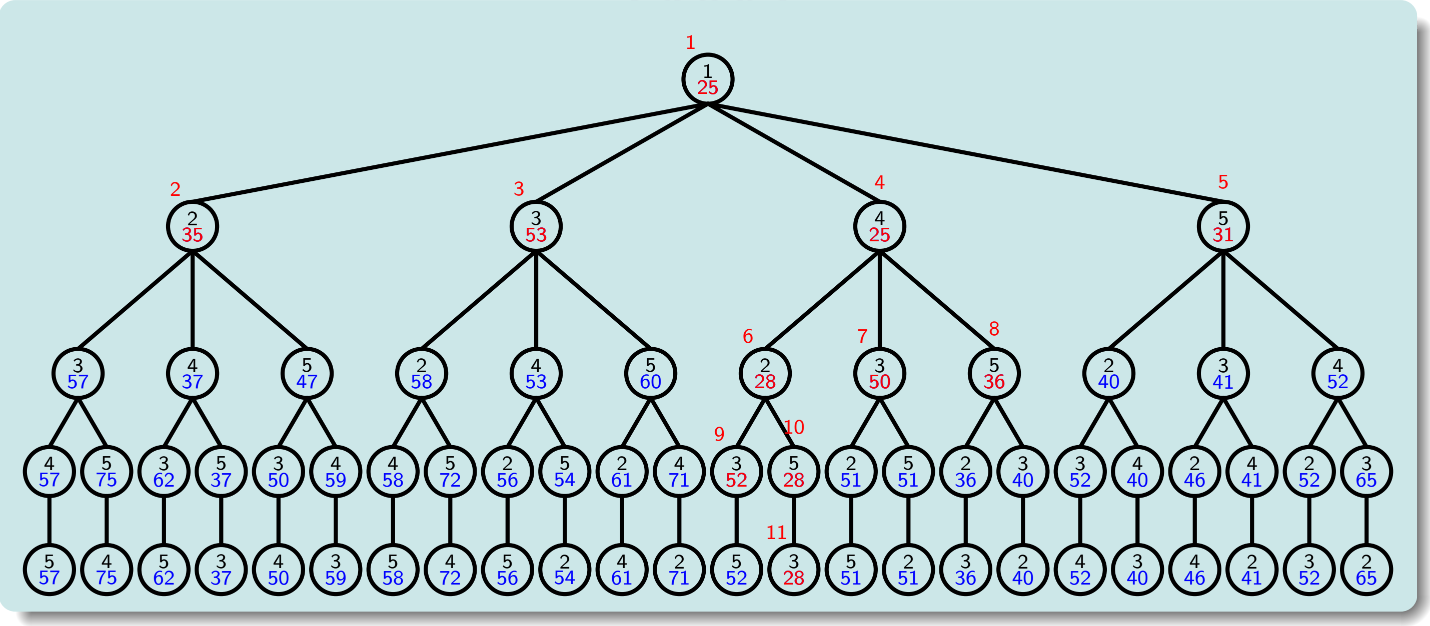


- DFS BB stops in 19 steps
- The solution is $1-4-2-5-3-1$, total cost is 28.

---

# Traveling Salesperson Problem — Least-Cost Search BB

- Using least-cost search with BB, we have



- LCBB stops in 11 steps
- The solution is $1-4-2-5-3-1$, total cost is 28.

# Theories

- Some theories concerning branch-and-bound approaches.

**Theorem 7.2.3.**

Let $t$ be a state space tree. The number of nodes of $t$ generated by FIFO, LIFO and LC branch-and-bound algorithms cannot be decreased by the expansion of any node $x$ with $lb(x) >= upper$, where $upper$ is the upper bound on the cost of a minimum-cost solution node in the tree $t$.

**Theorem 7.2.4.**

Let $U_1$ and $U_2$, $U_1 < U_2$, be two initial upper bounds on the cost of a minimum-cost solution node in the state space tree $t$. The FIFO, LIFO, and LC branch-and-bound algorithms beginning with $U_1$ will generate no more nodes than they would if they started with $U_2$ as the initial upper bound.

**Theorem 7.2.5.**

The use of a better lower bound function $lb$ in conjunction with FIFO and LIFO branch-and-bound algorithms does not increase the number of nodes generated.

---

# Theories, II

**Theorem 7.2.6.**

If a better lower bound function is used in a LC branch-and-bound algorithm, the number of nodes generated may increase.

**Theorem 7.2.7.**

The number of nodes generated during FIFO and LIFO branch-and-bound search for a least-cost solution the number of nodes generated may increase when a stronger dominance relation is used.

**Theorem 7.2.8.**

Let $D_1$ and $D_2$ be two dominance relations. Let $D_2$ be stronger than $D_1$ such that $(i, j) \in D_2$, $i \neq j$, implies $lb(i) < lb(j)$. An LC branch-and-bound using $D_1$ generates at least as many nodes as the one using $D_2$.

# Branch and Bound

- Branch and bound methods belong to the all state space search method.
- To avoid extensive searching of all states, bounding functions for lower bound and upper bound are keys.
- Accurate bounding functions can decrease the state space that needs to be searched.
- Three traversal techniques can be used to explore the state space
  – depth first search, breadth first search and least cost search.
- Least cost searching is shown to be effective in some problems.
- With good bounding function and effective traversal method, branch and bound can solve real problems with significant time saving.

# Summary

- 0/1 knapsack problem
- Branch-and-bound algorithms
- Least-cost branch-and-bound
- The salesperson traveling problem
- Theories on branch-and bound algorithms

# Unit 8. Lower Bound Theory

Algorithms

EE3980

May 16, 2018

---

## Lower Bounds

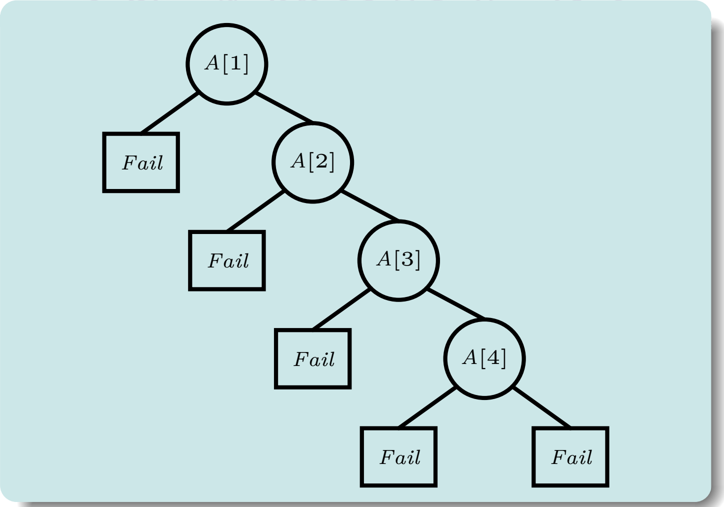
- Given a problem, one can device algorithms to solve the problem.
  - Once an algorithm is developed, we know how to analyze the time and space complexity.
  - Over all the algorithms, the one with the minimum complexity is usually preferred.
  - If we know the lower bound of a given problem, then we can strive to solve it with the lowest complexity possible.
- Some problems have been studied extensively and the results are listed in this unit.
- Lower bounds for searching and sorting algorithms are studied first.

# Ordered Searching

- Comparison based complexity analysis is assumed.
- To find $x$ in an ordered array $A[i]$, $A[i] < A[j]$ if $i < j$.
- A series of comparisons are to be performed.
- Each comparison can have one of three results:

$$x < A[i],\ x = A[i],\ \text{or}\ x > A[i].$$

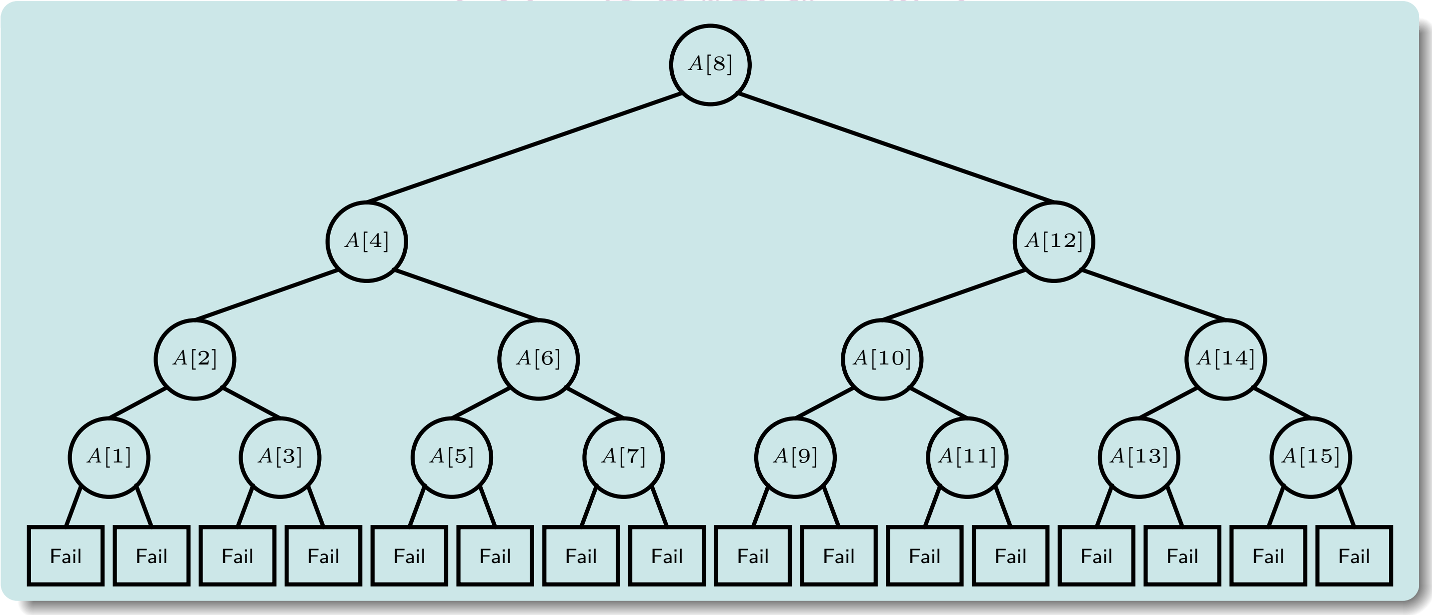
- Array $A$ can be stored as a tree.
- A linear search is shown below.
    - The worst-case complexity is $\mathcal{O}(n)$.



---

# Ordered Searching, II

- A binary search tree is shown below.
- For any array of $n$ elements, there are $n+1$ possible fails.
- If there are $k$ levels in the tree, then there are at most $2^k - 1$ internal nodes.
- Therefore, for an array with $n$ elements for the tree with $k$ levels, $n \le 2^k - 1$, or $k \ge \lg(n+1)$.



# Ordered Searching, III

## Theorem 8.1.1.

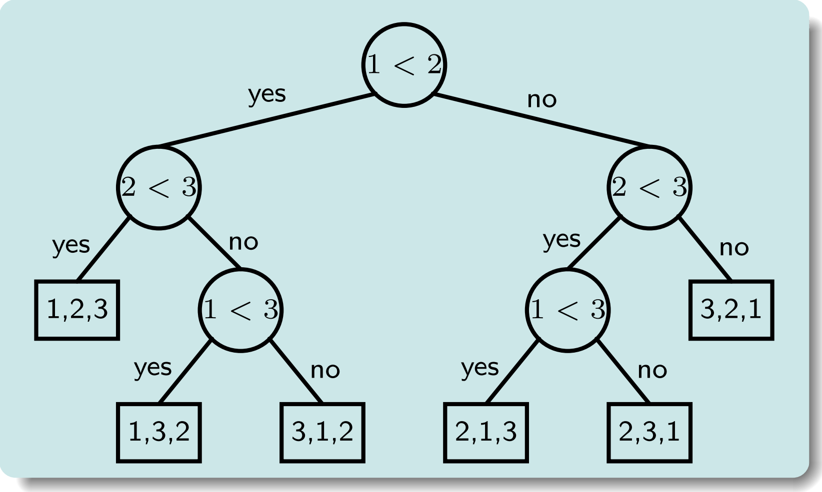
Let $A[1:n]$, $n \geq 1$, contains $n$ distinct elements, ordered so that $A[1] < A[2] < \cdots < A[n]$. Let $\texttt{FIND}(n)$ be the minimum number of comparisons needed, in the worst case, by any comparison-based algorithm to recognize whether $x \in A[1:n]$. Then $\texttt{FIND}(n) \geq \lceil \lg(n+1) \rceil$.

- As a consequence of this algorithm, the binary search algorithm is an optimal worst-case algorithm for the ordered searching problem.

---

# Sorting

- Given an array $A[1:n]$ with all elements distinct. The sorting problem is to rearrange the array $A$ such that $A[i] < A[j]$, if $1 \leq i < j \leq n$.
- An example of sorting 3-integer array, $\{1, 2, 3\}$, is shown below.
  - Each internal node performs a comparison, $A[i] < A[j]$.
    - The comparison can have only two results: `true` or `false`.
  - Each external node represents one of the possible sorting results.
    - With 3 elements, there are $6 = 3!$ external nodes.



# Sorting — Lower Bound

- Given $A[1:n]$, the comparison based algorithm should have a state space with $n!$ external nodes, and these external nodes are the leaves of the binary tree.
- Assuming that the binary tree has $k$ levels, it takes $k$ comparisons to perform the sorting algorithm.
- Let $T(n)$ be the minimum number of comparisons to sort $A[1:n]$, then

  $$2^{T(n)} \geq n!$$

  And

  $$T(n) \geq \lceil \lg n! \rceil$$

  By Stirling's approximation

  $$\lg n! = n \lg n - n/(\lg 2) + (\lg n)/2 + \mathcal{O}(1)$$

- Thus, any comparison-based sorting algorithm needs at least $\Omega(n \lg n)$ time.

# Sorting Complexity Example — Merge Sort

- Merge sort starts by comparing two elements to form $n/2$ groups of 2 elements.
- Then two two-element groups are sorted.
  - 3 comparisons are needed to form $n/4$ groups.
- The next step compares 4-element groups to form $n/8$ groups.
  - 7 comparisons are needed to sort two 4-element groups.
- Thus, the total number of comparisons is

  $$T(n) = \sum_{i=1}^{k} \frac{n}{2^i}(2^i - 1) = \sum_{i=1}^{k} n - n \sum_{i=1}^{k} \frac{1}{2^i}$$

  where $k = \lg n$.
- Thus, $T(n) = n \lg n - \mathcal{O}(n)$.
- Merge sort achieves the lowest time complexity, but the coefficients can still be improved.
  - See textbook [Horowitz], pp. 481-483.

# Merging

- Given two ordered arrays $A[1:m]$ and $B[1:n]$, a third ordered array $C[1:m+n]$ is formed by merging these two arrays together.
- Given the numbers $m$ and $n$, there are $\binom{m+n}{n}$ combinations of possibilities combining $A[1:m]$ and $B[1:n]$.
- Using comparison based algorithms, a tree can be formed and there should be at least $\binom{m+n}{n}$ external nodes.
- Let $\texttt{MERGE}(m,n)$ be the minimum number of comparisons to merge $A[1:m]$ and $B[1:n]$, then

$$\texttt{MERGE}(m,n) \geq \left\lceil \lg \binom{m+n}{n} \right\rceil.$$

- It has been shown in Unit 3 that the upper bound of $\texttt{MERGE}(m,n)$, thus

$$\left\lceil \lg \binom{m+n}{n} \right\rceil \leq \texttt{MERGE}(m,n) \leq m+n-1.$$

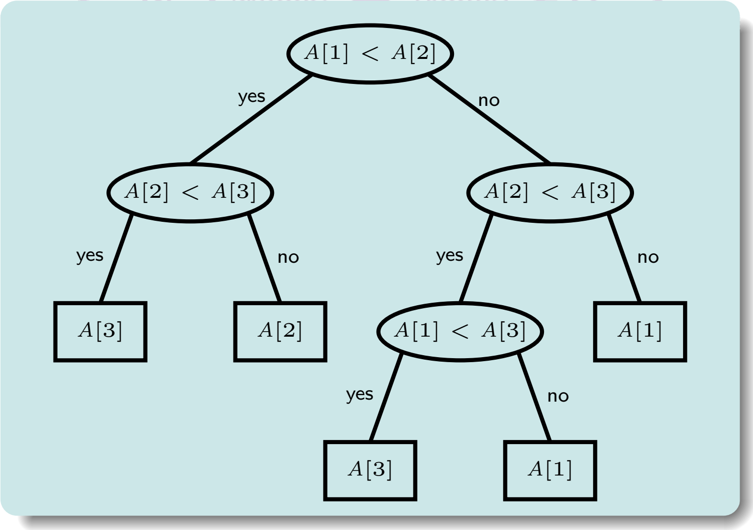
- A special case when $m = n$

**Theorem 8.1.2.**

$\texttt{MERGE}(m,m) = 2m-1$, for $m \geq 1$.

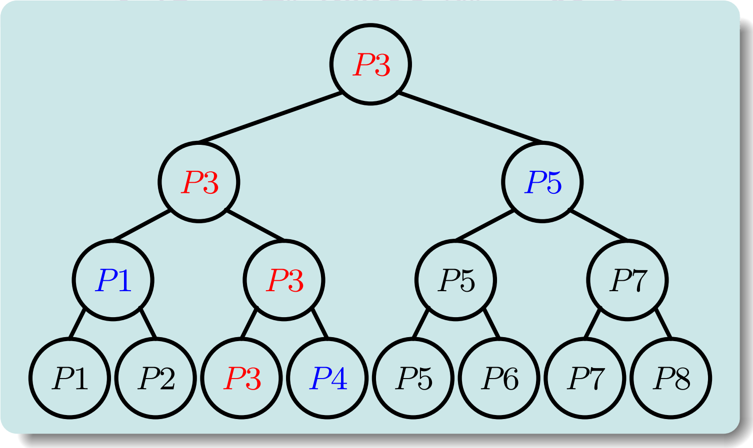
# Finding the Largest Element

- To find the largest element of an $n$-element array $A$, there must be at least $n-1$ nodes in the tree.
  - After $k$ comparisons, only one element remains that is greater than any other element. The smallest $k$ is $n-1$.
- Thus, the minimum number of comparisons for finding the largest elements of an $n$-element array is $L_1(n) = n-1$.
- Example of the comparison tree of finding the largest element of a 3-element array, $A[1:3]$.



# Largest and 2nd Largest

- Given an unordered set $A[1:n]$, finding the largest element needs $n-1$ comparison.
- The comparison tree can be arranged as the following.



---

# Largest and 2nd Largest, II

- To find the 2nd largest element, one needs to compare only those elements that compared to the largest element and were found to be smaller.
  - There are only $\lg n$ such elements.
  - To find the largest among them needs $\lg n - 1$ comparison.
- Thus to find the largest and second largest elements needs $n + \lg n - 2$ comparisons.

**Theorem 8.1.3.**

Any comparison-based algorithm that computes the largest and the second largest element of a set of $n$ unordered elements requires $n - 2 + \lceil \lg n \rceil$ comparisons.

# The Largest to the $k$-th Largest Elements

- The comparison tree of finding the largest to the $k$-th largest elements of $A[1:n]$ needs to have $n \cdot (n-1) \cdots (n-k+1)$ external nodes.
- Thus, let $L_k(n)$ be the minimum number of comparisons of finding the largest to the $k$ largest elements

$$L_k(n) \ge \left\lceil \lg \left( n \cdot (n-1) \cdots (n-k+1) \right) \right\rceil .$$

- More detailed analysis shows that

**Theorem 8.1.4.**

$L_k(n) \ge n - k + \left\lceil \lg \left( n \cdot (n-1) \cdots (n-k+2) \right) \right\rceil$ for all integers $k$ and $n$, where $1 \le k \le n$.

- Note that this is an estimate of the lower bound.

---

# Find the Largest $k$ elements

**Theorem 8.1.5.**

Given an unordered set with $n$ elements, the $(k-1)$th largest element itself needs at least $(k-1)\left\lceil \lg \dfrac{n}{2(k-1)} \right\rceil$ comparisons to be identified.

- Proof please see textbook [Horowitz], p. 491.

**Theorem 8.1.6.**

Given an unordered set with $n$ elements, all $k-1$ largest elements can be found with at least $n - k + (k-1)\left\lceil \lg \dfrac{n}{2(k-1)} \right\rceil$ comparisons.

- Proof please see textbook [Horowitz], pp. 491-492.

# Finding the Maximum and Minimum

- Given $n$ distinct elements, find the maximum and the minimum.
- Using comparison-based algorithms, define 4-tuple $(a, b, c, d)$ as
  - $a$ is the number of elements that have not been compared,
  - $b$ is the number of elements that have won and never lost,
  - $c$ is the number of elements that have lost and never won,
  - $d$ is the number of elements that have both won and lost.
- Then given a state $(a, b, c, d)$, an additional comparison can result in one of the following states:

| | | |
|---|---|---|
| $(a-2, b+1, c+1, d)$ | if $a \geq 2$ | // Compare two items from $a$. |
| $(a-1, b+1, c, d)$<br>$(a-1, b, c+1, d)$<br>$(a-1, b, c, d+1)$ | if $a \geq 1$ | // Compare one item from $a$<br>// with one item from $b$<br>// or from $c$. |
| $(a, b-1, c, d+1)$ | if $b \geq 2$ | // Compare two items from $b$. |
| $(a, b, c-1, d+1)$ | if $c \geq 2$ | // Compare two items from $c$. |

# Finding the Maximum and Minimum, II

- The initial state is $(n, 0, 0, 0)$ since all elements have not been compared.
- Then it takes $n/2$ comparisons, comparing elements in $a$, to move to the state $(0, n/2, n/2, 0)$.
- The final state is $(0, 1, 1, n-2)$ since we want to find the maximum, only one element left in $a$, and the minimum, only one element left in $b$, the rest elements must be in $d$.
  - The minimum number is $n-2$ since $d$ can only be increased by 1 with each comparison.

## Theorem 8.1.7.

Any algorithm that computes the largest and the smallest elements of a set of $n$ unordered elements requires $\lceil 3n/2 \rceil - 2$ comparisons.

# Problem Reduction

## Definition 8.1.8. Problem reduction.

Let $P_1$ and $P_2$ be any two problems. We say $P_1$ reduces to $P_2$, denoted by $P_1 \propto P_2$, in time $\tau(n)$ if an instance of $P_1$ can be converted into an instance of $P_2$ and solution for $P_1$ can be obtained from a solution of $P_2$ in time $\le \tau(n)$.

- Example
  - $P_1$ is the problem of selection (Finding the $k$th smallest element.)
  - $P_2$ is the problem of sorting.
  - If the input have $n$ numbers and the number are sorted in an array $A[1:n]$,
  - The $k$th smallest element of the input can be obtained as $A[k]$.
  - Thus, $P_1$ reduces to $P_2$ in $\mathcal{O}(1)$ time.
- Note there are three steps in this formulation
  - Convert the inputs of problem $P_1$ to $P_2$
    - In this example, no special action is required.
  - Solve problem $P_2$.
    - $\mathcal{O}(n \lg n)$ if comparison based algorithm is adopted.
  - Convert the solution of $P_2$ to that of $P_1$.
    - $\mathcal{O}(1)$ since $A[k]$ is the solution of $P_1$.

---

# Problem Reduction, II

- Example 2
  - Given two sets $S_1$ and $S_2$ with $m$ elements each.
  - $P_1$ is the problem to check if $S_1$ and $S_2$ are disjoint, i.e., $S_1 \cap S_2 = \emptyset$.
  - $P_2$ is the sorting problem.
  - Then $P_1 \propto P_2$ in $\mathcal{O}(m)$ time.

  - Let $S_1 = \{k_1, k_2, \cdots, k_m\}$ and $S_2 = \{h_1, h_2, \cdots, h_m\}$, then we can create a set $X = \{(k_1, 1), (k_2, 1), \cdots, (k_m, 1), (h_1, 2), (h_2, 2), \cdots, (h_m, 2)\}$.
  - This $X$ can be created in $2m$ time ($\mathcal{O}(m)$).
  - Then $X$ can be sorted by the first element of each tuple.
    - $\mathcal{O}(n \lg n)$, $n = 2m$, if comparison-based method is used.
  - After sorting, we can check whether there are two successive elements $(x, 1)$ and $(y, 2)$ such that $x = y$.
    - $2m - 1$ comparisons are needed ($\mathcal{O}(m)$).
  - If there are no such elements, then $S_1$ and $S_2$ are disjoint; otherwise they are not.

# Lower Bounds Through Reductions

- Given two problems $P_1$ and $P_2$ such that $P_1$ reduces to $P_2$ in $\tau(n)$,
  - The input of $P_1$ is converted to the input of $P_2$ and the solution is obtained from $P_2$ in $\tau(n)$.
  - Suppose problem $P_1$ can be solved in time $T_1(n)$ and
  - Problem $P_2$ can be solved in time $T_2(n)$, then

$$T_1(n) \le \tau(n) + T_2(n). \tag{8.1.1}$$

Or,

$$T_2(n) \ge T_1(n) - \tau(n). \tag{8.1.2}$$

- Thus, the lower bound for solving problem $P_2$ is $T_1(n) - \tau(n)$.

# Finding Convex Hull

- Let $P_1$ be a sorting problem on $n$ numbers.
  - $T_1(n) = \mathcal{O}(n \lg n)$.
- These numbers can be transformed into $n$ points on a 2-D plane as $\{(k_1, k_1^2), (k_2, k_2^2), \cdots, (k_n, k_n^2)\}$.
  - This transformation takes $\mathcal{O}(n)$ time.
- Let $P_2$ be the problem of finding the convex hull of the $n$ points.
  - $T_2(n)$ is solution time for $P_2(n)$.
- Note that the $n$ points arranged in sorted order (sorted by $x$ coordinate) form a convex hull with the first point appended to the end.
- In this case

$$T_2(n) \le T_1(n) - \mathcal{O}(n) = \mathcal{O}(n \lg n) - \mathcal{O}(n). \tag{8.1.3}$$

- Thus, we have

**Lemma 8.1.9. Find Convex Hull**

Computing the convex hull of $n$ given points in the plane needs $\Omega(n \lg n)$ time.

# Multiplying Triangular Matrices

- Given an $n \times n$ matrix $A$ whose elements are $\{a_{i,j} | 1 \le i, j \le n\}$
- $A$ is said to be upper triangular if $a_{ij} = 0$ whenever $i > j$.
- $A$ is said to be lower triangular if $a_{ij} = 0$ for $i < j$.
- $A$ is said to be triangular if it is either upper triangular or lower triangular.
- We are interested in the question if multiplying two lower (or upper) triangular matrices is faster than multiplying two full matrices.
- Let $M(n)$ be the time complexity of multiplying two full matrices, and $M_t(n)$ be the time complexity of multiplying two lower triangular matrices.
    - Note that $M_t(n) \le M(n)$.
- And $M(n) = \Omega(n^2)$ since there are $2n^2$ elements in the input and $n^2$ elements in the output.

---

# Multiplying Triangular Matrices, II

- Let $P_1$ be the problem of multiplying two full matrices $A$ and $B$, each of size $n \times n$.
- Let $P_2$ be the problem of multiplying two lower triangular matrices.
- The problem of $P_1$ can be transformed into an instance of $P_2$ problem as

$$A' = \begin{bmatrix} \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & A & \mathbf{0} \end{bmatrix} \qquad B' = \begin{bmatrix} \mathbf{0} & \mathbf{0} & \mathbf{0} \\ B & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix}$$

where $\mathbf{0}$ denotes a zero matrix, that is, an $n \times n$ matrix with all elements 0.

- Note that both $A'$ nd $B'$ are lower triangular matrices.
- And

$$A'B' = \begin{bmatrix} \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \\ AB & \mathbf{0} & \mathbf{0} \end{bmatrix}$$

# Multiplying Triangular Matrices, III

- Thus, the product of full matrices can be obtained from product of lower triangular matrices.
- Transforming full matrices to triangular matrices takes $\mathcal{O}(n^2)$ time.
- Getting the product $AB$ from $A'B'$ also takes $\mathcal{O}(n^2)$.
- And we have

$$M_t(3n) \geq M(n) - \mathcal{O}(n^2) = \Omega(n^2) - \mathcal{O}(n^2) = \Omega(n^2) \tag{8.1.4}$$

- Or

$$M_t(n) \geq \Omega((\frac{n}{3})^2) = \Omega(n^2) = \Omega(M(n)). \tag{8.1.5}$$

- Thus we have

**Lemma 8.1.10. Multiplying triangular matrices**

$M_t(n) = \Omega(M(n))$.

- Since $M(n) \geq M_t(n)$ we conclude that $M_t(n) = \Theta(M(n))$.

---

# Inverting a Lower Triangular Matrix

- An $n \times n$ matrix $I$ is an identity matrix if

$$I_{j,k} = \begin{cases} 1, & \text{if } j = k, \\ 0, & \text{otherwise.} \end{cases} \tag{8.1.6}$$

- Given an $n \times n$ matrix $A$, if there exists a matrix $B$ such that $AB = I$, then $B$ is called the inverse of $A$ and $A$ is said to be invertible. Also, the inverse of $A$ is denoted as $A^{-1}$.
- Note that not every matrix is invertible.
- Given an $n \times n$ lower triangular matrix $A$, if all the diagonal elements $a_{i,j} \neq 0$, $1 \leq i, j \leq n$, then $A$ is invertible.
- In the following we are interested in the time complexity of inverting a lower triangular matrix, especially, compared to the full matrix multiplication.

# Inverting a Lower Triangular Matrix, II

- Let $P_1$ be the problem of multiplying two full matrices, and $P_2$ be the problem of inverting a lower triangular matrix.
- Let $I_t(n)$ be the time complexity of inverting a lower triangular matrix of dimension $n \times n$, and $M(n)$ is the complexity of multiplying two full matrices.
- Given two full $n \times n$ matrices $A$ and $B$, the following $3n \times 3n$ lower triangular matrix can be constructed

$$C = \begin{bmatrix} I & \mathbf{0} & \mathbf{0} \\ B & I & \mathbf{0} \\ \mathbf{0} & A & I \end{bmatrix} \tag{8.1.7}$$

where $I$ is the identity matrix of dimension $n \times n$ and $\mathbf{0}$ is the zero matrix of the same dimension.

---

# Inverting a Lower Triangular Matrix, III

- And it can be shown that the inverse matrix is

$$C^{-1} = \begin{bmatrix} I & \mathbf{0} & \mathbf{0} \\ -B & I & \mathbf{0} \\ AB & -A & I \end{bmatrix} \tag{8.1.8}$$

- Thus, matrix product can be obtained from inverting a matrix.
- Furthermore, we have $I_t(3n) \le M(n) - \mathcal{O}(n^2)$.
- Since $M(n) = \Omega(n^2)$ we have the following Lemma.

**Lemma 8.1.11.**

$I_t(n) = \Omega(M(n))$.

# Inverting a Lower Triangular Matrix, IV

- Given an $n \times n$ lower triangular matrix $A$, we can partition it into 4 submatrices of dimension $\frac{n}{2} \times \frac{n}{2}$ each as

$$A = \begin{bmatrix} A_{11} & \mathbf{0} \\ A_{21} & A_{22} \end{bmatrix} \tag{8.1.9}$$

where both $A_{11}$ and $A_{22}$ are lower triangular matrices, but $A_{21}$ can be full.

- It can be shown that

$$A^{-1} = \begin{bmatrix} A_{11}^{-1} & \mathbf{0} \\ -A_{22}^{-1} A_{21} A_{11}^{-1} & A_{22}^{-1} \end{bmatrix} \tag{8.1.10}$$

Thus, the inverse of $A$ can be constructed using divide-and-conquer approach.

- The inverse of submatrices $A_{11}$ and $A_{22}$ are first found, $2I_t(\frac{n}{2})$, and then two matrix multiplications are performed, $2M(\frac{n}{2})$, followed by negating all elements of the products, $\mathcal{O}(\frac{n}{4})$).

---

# Inverting a Lower Triangular Matrix, V

- And the currence equation is

$$\begin{aligned} I_t(n) &= 2I_t(\frac{n}{2}) + 2M(\frac{n}{2}) + \frac{n^2}{4} \\ &= 4I_t(\frac{n}{4}) + 4M(\frac{n}{4}) + 2\frac{n^2}{16} + 2M(\frac{n}{2}) + \frac{n^2}{4} \\ &= 2M(\frac{n}{2}) + 4M(\frac{n}{4}) + \cdots + \frac{n^2}{4} + \frac{n^2}{8} + \cdots \\ &= \mathcal{O}(M(n) + n^2) \end{aligned}$$

The last equality comes from $M(n) = \Omega(n^2)$. The following Lemma is obtained.

**Lemma 8.1.12.**

$I_t(n) = \mathcal{O}(M(n))$.

- Combining the last two lemmas, we conclude that $I_t(n) = \Theta(M(n))$. That is inverting a lower triangular matrix has the same time complexity as multiplying two full matrices.

# Summary

- Theoretical lower bounds
- Ordered searching
- Sorting
  - Merge sort
- Merging ordered arrays
- Finding the largest element
- The largest and 2nd largest elements
- The largest to the $k$-th largest elements
- Finding the maximum and the minimum
- Problem reduction
- Lower bound through problem reduction
- Finding convex hull.
- Lower triangular matrix multiplication
- Lower triangular matrix inversion

# Unit 9. $\mathcal{NP}$-complete Problems

Algorithms

EE3980

May 21, 2018

---

## Algorithm Time Complexities

- Time complexity of an algorithm depicts the execution time as a function of the input size.
  - It is desirable to have the time complexity as a polynomial of the input size with a small degree.
    - $\mathcal{O}(n)$, $\mathcal{O}(n \lg n)$, $\mathcal{O}(n^2)$
  - For some problems the algorithms have been found are not polynomials.
    - For example, the traveling salesperson problem and 0/1 knapsack problem.
    - $\mathcal{O}(n^2 2^n)$, $\mathcal{O}(2^{n/2})$.
    - These problems can have extreme long execution time for a moderate size problem.
- The goal of the unit is to identify those problems that have no known algorithms with polynomial time complexity.

# Nondeterministic Algorithms

- The algorithms described so far can always be executed with exact results – deterministic algorithms.
- A different class of algorithms, nondeterministic algorithms, allow the execution results to be not uniquely defined.
    - Three extra functions as following
        1. `Choice`$(S)$: chooses one of the elements of set $S$ arbitrarily.
        2. `Failure`$()$: signals an unsuccessful completion.
        3. `Success`$()$: signals an successful completion.
    - All three functions can be execute efficiently, i.e., $\mathcal{O}(1)$.
- Example
    - $x :=$ `Choice`$(1, n)$
    - $x$ is assigned with an integer in the range $[1, n]$.

---

# Nondeterministic Algorithms — Example

- Example: Nondeterministic search
  Given an array $A[1 : n]$ with $n$ integers, the following algorithm will find the index $j$ such that $A[j] = x$ or $j = 0$ if $x \notin A$.

### Algorithm 9.1.1. Nondeterministic Search

```
Algorithm NDSearch$(A, n, x)$
// A nondeterministic search algorithm.
{
    $j :=$ Choice $(1, n)$;
    if $(A[j] = x)$ then { write $(j)$; Success $()$; }
    write $(0)$; Failure $()$;
}
```

- It is assumed that the nondeterministic algorithm `NDSearch`$(A, n, x)$ can find the correct index $j$ such that $A[j] = x$ or 0 if no such $x$ in $A[1 : n]$.
- And it takes $\mathcal{O}(1)$ time to execute.
- As compared to the deterministic algorithm that has time complexity of $\mathcal{O}(n)$.
- It can be assumed there are $n$ processors to make choices then one of them will succeed.

# Nondeterministic Algorithms — Example, II

- Nondeterministic sort algorithm:
  Given an $n$-integer array $A$, the following algorithm sorts $A$ into a nondecreasing order.

## Algorithm 9.1.2. Nondeterministic Sort

```
1  Algorithm NDSort($A, n$)
2  // Sort $n$ positive integers.
3  {
4        for $i := 1$ to $n$ do $B[i] := 0$ ; // initialize $B$ array.
5        for $i := 1$ to $n$ do {
6            $j :=$ Choice $(1, n)$ ;
7            if $(B[j] \neq 0)$ Failure () ; // Repeated assignment.
8            $B[j] := A[i]$ ;
9        }
10       for $i := 1$ to $n - 1$ do // Verify order.
11           if $(B[i] > B[i + 1])$ then Failure () ;
12       write $(B[1 : n])$ ;
13       Success () ;
14 }
```

---

# Nondeterministic Algorithms — Example, III

- Note that an auxiliary array $B$ is used.
- If the `for` loop on lines 5-9 is successfully executed, array $B$ is a permutation of array $A$.
- Lines 10, 11 check if a nondecreasing order is achieved.
  If so, the sorting is done.
- The time complexity of `NDSort` algorithm is $\mathcal{O}(n)$.
  - As compared to $\mathcal{O}(n \lg n)$ in the deterministic case.
- There is no programming language or computer that can implement or execute the nondeterministic algorithms.
- The nondeterministic algorithms are tools for theoretical study in computer science.
- The primary objective of nondeterministic algorithm is whether an algorithm can result in a success
  – Verification Algorithms.

# Decision and Optimization Problems

**Definition. 9.1.3.**

1. Any problem for which the answer is either one or zero (true or false) is called a decision problem.
2. An algorithm for a decision problem is termed a decision algorithm.
3. Any problem that involves the identification of an optimal (either minimum or maximum) value of a given cost function is known as an optimization problem.
4. An optimization algorithm is used to solve an optimization problem.

- The nondeterministic algorithms are mostly for studying decision problems.
- Though there might be many failures when a nondeterministic algorithm executes, the concern is whether a success can be achieved.
- If a decision problem can be solved in polynomial time, then the corresponding optimization problem can solve in polynomial time, too.
- On the other hand, if a optimization problem cannot be solved in polynomial time, then the corresponding decision problem cannot be solved in polynomial time, either.

# Decision and Optimization Problems — Example

- Example: Maximum Clique Problem.
  - A maximal complete subgraph of a graph $G(V, E)$ is a clique.
  - The size of a clique is the number of vertices in the clique.
  - The maximum clique problem is an optimization problem that is to determine the largest clique in $G$.
  - The corresponding decision problem is to determine whether $G$ has a clique of size at least $k$ for some given $k$.
  - Let `DClique`$(G, k)$ be the deterministic algorithm for the decision problem.
  - If the number of vertices in $G$ is $n$, then the optimization problem can be solved by applying `DClique` repeatedly for different $k$, $k = n, n-1, \cdots$, until the output of `DClique` is 1.
  - If the time complexity of `DClique` is $f(n)$ then the optimization problem has the complexity less than or equal to $n \cdot f(n)$.
  - On the other hand, if the optimization problem can be solved in $g(n)$ time, then the decision problem can be solved in time $\leq g(n)$.
  - If the decision problem can be solved in polynomial time, then the optimization problem can also be solved in polynomial time.
  - If the optimization problem cannot be solved in polynomial time, then the corresponding decision problem cannot be solved in polynomial time, either.

# Nondeterministic Algorithm Time Complexity

### Definition 9.1.4.

The time required by a nondeterministic algorithm performing on any given input is the minimum number of steps needed to reach a successful completion if there exists a sequence of choices leading to such a completion. In case a successful completion is not possible, then the time required is $\mathcal{O}(1)$. A nondeterministic algorithm is of complexity $\mathcal{O}(f(n))$ if for all inputs of size $n$, $n \geq n_0$, that result in a successful completion, the time required is at most $c \cdot f(n)$ for some constants $c$ and $n_0$.

- Note the difference to the time complexity of a deterministic algorithm.

---

# Nondeterministic Algorithm Time Complexity Example

- Given $n$ objects with profits $p[1:n]$ and weights $w[1:n]$, and numbers $m$ and $r$, the following nondeterministic algorithm determined if there is an assignment $x[1:n]$, $x[i] = 0$ or 1, $1 \leq i \leq n$, such that

$$\sum_{i=1}^{n} x[i] \cdot p[i] \geq r \quad \text{and} \quad \sum_{i=1}^{n} x[i] \cdot w[i] \leq m.$$

### Algorithm 9.1.5. 0/1 Knapsack Decision Algorithm.

```
Algorithm NDKP($p, w, n, m, r, x$)
// Nondeterministic algorithm to decide if there is a solution assignment.
{
    $W := 0$ ; $P := 0$ ;
    for $i := 1$ to $n$ do {
        $x[i]$ := Choice $(0, 1)$ ; // assign $x[]$
        $W := W + x[i] \times w[i]$ ; $P := P + x[i] \times p[i]$ ;
    }
    if (($W > m$) or ($P < r$)) then Failure () ;
    else Success () ;
}
```

- The time complexity of a successful completion of this algorithm is $\mathcal{O}(n)$.

# Nondeterministic Algorithm Time Complexity Example, II

- Given a graph $G(V, E)$ with $n$ vertices, the following algorithm determines if there is a clique of size $k$ in $G$.

### Algorithm 9.1.6. Nondeterministic Graph Clique Decision

```
1  Algorithm NDCK($V, E, n, k$)
2  // If $G(V, E)$ contains a clique of size $k$.
3  {
4        $S := \emptyset$ ; // initialize $S$ to be empty set.
5        for $i := 1$ to $k$ do { // find $k$ distinct vertices
6            $t :=$ Choice $(1, n)$ ;
7            if $(t \in S)$ then Failure () ;
8            $S := S \cup \{t\}$ ; // Add $t$ to set $S$.
9        }
10       for ( all pairs $(i, j)$ such that $i \in S, j \in S$ and $i \neq j$) do
11           if $(i, j) \notin E$ then Failure () ;
12       Success () ;
13 }
```

- The time complexity is dominated by the for loop on lines 10,11, $\mathcal{O}(k^2) \leq \mathcal{O}(n^2)$.
- There is no known polynomial time algorithm for the deterministic graph clique decision problem.

---

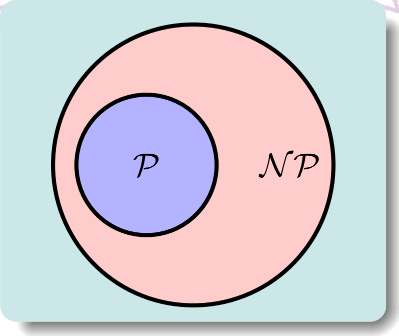
# $\mathcal{P}$ and $\mathcal{NP}$

- An algorithm $A$ is of polynomial complexity if there exists a polynomial $p$ such that the computing time of $A$ is $\mathcal{O}(p(n))$ for every input of size $n$.

### Definition 9.1.7. $\mathcal{P}$ and $\mathcal{NP}$

$\mathcal{P}$ is the set of all decision problems solvable by deterministic algorithms in polynomial time. $\mathcal{NP}$ is the set of all decision problems solvable by nondeterministic algorithms in polynomial time.

- Since deterministic algorithms are special cases of nondeterministic algorithms, we have $\mathcal{P} \subseteq \mathcal{NP}$.
- It is not known which of the following is true: $\mathcal{P} = \mathcal{NP}$ or $\mathcal{P} \neq \mathcal{NP}$.
- The common belief of their relationship is shown below



# Polynomial Time Transformation (Reducibility)

- Given two problems $Q_1$ and $Q_2$, if there is a polynomial time transformation such that $Q_1$ can be transformed into $Q_2$ we say that $Q_1$ transforms to $Q_2$ and denotes $Q_1 \propto Q_2$.
  - It is also commonly referred as $Q_1$ reduces to $Q_2$.
- Given the polynomial transformation $Q_1 \propto Q_2$, if $Q_2$ can be solved in polynomial time, then $Q_1$ can be solved in polynomial time as well.

**Lemma 9.1.8.**

If $Q_1 \propto Q_2$, then if $Q_2 \in \mathcal{P}$ then $Q_1 \in \mathcal{P}$ (and, equivalently, $Q_1 \notin \mathcal{P}$ then $Q_2 \notin \mathcal{P}$).

**Lemma 9.1.9.**

If $Q_1 \propto Q_2$ and $Q_2 \propto Q_3$, then $Q_1 \propto Q_3$.

---

# $\mathcal{NP}$-complete

- A problem $Q$ is said to be $\mathcal{NP}$-complete if $Q \in \mathcal{NP}$ and for all other $Q' \in \mathcal{NP}$, $Q' \propto Q$.
  - Thus, the $\mathcal{NP}$-complete problems are the hardest problems in $\mathcal{NP}$.
  - If any one can be solved in polynomial time, then all problems in $\mathcal{NP}$ can be solved in polynomial time.

**Lemma 9.1.10.**

If $Q_1$ and $Q_2$ belong to $\mathcal{NP}$, if $Q_1$ is $\mathcal{NP}$-complete and $Q_1 \propto Q_2$ then $Q_2$ is $\mathcal{NP}$-complete.

**Definition 9.1.11. Polynomial equivalency.**

Two problems $Q_1$ and $Q_2$ are said to be polynomial equivalent if and only if $Q_1 \propto Q_2$ and $Q_2 \propto Q_1$.

- To show a problem $Q_2$ is $\mathcal{NP}$-complete, it is adequate to show $Q_1 \propto Q_2$, where $Q_1$ is a problem already known to be $\mathcal{NP}$-complete.

# Satisfiability Problem

- Let $x_1, x_2, \cdots, x_n$ be boolean variables such that $x_i$ can be either *true* or *false*.
- Let $\overline{x_i}$ denote the negation of $x_i$.
- A literal is either a boolean variable or its negation.
- A formula in the propositional calculus is an expression that can be constructed using literals and the operators and and or.
- Examples of formulas

  $$(x_1 \wedge x_2) \vee (x_3 \wedge \overline{x_4}), \qquad (x_3 \vee \overline{x_4}) \wedge (x_1 \vee \overline{x_2})$$

  The symbol $\vee$ denotes or and $\wedge$ denotes and.
- A formula is in conjunctive normal form (CNF) if and only if it is represented as $\bigwedge_{i=1}^{k} c_i$, where $c_i$ are clauses each represented as $\bigvee l_{ij}$. The $l_{ij}$ are literals.
  - Example of CNF: $(x_3 \vee \overline{x_4}) \wedge (x_1 \vee \overline{x_2})$.
- A formula is in disjunctive normal form if and only if it is represented as $\bigvee_{i=1}^{k} c_i$ and each clause $c_i$ is represented as $\bigwedge l_{i,j}$.
  - Example of DNF: $(x_1 \wedge x_2) \vee (x_3 \wedge \overline{x_4})$.

---

# Satisfiability Problem, II

- The satisfiability problem is to determine whether a formula is true for some assignment of truth values to the variables.
- The CNF-satisfiability is the satisfiability problem for CNF formula.
- Given an expression $E$ and $n$ boolean variables represented by the array $x[1:n] = (x_1, x_2, \cdots, x_n)$, the following nondeterministic algorithm find a set of truth value assignments that satisfies $E$, that is, $E(x_1, x_2, \cdots, x_n) =$ `true`.

**Algorithm 9.1.12. Nondeterministic Satisfiability.**

```
1 Algorithm NSat($E, n, x$)
2 // Nondeterministic algorithm for satisfiability problem.
3 {
4       for $i := 1$ to $n$ do // Choose a truth value assignment.
5           $x[i]$ := Choice ( false , true );
6       if $E(x)$ then Success ();
7       else Failure ();
8 }
```

- The time complexity is $\mathcal{O}(n)$ (`for` loop on lines 4-5) plus the time to evaluation expression $E$.

# Cook's Theorem

- It is known from Algorithm (9.1.12) that the satisfiability decision problem is in $\mathcal{NP}$, and we have the following theorem by Cook.

**Theorem 9.1.13. Cook's Theorem.**

Satisfiability is in $\mathcal{P}$ if and only if $\mathcal{P} = \mathcal{NP}$.

- Proof please see textbook [Horowitz], pp. 527-535, or [Cormen] pp. 1074-1077.
- S.A. Cook, "The complexity of theorem proving procedures." In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, pp. 151-158, 1971.

- In other words, satisfiability problem is $\mathcal{NP}$-complete.
- This is the first known $\mathcal{NP}$-complete problem.
  - Then others can be reduced from it.

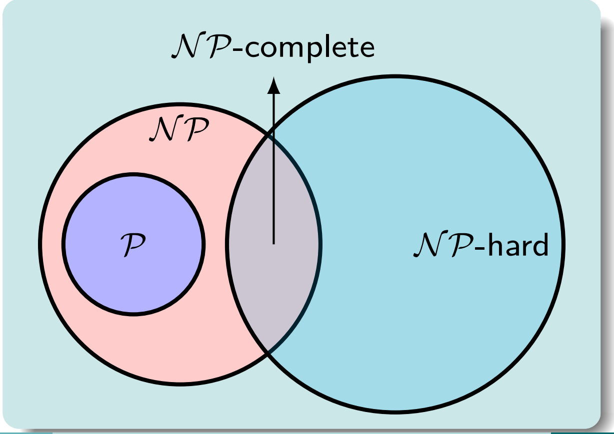
---

# $\mathcal{NP}$-Hard and $\mathcal{NP}$-Complete

**Definition. 9.1.14. $\mathcal{NP}$-hard and $\mathcal{NP}$-complete.**

A problem $Q$ is $\mathcal{NP}$-hard if and only if satisfiability reduces to $Q$ (satisfiability $\propto Q$). A problem $Q$ is $\mathcal{NP}$-complete if and only if $Q$ is $\mathcal{NP}$-hard and $Q \in \mathcal{NP}$.

- There are $\mathcal{NP}$-hard problems that are not $\mathcal{NP}$-complete.
- Only a decision problem can be $\mathcal{NP}$-complete.
- If $Q_1$ is a decision problem and $Q_2$ is the corresponding optimization problem, then it is quite possible that $Q_1 \propto Q_2$.
- An $\mathcal{NP}$-complete decision problem may have its corresponding optimization problem be $\mathcal{NP}$-hard.
- There are also decision problems that are $\mathcal{NP}$-hard.



# 3-Satisfiability Problem (3-SAT)

- 3-satisfiability problem is a special case of the CNF-satisfiability problem, where each clause has exactly three literals.
- A clause, $C_k$, of $k$ literals can be converted into a CNF of 3 literals, $C'_k$, as the following. ($y_i$'s are auxiliary variables.)

$$\begin{aligned}
k = 1, \quad & C_1 = x_1, \\
& C'_1 = (x_1 \vee y_1 \vee y_2) \wedge (x_1 \vee \overline{y}_1 \vee y_2) \wedge (x_1 \vee y_1 \vee \overline{y}_2) \wedge (x_1 \vee \overline{y}_1 \vee \overline{y}_2), \\
k = 2, \quad & C_2 = x_1 \vee x_2, \\
& C'_2 = (x_1 \vee x_2 \vee y_1) \wedge (x_1 \vee x_2 \vee \overline{y}_1), \\
k = 3, \quad & C_3 = x_1 \vee x_2 \vee x_3, \\
& C'_3 = x_1 \vee x_2 \vee x_3, \\
k > 3, \quad & C_k = x_1 \vee x_2 \vee \cdots \vee x_k, \\
& C'_k = (x_1 \vee x_2 \vee y_1) \wedge (\overline{y}_1 \vee x_3 \vee y_2) \wedge \cdots \wedge (\overline{y}_{k-3} \vee x_{k-1} \vee x_k).
\end{aligned}$$

**Theorem 9.1.15. 3-SAT**

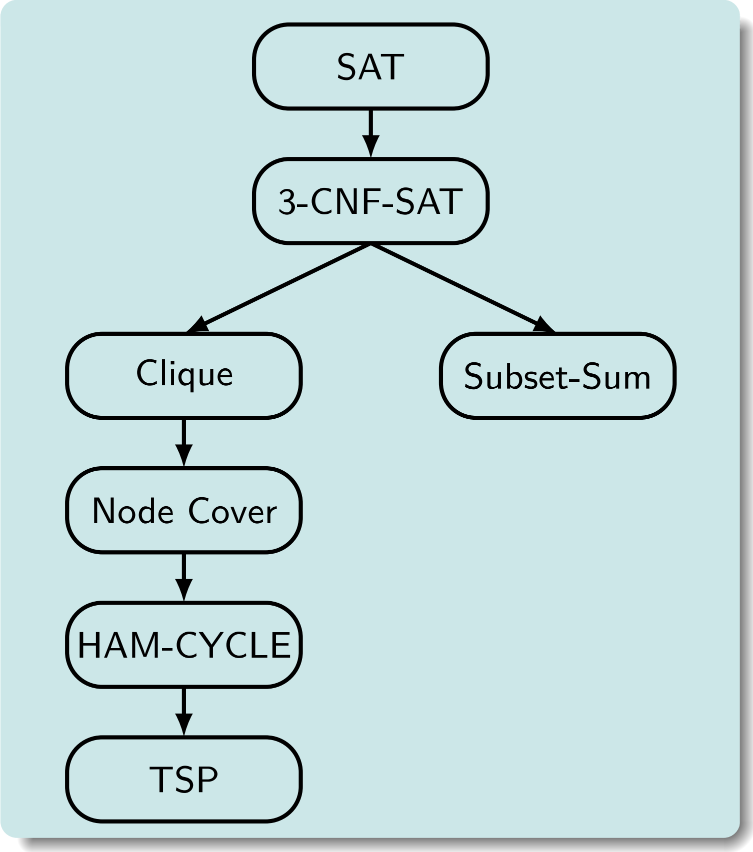
CNF-satisfiability problem $\propto$ 3-satisfiability problem.

- Thus, 3-satisfiability problem is $\mathcal{NP}$-complete.

---

# Finding Other $\mathcal{NP}$-Complete Problems

- From the Satisfiability problem, more $\mathcal{NP}$-complete problems were identified.



# Clique Decision Problem (CDP)

- A graph clique decision problem (CDP) is given a graph $G(V, E)$ to decide if there are cliques of size $k$ in $G$.
- CDP is $\mathcal{NP}$-complete.

### Theorem 9.1.16. CDF

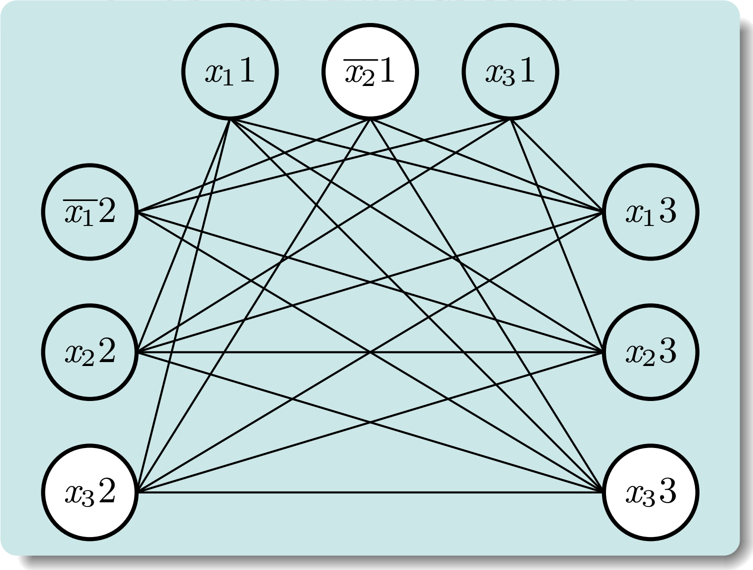
CNF-satisfiability $\propto$ clique decision problem.

- Let $F = \bigwedge_{i=1}^{k} C_i$ be a propositional formula in CNF.
  - Let $x_i, 1 \le i \le n$ be a variable in $F$.
- Define $G = (V, E)$ as follows:
  - $V = \{\langle \sigma, i \rangle | \sigma$ is a literal in clause $C_i\}$.
  - $E = \{(\langle \sigma, i \rangle, \langle \delta, j \rangle) | i \neq j$ and $\sigma \neq \overline{\delta}\}$.
- The $F$ is satisfiable if and only if $G$ has a clique of size $k$.
- If the length of $F$ is $m$, the sum variables of each clause, then $G$ is obtainable from $F$ in $\mathcal{O}(m^2)$ time.

---

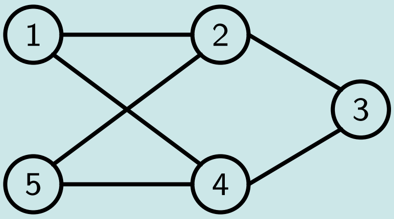
# Clique Decision Problem (CDP), II

- $Q_1$: 3-Satisfiability.
  $\mathcal{I} = (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$.
- $Q_2$: Clique Decision problem.
  $G(V_{\mathcal{I}}, E_{\mathcal{I}})$ has a clique of size 3?

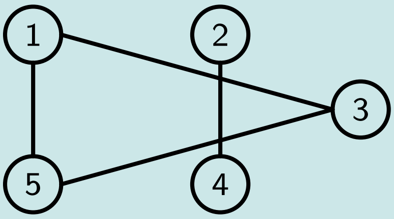


# Node Cover Decision Problem (NCDP)

- A set $S \subseteq V$ is a node cover for a graph $G(V, E)$ if and only if all edges in $E$ are incident to at least one vertex in $S$. The size $|S|$ of the cover is the number of vertices in $S$.
- The node cover decision problem is given a graph $G(V, E)$ and an integer $k$ to determine if there is a node cover of size at most $k$.
- Example: Given a graph shown below.
  - $S_1 = \{2, 4\}$ is a node cover of size 2.
  - $S_2 = \{1, 3, 5\}$ is a node cover of size 3.



Graph $G$.



Graph $G'$.

---

# Node Cover Decision Problem (NCDP), II

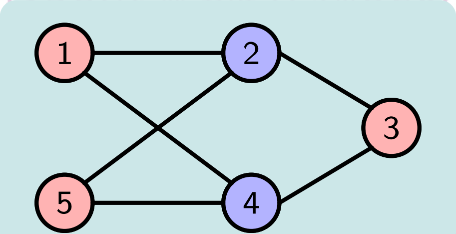
**Theorem 9.1.17. NCDP**

The clique decision problem $\propto$ the node cover decision problem.

- Given a $G(V, E)$ and an integer $k$, and instance of clique decision problem is defined. Assume that $|V| = n$.
- Construct a graph $G'(V, E')$, where $E' = \{(u, v) | u \in V, v \in V$ and $(u, v) \notin E\}$.
- This graph $G'$ is known as the complement of $G$.
- If $K$ is a clique in $G$, since there are no edges in $E'$ connecting vertices in $K$, the remaining $n - |K|$ vertices in $G'$ must cover all edges in $E'$.
- Thus if $G$ has a clique of size at least $k$ if and only if $G'$ has a node cover of size at most $n - k$.
- Note that $G'$ can be constructed from $G$ in $\mathcal{O}(n^2)$ time, thus theorem is proved.
- Note also that since CNF-satisfiability $\propto$ CDP, and CDP $\propto$ NCDP, therefore NCDP is $\mathcal{NP}$-hard.
- NCDP is also $\mathcal{NP}$, so NCDP is $\mathcal{NP}$-complete.

# Chromatic Number Decision Problem (CNDP)

- A coloring of a graph $G(V, E)$ is a function $f: V \to \{1, 2, \ldots, k\}$ defined for all $i \in V$. If $(u, v) \in E$, then $f(u) \neq f(v)$.
- The chromatic number decision problem is to determine whether $G$ has a coloring for a given $k$.
- Example: a two-coloring graph.



## Theorem 9.1.18. CNDP

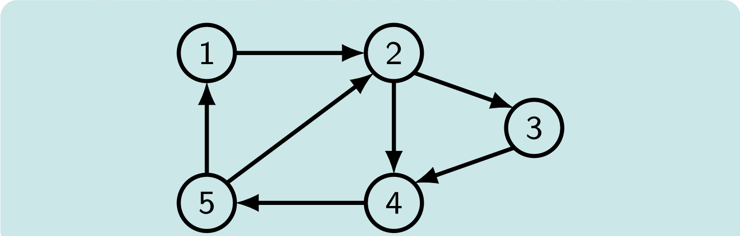
3-satisfiability problem $\propto$ chromatic number decision problem.

- Proof see textbook [Horowitz], pp. 540-541.

---

# Directed Hamiltonian Cycle (DHC) Problem

- A directed Hamiltonian cycle in a directed graph $G(V, E)$ is a directed cycle of length $n = |V|$.
- The directed Hamiltonian cycle goes through every vertex exactly once and returns to the starting vertex.
- The DHC problem is to determine whether $G$ has a directed Hamiltonian cycle.
- Example: $(1, 2, 3, 4, 5, 1)$ is a Hamiltonian cycle.



## Theorem 9.1.19. DHC

CNF-satisfiability $\propto$ directed Hamiltonian cycle.

- Directed Hamiltonian cycle problem is $\mathcal{NP}$-complete.
- Proof please see textbook [Horowitz], pp. 542-545, or [Cormen], pp. 1091-1096 (for undirected graph).

# Traveling Salesperson Decision Problem (TSP)

- The traveling salesperson decision problem (TSP) is to determine whether a complete directed graph $G(V, E)$ with edge cost $c(u, v)$, $u, v \in V$, has a tour of cost at most $M$.

**Theorem 9.1.20. TSP**

Directed Hamiltonian cycle (DHC) $\propto$ the traveling salesperson decision problem (TSP).

- Given a directed graph $G(V, E)$ for the DHC problem, construct a complete directed graph $G'(V, E')$, $E' = \{\langle i, j\rangle | i \neq j\}$ and $c(i, j) = 1$ if $\langle i, j\rangle \in E$; $c(i, j) = 2$ if $i \neq j$ and $\langle i, j\rangle \notin E$. In this case, $G'$ has a tour of cost at most $n$ if and only if $G$ has a directed Hamiltonian cycle.
- TSP is an $\mathcal{NP}$-completeproblem.
- Both Hamiltonian Cycle and Travelling Salesperson Problem can be defined for undirected graph as well.
- Both undirected Hamiltonian Cycle and Travelling salesperson Problem are also $\mathcal{NP}$-complete.
- Proof please see textbook [Cormen], pp. 1091-1097.

---

# Partition Problem

- Given a set $A = \{a_1, a_2, \cdots, a_n\}$ of $n$ integers. The partition problem is to determine whether there is a partition $P$ such that

$$\sum_{i \in P} a_i = \sum_{i \notin P} a_i.$$

**Theorem 9.1.21. Partition Problem.**

3-satisfiability problem $\propto$ partition problem.

- Proof see Garey and Johnson, *Computers and Intractability*, Freeman, 1979, p. 60.
- Thus, partition problem is a $\mathcal{NP}$-complete problem.

# Sum of Subsets Problem

- Given a set $A = \{a_1, a_2, \cdots, a_n\}$ of $n$ integers and an integer $M$. The sum of subsets problem is to determine whether there is a subset $S \subseteq A$ such that
$$\sum_{a_i \in S} a_i = M.$$
- Given the $n$-integer set $A$, an $n+2$ set $B$ can be constructed as
$$\begin{aligned} b_i &= a_i, \qquad 1 \le i \le n, \\ b_{n+1} &= M + 1, \\ b_{n+2} &= (\sum_{i=1}^{n} a_i) - M + 1, \end{aligned}$$
Then $$b_{n+2} + \sum_{b_i \in S} b_i = b_{n+1} + \sum_{b_i \notin S} b_i.$$
The partition problem in $B$ is equivalent to the sum of subsets problem in $A$.

**Theorem 9.1.22. Sum of subsets.**

Sum of subsets problem $\propto$ partition problem.

- The sum of subsets problem is $\mathcal{NP}$-complete.

# Scheduling Identical Processors Problems

- Let $P_i$, $1 \le i \le m$, be $m$ identical processors.
- Let $J_i$, $1 \le i \le n$, be $n$ jobs. Each job $J_i$ requires $t_i$ processing time.
- A schedule $S$ is an assignment of jobs to processors. For each job $J_i$, $S$ specifies the time interval and the processor that processes $J_i$.
  - A job cannot be processed by more than one processor at any given time.
- Let $f_i$ be the time at which job $J_i$ complete processing. The mean finish time (MFT) of schedule $S$ is
$$\text{MFT}(S) = \frac{1}{n} \sum_{i=1}^{n} f_i. \tag{9.1.1}$$
- Let $w_i$ be a weight associated with each job $J_i$. The weighted mean finish time (WMFT) of schedule $S$ is
$$\text{WMFT}(S) = \frac{1}{n} \sum_{i=1}^{n} w_i \cdot f_i. \tag{9.1.2}$$
- Let $T_i$ be the time at which $P_i$ finishes processing all jobs assigned to it. The finish time (FT) of schedule $S$ is
$$\text{FT}(S) = \max_{i=1}^{m} T_i. \tag{9.1.3}$$
- Schedule $S$ is a nonpreemptive schedule if and only if each job $J_i$ is processed continuously from start to finish on the same processor. Otherwise, it is preemptive.

## Scheduling Problems – Complexities

### Theorem 9.1.23. MFT

Partition problem $\propto$ minimum finish time nonpreemptive schedule problem.

- For $m = 2$ case, given the set $\{a_1, a_2, \cdots, a_n\}$ as an instance of the partition problem. Define $n$ jobs with processing time $t_i = a_i$, $1 \le i \le n$. There is a nonpreemptive schedule for this set of jobs on two processors with finish time at most $\sum t_i/2$ if and only if there is a partition of the set $\{a_i | 1 \le i \le n\}$. It can also be proved for $m > 2$ cases.

### Theorem 9.1.24. WMFT

Partition problem $\propto$ minimum WMFT nonpreemptive schedule problem.

- For $m = 2$ case, given the set $\{a_1, a_2, \cdots, a_n\}$ define a two-processor scheduling problem with $w_i = t_i = a_i$. Then there is a nonpreemptive schedule $S$ with weighted mean finish time at most $1/2 \sum a_i^2 + 1/4(\sum a_i)^2$ if and only if the set $\{a_i | 1 \le i \le n\}$ has a partition.
  The rest of the proof please see textbook [Horowitz], pp. 554-555.

---

## Scheduling Problems – Complexities, II

### Theorem 9.1.25. Flow Shop Scheduling

Partition problem $\propto$ the minimum finish time preemptive flow shop schedule with $m > 2$. ($m$ is the number of processors.)

- Proof please see textbook [Horowitz], pp. 555-556.

### Theorem 9.1.26. 2-processor Flow Shop Scheduling

2-processor flow shop schedule $\in \mathcal{P}$.

- Dynamic programming approach can solve this problem in polynomial time. Please see textbook [Horowitz], pp. 321-325.

### Theorem 9.1.27. Job Shop Scheduling

Partition problem $\propto$ the minimum finish time preemptive job shop schedule with $m > 1$. ($m$ is the number of processors.)

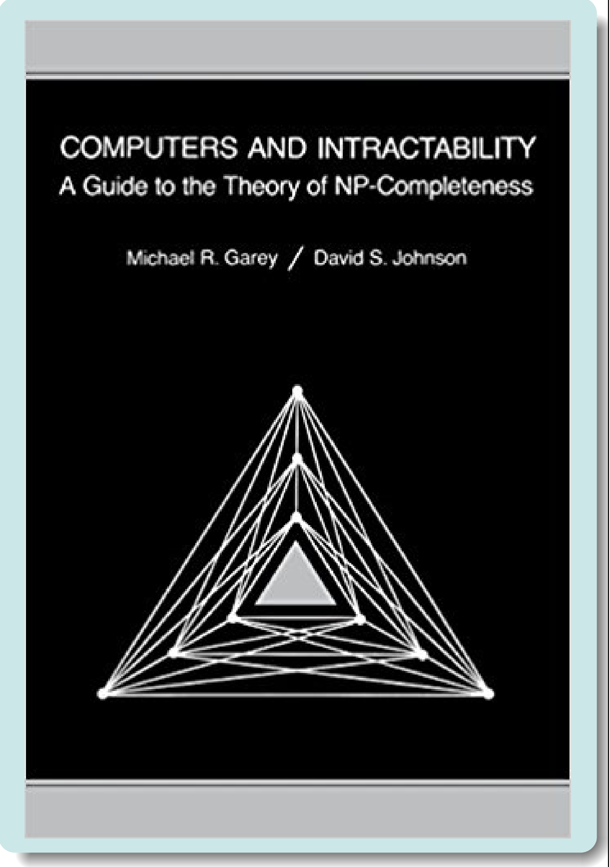
- Proof please see textbook [Horowitz], pp. 557-558.

# Other $\mathcal{NP}$-complete Problems

- Since 1971, many $\mathcal{NP}$-complete problems have been found.
- A good source book is

  M.R. Garey and D.S. Johnson,
  *Computers and Intractability – A Guide to the Theory of NP-Completeness*,
  W.H. Freeman, 1979.

- More than 320 $\mathcal{NP}$-complete problems listed in its reference, pp. 190-288.



COMPUTERS AND INTRACTABILITY
A Guide to the Theory of NP-Completeness
Michael R. Garey / David S. Johnson

---

# 2-SAT Problem

- It has been shown that Satisfiability (SAT) and 3-SAT problems are $\mathcal{NP}$-complete.
- In the following we study 2-SAT problem.
- 2-SAT problem is also a special case of SAT problem. In this problem, each clause has exactly two literals.
- Example

$$F(x_1, x_2, x_3, x_4) = (x_1 \vee x_2) \wedge (x_2 \vee \overline{x_3}) \wedge (\overline{x_2} \vee \overline{x_4}) \wedge (x_2 \vee x_4) \wedge (x_4 \vee x_1).$$

  - Given formula shown above, is it satisfiable? That is, can one set $x_i = \mathit{true}$ or $x_i = \mathit{false}$ for each $x_i$ such that the formula is evaluated to be $\mathit{true}$.

# 2-SAT Problem, II

- Example

$$F(x_1, x_2, x_3, x_4) = (x_1 \vee x_2) \wedge (x_2 \vee \overline{x_3}) \wedge (\overline{x_2} \vee \overline{x_4}) \wedge (x_2 \vee x_4) \wedge (x_4 \vee x_1).$$

- In evaluating $F(x_1, x_2, x_3, x_4)$, one can set $x_2 = 1$ ($true$), then $\overline{x_2} = 0$ ($false$) and the formula becomes

$$F(x_1, x_2 = 1, x_3, x_4) = (\overline{x_4}) \wedge (x_4 \vee x_1).$$

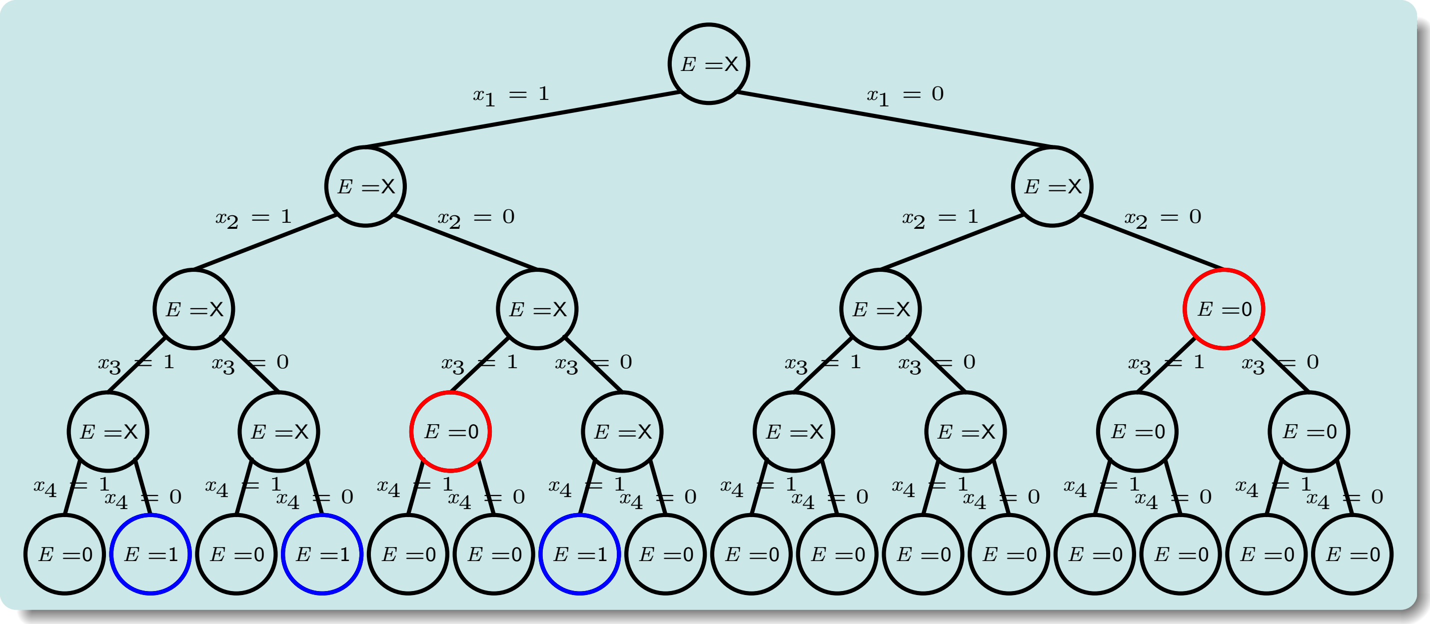
- Three clauses, $(x_1 \vee x_2)$, $(x_2 \vee \overline{x_3})$, and $(x_2 \vee x_4)$, become $true$, and thus can be eliminated from the formula.
- The clause $(\overline{x_2} \vee \overline{x_4})$ reduces to $(\overline{x_4})$ since $\overline{x_2} = 0$.
- In order $F(x_1, x_2, x_3, x_4) = 1$, one must have $x_4 = 0$ and $x_1 = 1$.
- The value of $x_3$ does not impact $F$ and can be either 0 or 1 (don't care).
- This shows that $F(x_1, x_2, x_3, x_4)$ is satisfiable with $(x_1, x_2, x_3, x_4) = (1, 1, \times, 0)$.

---

# 2-SAT Problem, III

$$F(x_1, x_2, x_3, x_4) = (x_1 \vee x_2) \wedge (x_2 \vee \overline{x_3}) \wedge (\overline{x_2} \vee \overline{x_4}) \wedge (x_2 \vee x_4) \wedge (x_4 \vee x_1).$$

- The complete state space for the formula
  - Backtracking or branch-and-bound can be used to find the answer.
  - $\mathcal{O}(2^n)$, $n$ is the number of boolean variables.



# 2-SAT Problem – Implicative Form

- In propositional calculus, the following two simple formulas are equivalent.

$$\begin{aligned} F_1 &= x_1 \vee x_2 \\ F_2 &= \overline{x_1} \rightarrow x_2 \end{aligned} \tag{9.1.4}$$

- Since $x_1 \vee x_2 = x_2 \vee x_1$, the following three are equivalent

$$\begin{aligned} F_1 &= x_1 \vee x_2 \\ F_2 &= \overline{x_1} \rightarrow x_2 \\ F_3 &= \overline{x_2} \rightarrow x_1 \end{aligned} \tag{9.1.5}$$

- It is easy to see the followings.

$$F_4 = x_1 \rightarrow x_1 \equiv \overline{x_1} \vee x_1 = \mathit{true}, \tag{9.1.6}$$

$$F_5 = \overline{x_1} \rightarrow \overline{x_1} \equiv x_1 \vee \overline{x_1} = \mathit{true}. \tag{9.1.7}$$

Yet,

$$F_6 = x_1 \rightarrow \overline{x_1} \equiv \overline{x_1} \vee \overline{x_1} \tag{9.1.8}$$

$$F_7 = \overline{x_1} \rightarrow x_1 \equiv x_1 \vee x_1 \tag{9.1.9}$$

$F_6$ can be *true* if $x_1 = \mathit{false}$, and $F_7$ can be *true* if $x_1 = \mathit{true}$.

---

# 2-SAT Problem – Implicative Form, II

- But,

$$\begin{aligned} F_8 &= (x_1 \rightarrow \overline{x_1}) \wedge (\overline{x_1} \rightarrow x_1) \\ &\equiv (\overline{x_1} \vee \overline{x_1}) \wedge (x_1 \vee x_1) \\ &= \overline{x_1} \wedge x_1 = \mathit{false}. \end{aligned} \tag{9.1.10}$$

- Using this equivalent relationship, the formulas in conjunctive normal form can be easily translated to the implicative form.

$$\begin{aligned} F(x_1, x_2, x_3, x_4) =& (x_1 \vee x_2) \wedge (x_2 \vee \overline{x_3}) \wedge (\overline{x_2} \vee \overline{x_4}) \wedge (x_2 \vee x_4) \wedge (x_4 \vee x_1) \\ \equiv& (\overline{x_1} \rightarrow x_2) \wedge (\overline{x_2} \rightarrow \overline{x_3}) \wedge (x_2 \rightarrow \overline{x_4}) \wedge (\overline{x_2} \rightarrow x_4) \wedge (\overline{x_4} \rightarrow x_1) \wedge \\ & (\overline{x_2} \rightarrow x_1) \wedge (x_3 \rightarrow x_2) \wedge (x_4 \rightarrow \overline{x_2}) \wedge (\overline{x_4} \rightarrow x_2) \wedge (\overline{x_1} \rightarrow x_4). \end{aligned}$$

- And, a directed graph $G(V, E)$ can be constructed from the conjunctive normal form $(F(x_1, x_2, \ldots, x_n) = \bigwedge_{j=1}^{m}(x_i \vee x_j))$.

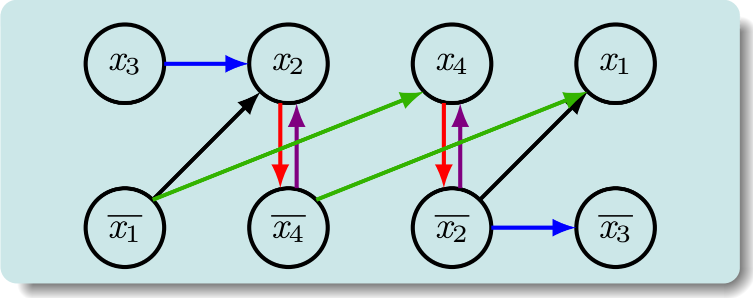
$$\begin{aligned} V &= \{y_i \,|\, y_i = x_i \text{ or } y_i = \overline{x_i}, i = 1, \ldots, n\}, \\ E &= \{(\overline{y_i}, y_j)(\overline{y_j}, y_i) \,|\, (y_i \vee y_j) \text{ is one clause in } F\}. \end{aligned}$$

Note that $|V| = 2n$ and $|E| = 2m$, where $n$ is the number of variables and $m$ is the number of clauses in $F$.

# Implicative Graph

- Given the formula, the following graph is constructed.
  - Two strongly connected components, $\{x_2, \overline{x_4}\}$ and $\{\overline{x_2}, x_4\}$ can be observed.

$$F(x_1, x_2, x_3, x_4) = (x_1 \vee x_2) \wedge (x_2 \vee \overline{x_3}) \wedge (\overline{x_2} \vee \overline{x_4}) \wedge (x_2 \vee x_4) \wedge (x_4 \vee x_1)$$



**Lemma 9.1.28. 2SAT**

Given a formula $F(x_1, x_2, \ldots, x_n)$ and its implicative graph $G(V, E)$ then $F$ is NOT satisfiable if and only if there is a strongly connected component in $G$ that contains a boolean variable $x_i$ and its complement $\overline{x_i}$.

- By the preceding lemma, the formula given above is satisfiable since those two strongly connected components contain no boolean variable together with its complement.

---

# Solving 2-SAT Problems

- From the lemma, one can solve the 2-SAT problem by
  1. Construct the implicative graph, $G(V, E)$, of the formula $F(x_1, x_2, \ldots, x_n)$.
  2. Find all the strongly connected components, $S_i$, of $G(V, E)$.
  3. Check all the strongly connected components to see if any $S_i$ contains both $x_j$ and $\overline{x_j}$.
  4. If no such $S_i$ and $x_j$ exist, then $F(x_1, x_2, \ldots, x_n)$ is satisfiable; Otherwise, $F(x_1, x_2, \ldots, x_n)$ is not satisfiable.
- Note that
  1. $G(V, E)$ can be constructed in $\mathcal{O}(n + m)$ time, since $|V| = 2n$ and $|E| = 2m$. ($n$ is the number of boolean variables and $m$ is the number of clauses in $F$).
  2. The strongly connected graph can be find in $\mathcal{O}(|V| + |E|)$ time.
  3. Check for if both $x_j$ and $\overline{x_j}$ are in $S_i$ can be done in $\mathcal{O}(|S_i|)$ time.
  4. Thus, determine if $F(x_1, x_2, \ldots, x_n)$ is satisfiable can be done in $\mathcal{O}(n + m)$ time.

**Lemma 9.1.29.**

2-SAT $\in \mathcal{P}$.

# Summary

- Nondeterministic algorithms
    - Examples
    - Complexity
- Decision and optimization problems
- Polynomial time transformation
- $\mathcal{P}$, $\mathcal{NP}$ and $\mathcal{NP}$-complete
- Satisfiability problem
- $\mathcal{NP}$-complete problems
    - 3-SAT
    - Graph clique problem
    - Node cover problem
    - Chromatic number problem
    - Hamiltonian cycle problem
    - Traveling salesperson problem
    - Partition problem
    - Sum of subsets problem
    - Scheduling identical processors problem
- 2-SAT problem

# Unit 10. Approximation Algorithms

Algorithms

EE3980

May 30, 2018

## 0/1 Knapsack Problem

- Given $n$ objects, each with profit $p_i$ and weight $w_i$, $1 \le i \le n$, to be placed into a sack that can hold maximum of $m$ weight. However, there is an additional constraint that each object must be placed as a whole into the sack, or not at all. That is, find $x_i$, $1 \le i \le n$, such that

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^{n} p_i x_i, \\ \text{subject to} \quad & \sum_{i=1}^{n} w_i x_i \le m, \\ \text{and} \quad & x_i = 0 \text{ or } 1, \qquad 1 \le i \le n. \end{aligned} \tag{10.1.1}$$

- We need $\sum_{i=1}^{n} w_i > m$ for nontrivial solutions.
- It is assumed that the $n$ objects are ordered by $p_i/w_i$ in a nonincreasing order.
- It is also assumed that the optimal profit is $p^*$.
- The following greedy algorithm can find a feasible but not necessarily the optimal solution.

# 0/1 Knapsack Problem – Greedy Algorithm

## Algorithm 10.1.1. Greedy Knapsack

```
Algorithm GKnap0($n, p, w, x, m$)
// Find solution $x[1:n]$ given $n$ objects with profits $p[1:n]$, weights $w[1:n]$
// and capacity $m$.
// The objects are assumed to be sorted by $p[i]/w[i]$ in nonincreasing order.
{
    for $i := 1$ to $n$ do $x[i] := 0$;
    $i := 1$; $fp_1 := 0$;
    while ($m \geq w[i]$) do {
        $x[i] := 1$; $fp_1 := fp_1 + p[i]$; $m := m - w[i]$; $i := i + 1$;
    }
}
```

- At the end of the algorithm `GKnap0` object $i$ is placed into the sack if $x[i] = 1$, and $fp_1$ is the final profit.
- It is easy to see that $fp_1 \leq p^*$, and $fp_1 < p^*$ most of the time.

---

# 0/1 Knapsack Problem – An example

- An example of the knapsack problem:
  Given $n$ objects, $p_i = 1$ and $w_i = 1$ for $i = 1, \ldots, n-1$, and $p_n = k \cdot n - 1$, $w_n = m = k \cdot n$, $k \gg 1$.
- The optimal profit for this problem is $p^* = k \cdot n - 1$ with $x_n = 1$ and $x_i = 0$, $i = 1, \ldots, n-1$.
- Note that $p_i/w_i = 1$ for $i = 1, \ldots, n-1$ and $p_n/w_n = (k \cdot n - 1)/(k \cdot n) = 1 - 1/(k \cdot n) < 1$. Thus, the objects are already in a nonincreasing order.
- The Greedy Knapsack algorithm finds a solution $x_i = 1$, $i = 1, \ldots, n-1$, and $x_n = 0$ with a profit $fp_1 = n - 1$.
- The ratio $p^*/fp_1 = (k \cdot n - 1)/(n-1) \gg 1$.
- The greedy Knapsack algorithm can be modified as the following to fix this problem.

# 0/1 Knapsack Problem – Revised Greedy Algorithm

## Algorithm 10.1.2. Revised Greedy Knapsack

```
1  Algorithm GKnap($n, p, w, x, m$)
2  // Find solution $x[1:n]$ given $n$ objects with profits $p[1:n]$, weights $w[1:n]$
3  // and capacity $m$.
4  // The objects are assumed to be sorted by $p[i]/w[i]$ in nonincreasing order.
5  {
6      for $i := 1$ to $n$ do $x[i] := 0$;
7      $i := 1$; $fp_2 := 0$; $m' := m$;
8      while ($m' \geq w[i]$) do { // Greedy method.
9          $x[i] := 1$; $fp_2 := fp_2 + p[i]$; $m' := m' - w[i]$; $i := i + 1$;
10     }
11     Find $j$ such that $p[j] = \max(p[1:n])$; // Object $j$ has the max profit.
12     if ( $p[j] > fp_2$ and $w[j] \leq m$ ) then { // Choose the object $j$.
13         for $i := 1$ to $n$ do $x[i] := 0$;
14         $x[j] := 1$; $fp_2 := p[j]$;
15     }
16 }
```

- This revised algorithm adds lines 11-15 for the possibility of choosing the object with the largest profit.

# 0/1 Knapsack Problem – The Profit

- In the preceding algorithm, let $i = h$ when the while loop on line 8 terminates.
- At this time, we have

$$fp_1 = \sum_{i=1}^{h-1} p_i < p^* < fp_1 + p_h \cdot \frac{m'}{w_h} < fp_1 + p_h.$$

- Consider two cases
  - Case 1: $p_h < fp_1$ then

$$p^* < fp_1 + p_h < 2 \cdot fp_1 \leq 2 \cdot fp_2.$$

  - Case 2: $p_h > fp_1$, then

$$p^* < fp_1 + p_h < 2 \cdot p_h \leq 2 \cdot \max\{p_i\} \leq 2 \cdot fp_2.$$

- Thus, we have the following lemma.

# 0/1 Knapsack Problem – Bound of The Profit

**Lemma 10.1.3.**

Given a 0/1 knapsack problem, let the optimal profit be $p^*$ and the profit found by Algorithm (10.1.2) be $fp_2$, then

$$\frac{p^*}{fp_2} \leq 2. \tag{10.1.2}$$

- The greedy algorithm to solve the knapsack problem always finds a profit $fp_2$ such that $\frac{p^*}{2} < fp_2 < p^*$.
- This algorithm finds an approximate solution given the bound above. Though it is not an optimal solution, it has very low time complexity.

# Approximation Algorithms

- There are no known polynomial time algorithms to solve $\mathcal{NP}$-complete problems.
- Solving these problems can take a long time if the problem size is not small.
- But, there are many practical problems that are $\mathcal{NP}$-complete.
- Heuristics might be used with existing algorithms to reduce solution time.
  - Backtracking and branch and bound algorithms.
  - The solution quality can vary significantly from instance to instance.
  - Exponential time complexity can still take formidable time.
- Instead of finding the optimal solution, a different approach is to find an approximate solution, which is a feasible solution with value close the optimal solution.
- An approximation algorithm for a problem $\mathcal{Q}$ is an algorithm that generates approximate solutions for $\mathcal{Q}$.

# Approximation Algorithms — Definitions

- Let $\mathcal{Q}$ be a problem such as the knapsack (or the traveling salesperson) problem.
- Let $I$ is an instance of problem $\mathcal{Q}$ and $F^*(I)$ be the value of an optimal solution to $I$.
- An approximation algorithm generally produces a feasible solution to $I$ whose value $\hat{F}(I)$ is less than (greater than) $F^*(I)$ if $\mathcal{Q}$ is a maximization (minimization) problem.

### Definition. 10.1.4. Absolute approximation.

$\mathcal{A}$ is an absolute approximation algorithm for problem $\mathcal{Q}$ if and only if for every instance $I$ of $\mathcal{Q}$, $|F^*(I) - \hat{F}(I)| \le k$ for some constant $k$.

### Definition. 10.1.5.

$\mathcal{A}$ is an $f(n)$-approximate algorithm for problem $\mathcal{Q}$ if and only if for every instance $I$ of size $n$, $|F^*(I) - \hat{F}(I)|/F^*(I) \le f(n)$ for $F^*(I) > 0$.

# Approximation Algorithms — Definitions, II

### Definition. 10.1.6.

An $\epsilon$-approximate algorithm is an $f(n)$-approximate algorithm for which $f(n) \le \epsilon$ for some constant $\epsilon$.

- Note that for maximization problems, $|F^* - \hat{F}(I)|/F^* \le 1$ for every feasible solution to $I$.
    - Thus, $\epsilon < 1$ is usually required for $\epsilon$-approximate algorithms.
- In the following, we assume $\epsilon$ is an input to algorithm $\mathcal{A}$.

### Definition. 10.1.7.

$\mathcal{A}(\epsilon)$ is an approximation scheme if and only if for every given $\epsilon > 0$ and problem instance $I$, $\mathcal{A}(\epsilon)$ generates a feasible solution such that $|F^*(I) - \hat{F}(I)|/F^* \le \epsilon$. ($F^* > 0$ is assumed.)

# Approximation Algorithms — Definitions, III

**Definition. 10.1.8.**

An approximation scheme is a polynomial time approximation scheme if and only of for every fixed $\epsilon > 0$, it has computing time that is polynomial in the problem size.

**Definition. 10.1.9.**

An approximation scheme whose computing time is a polynomial both in problem size and in $1/\epsilon$ is a fully polynomial time approximation scheme.

- For most $\mathcal{NP}$-complete problems, it can be shown the absolute approximation algorithms exist only if $\mathcal{P}$=$\mathcal{NP}$-complete.
    - For certain $\mathcal{NP}$-complete problems, the existence of $f(n)$-approximate algorithm is also shown only when $\mathcal{P}$=$\mathcal{NP}$-complete.

---

# Absolute Approximations

- There are very few $\mathcal{NP}$-hard optimization problems for which polynomial time absolute approximation algorithms are known.
- The problem of determining the minimum number of colors to color a planar graph is an exception.
    - It has been proven that every planar graph is four colorable.
    - One can also determine a planar graph is zero, one or two colorable.

**Algorithm. 10.1.10. Planar Graph Coloring.**

```
Algorithm AColor($G$)
// Approximate algorithm to determine minimum color for planar graph $G(V, E)$.
{
    if ($V = \emptyset$) then return $0$;
    else if ($E = \emptyset$) then return $1$;
    else if ($G$ is bipartite ) then return $2$;
    else return $4$;
}
```

# Planar Graph Coloring

- The time complexity of Algorithm (10.1.10) is dominated by line 6 which checks if the graph is bipartite.
- Checking the bipartite property of a graph can be done in $\mathcal{O}(|V| + |E|)$ time.
- Thus, Algorithm (10.1.10) is a polynomial time algorithm.
- Note that the planar graph coloring problem is $\mathcal{NP}$-hard since three color decision problem is $\mathcal{NP}$-complete.
- Algorithm (10.1.10) does not check for three color solution, thus avoiding the long execution time by returning an approximate solution.
- Algorithm (10.1.10) is an absolute approximation algorithm because $|F^*(I) - \hat{F}(I)| \leq 1$.

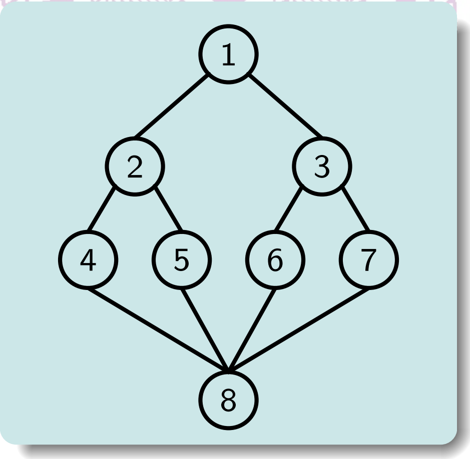
---

# Bipartite Graph

### Definition. 10.1.11. Bipartite Graph.

An undirected graph $G(V, E)$ is bipartite if $V$ can be partitioned into two disjoint sets $V_1$ and $V_2 = V - V_1$ such that no two vertices in $V_1$ are adjacent, and no two vertices in $V_2$ are adjacent.

- Example: The graph below is bipartite with $V_1 = \{1, 4, 5, 6, 7\}$ and $V_2 = \{2, 3, 8\}$.



- Determine if a graph is bipartite can be done in $\mathcal{O}(|V| + |E|)$ time.

# Maximum Programs Stored Problem

- Given $n$ programs and two storage devices. The $i$th program is of length $l_i$ and each storage device has capacity of $L$. The maximum programs stored problem is to determine the maximum number of programs that can be stored on these two storage devices without splitting any program.
- This maximum programs stored problem is $\mathcal{NP}$-hard because of the following.
- Example: Four programs with the lengths as $(l_1, l_2, l_3, l_4) = (2, 4, 5, 6)$ and storage device capacity $L = 10$.
  - The optimal solution is 4, which can be achieved by storing programs 1 and 4 on one device, and programs 2 and 3 on the other device.

**Theorem. 10.1.12.**

Partition problem $\propto$ maximum programs stored problem.

- Proof please see textbook [Horowitz] p. 581.

# Maximum Programs Stored Problem, II

- Assume the lengths of the $n$ program is stored in array $l[1 : n]$.
- Sort array $l[1 : n]$ in nondecreasing order, $l[i] \le l[i + 1]$, $1 \le i \le n$.

**Algorithm. 10.1.13. Approximate algorithm to store programs.**

```
Algorithm PStore($l, n, L$)
// Store $n$ program with $l[1 : n]$ lengths to 2 devices.
{
      $i := 1$;
      for $j := 1$ to 2 do { // store to device 1 then 2
            $sum := 0$; // Amount of device used.
            while ($sum + l[i] \le L$) do {
                  write (" store program ", $i$, " on device ", $j$);
                  $sum := sum + l[i]$; $i := i + 1$;
                  if $i > n$ then return;
            }
      }
}
```

# Maximum Programs Stored Problem, III

### Theorem 10.1.14.

Let $I$ be any instance of the maximum programs stored problem. Let $F^*(I)$ be the maximum number of programs that can be stored on two devices each with length $L$. Let $\hat{F}(I)$ be the number of programs stored using the function `PStore`. Then $|F^*(I) - \hat{F}(I)| \le 1$.

**Proof.** Consider the case that only one device with length $2L$ is used to store the programs, and $p$ programs are stored. Then $p > F^*(I)$ and $\sum_{i=1}^{p} l_i \le 2L$. Let $j$ be the largest index such that $\sum_{i=1}^{j} l_i \le L$. We must have $j \le p$ and that `PStore` assign the first $j$ programs to device 1. Also,

$$\sum_{i=j+1}^{p-1} l_i \le \sum_{i=j+2}^{p} l_i \le L.$$

Hence, `PStore` assigns at least $j+1, j+2, \cdots, p-1$ to device 2. So, $\hat{F}(I) \ge p-1$ and $|F^*(I) - \hat{F}(I)| \le 1$. ☐

- Algorithm `PStore` can be extended to be a $k-1$ absolute approximation algorithm for the case of $k$ devices.

# $\mathcal{NP}$-hard Absolute Approximations

- For a majority of the $\mathcal{NP}$-hard problems, however, the polynomial absolute approximation algorithm exists if and only if the original program has a polynomial time algorithm.
- For example, we have the following theorem.

### Theorem. 10.1.15.

The absolute knapsack problem is $\mathcal{NP}$-hard.

**Proof.** Suppose that we have a polynomial time algorithm to find $|F^*(I) - \hat{F}(I)| \le k$ for every instance $I$ and a fixed $k$. Let $(p_i, w_i)$, $1 \le i \le n$ and $m$ be the instance. Furthermore, we assume $p_i$ are integers. Form a new instance $I'$ by $((k+1)p_i, w_i)$, $1 \le i \le n$, and $m$. Note that any feasible solution for $I$ is also a feasible solution for $I'$, and $F^*(I') = (k+1)F^*(I)$ and $I$ and $I'$ have the same optimal solutions. Since $p_i$ are integers, the feasible solutions of $I'$ must have difference $\ge (k+1)$ due to the way $I'$ is constructed. Now, suppose the absolute algorithm $A$ finds the optimal solution such that $|F^*(I') - \hat{F}(I')| \le k$, then $\hat{F}(I')$ must be $F^*(I')$. Thus, the polynomial algorithm can be used to find the optimal solution, which is not possible. ☐

# $\mathcal{NP}$-hard Absolute Approximations, II

- Another example of absolute approximation algorithm is $\mathcal{NP}$-hard.

## Theorem. 10.1.16.

Max clique $\propto$ absolute approximation max clique.

**Proof.** Suppose there is an absolute approximation algorithm that finds a solution such that $|F^*(I) - \hat{F}(I)| \leq k$. For a given graph $G(V, E)$ construct a new graph $G^{'}(V^{'}, E^{'})$ so that $G^{'}$ consists of $(k+1)$ copies of $G$ connected together such that there is an edge between every two vertices in distinct copies of $G$. That is, if $V = \{v_1, v_2, \cdots, n\}$, then

$$V^{'} = \bigcup_{i=1}^{k+1} \{v_1^i, v_2^i, \cdots, v_n^i\},$$

and $$E^{'} = (\bigcup_{i=1}^{k+1} \{(v_p^i, v_r^i) | (v_p, v_r) \in E\}) \bigcup \{(v_p^i, v_r^j) | i \neq j\}$$

Then the maximum clique size is $q$ if and only if the maximum clique size if $G^{'}$ is $(k+1)q$. Furthermore, any clique in $G^{'}$ that is within $k$ of the maximum clique in $G^{'}$ must contain a subclique of size $q$ in $G$. Thus, we can use this absolute approximation algorithm to find the maximum clique of the original problem in polynomial time since constructing $G^{'}$ is of polynomial time. □

---

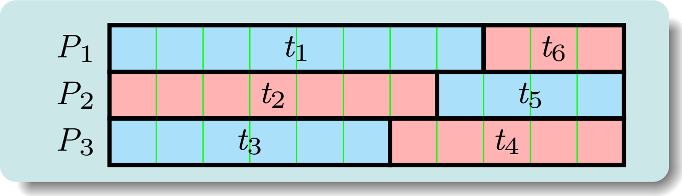
# $\epsilon$-Approximations

- Given a set of $n$ tasks with processing time $t_i$ each and $m$ identical processors, the minimum finish time schedule assign the tasks to the processors to achieve the minimum finish time.
- This minimum finish time scheduling problem has been shown to be $\mathcal{NP}$-hard.
- In this section we study a polynomial time scheduling algorithm.

## Definition. 10.1.17. LPT Schedule.

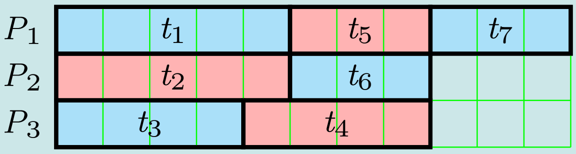
An LPT schedule is one that is the result of an algorithm that, whenever a processor becomes free, assigns to that processor a task whose processing time is the longest of those tasks not yet assigned. Ties are broken in an arbitrary manner.

- Example: $m = 3$, $n = 6$ and $(t_1, t_2, t_3, t_4, t_5, t_6) = (8, 7, 6, 5, 4, 3)$. The following is the result of a LPT schedule, which is also an optimal solution.

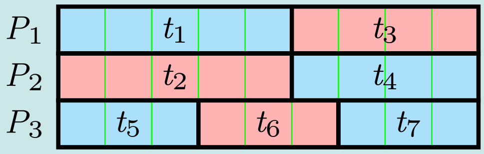


# LPT Scheduling

- Example 2: $m = 3$, $n = 7$ and $(t_1, t_2, t_3, t_4, t_5, t_6, t_7) = (5, 5, 4, 4, 3, 3, 3)$. The LPT schedule and the optimal schedule are shown below.



LPT schedule.



Optimal schedule.

## Theorem. 10.1.18.

Let $F^*(I)$ be the finish time of an optimal $m$-processor schedule for instance $I$ of the task scheduling problem. Let $\hat{F}(I)$ be the finish time of an LPT schedule for the same instance. Then

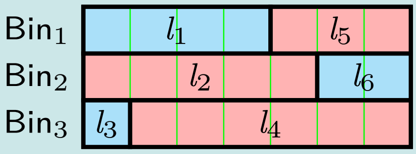
$$\frac{|F^*(I) - \hat{F}(I)|}{|F^*(I)|} \le \frac{1}{3} - \frac{1}{3m}. \tag{10.1.3}$$

**Proof.** See textbook [Horowitz] pp. 586-587. ☐

---

# Bin Packing Problem

- Given $n$ objects of $l_i$ units each to be placed in bins with equal capacity $L$. The bin packing problem is to determine the minimum number of bins to accommodate all objects.
- Example: $n = 6$, $(l_1, l_2, l_3, l_4, l_5, l_6) = (4, 5, 1, 6, 3, 2)$ and $L = 7$. An optimal solution is:



- This bin packing problem has many applications. The followings are examples.
  - $n$ tasks with $t_i$ processing time and all tasks must be completed before deadline $L$. Find the minimum number of processors, $m$.
  - $n$ programs with $l_i$ lengths each to be stored on devices with capacity $L$. Find the minimum number of storage devices, $m$.

# Bin Packing Problem, II

**Theorem 10.1.19.**

The bin packing problem is $\mathcal{NP}$-hard.

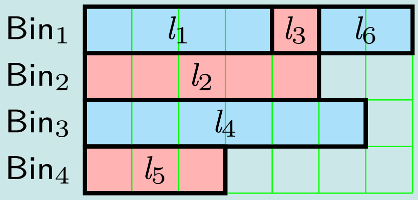
**Proof.** Let $\{a_1, a_2, \cdots, a_3\}$ be an instance of partition problem. A bin packing problem can be constructed by assigning $l_i = a_i$, $1 \le i \le n$, and $L = \sum_{i=1}^{n} a_i$. The minimum number of bins is 2 and the solution can be found if there is a partition for $\{a_1, a_2 \cdots, a_n\}$. Since the partition problem is $\mathcal{NP}$-hard, the bin packing problem is also $\mathcal{NP}$-hard. □

- Thus, finding the optimal solution for the bin packing problem can take long time if the number of input, $n$, is large.
- Heuristics can be used to find good feasible solutions.
  - These solutions are usually not optimal.

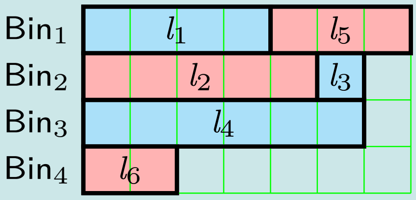
---

# Bin Packing Problem, III

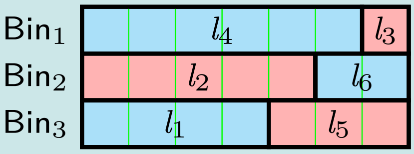
- Four heuristics are possible:
  1. First Fit (FF): Pack objects sequentially from 1 to $n$. All bins are initially filled to level zero. To pack object $i$, find the least index $j$ such that bin $j$ is filled to a level $r$, $r \le L - l_i$. Pack object $i$ into bin $j$. Bin $j$ is now filled to the level $r + l_i$.
  2. Best Fit (BF): The initial conditions on the bins and objects are the same as above. To pack object $i$, find the least $j$ such that bin $j$ is filled to a level $r$, $r \le L - l_i$ and is as large as possible. Pack object $i$ into bin $j$. Bin $j$ is now filled to the level $r + l_i$.
  3. First Fit Decreasing (FFD): Reorder the objects is a nonincreasing order, then use First Fit to pack the objects.
  4. Best Fit Decreasing (BFD): Reorder the objects is a nonincreasing order, then use Best Fit to pack the objects.
- Example: $n = 6$, $(l_1, l_2, l_3, l_4, l_5, l_6) = (4, 5, 1, 6, 3, 2)$, and $L = 7$.



FF.



BF.



FFD and BFD.

# Bin Packing Problem, IV

## Theorem. 10.1.20.

Let $I$ be an instance of the bin packing problem and $F^*(I)$ be the minimum number of bins needed for this instance. The packing generated by either FF or BF uses no more than

$$\frac{17}{10}F^*(I) + 2 \tag{10.1.4}$$

bins. The packing generated by either FFD or BFD used no more than

$$\frac{11}{9}F^*(I) + 4 \tag{10.1.5}$$

bins. These bounds are the best possible for the respective algorithms.

**Proof.** See the paper: D. Johnson, A. Demers, J. Ullman, M. Garey, and R. Graham, "Worst-case Performance Bounds for Simple One-Dimensional Packing Algorithms," *SIAM Journal on Computing* 3, No. 4, 1974, pp. 299-325. ☐

- Note these are worst-case bounds.
  - For some instances, these heuristics are capable of generating the optimal solutions.
- For large $n$, the FFD and BFD heuristics have the smaller bounds.

# $\mathcal{NP}$-hard $\epsilon$-approximation Problems

- Many $\mathcal{NP}$-hard optimization problems their corresponding $\epsilon$-approximation problems are also $\mathcal{NP}$-hard.
- Few examples are given here.

## Theorem. 10.1.21.

Hamiltonian cycle problem $\propto$ $\epsilon$-approximation traveling problem.

- Proof please see textbook [Horowitz] p. 591.

## Theorem. 10.1.22.

Partition problem $\propto$ $\epsilon$-approximation integer programming problem.

- Proof please see textbook [Horowitz] p. 592.

## Theorem. 10.1.23.

Hamiltonian cycle problem $\propto$ $\epsilon$-approximation quadratic assignment problem.

- Proof please see textbook [Horowitz] p. 593.

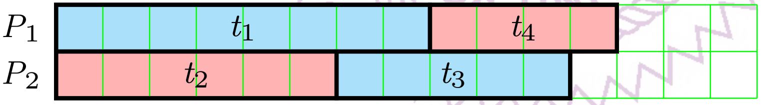
# Polynomial Time Approximation Schemes

- A different approximation scheme of the independent task scheduling problem.

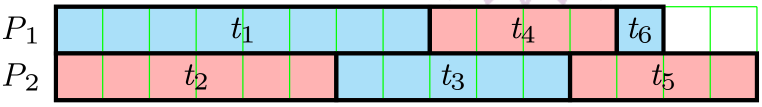
## Algorithm 10.1.24. Scheduling by Graham

```
Algorithm Graham($n, m, k, t$)
// Schedule $n$ tasks with processing time $t[1:n]$ on $m$ processors.
{
    Find the optimal schedule for the $k$ longest tasks ;
    Perform LPT scheduling for the rest of the tasks ;
}
```

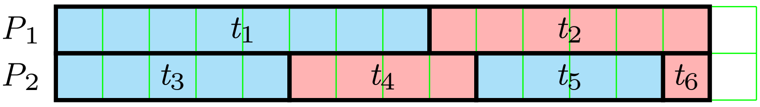
- Example: $n = 6$, $m = 2$, $(t_1, t_2, t_3, t_4, t_5, t_6) = (8, 6, 5, 4, 4, 1)$.



Optimal for 4 tasks.



Complete schedule.



Optimal schedule.

---

# Polynomial Time Approximation Schemes, II

## Theorem. 10.1.25. Graham Scheduling.

Let $I$ be an $m$-processor instance of the scheduling problem. Let $F^*(I)$ be the finish time of an optimal schedule for $I$ and let $\hat{F}(I)$ be the finish time of the schedule generated by the algorithm Graham. Then,

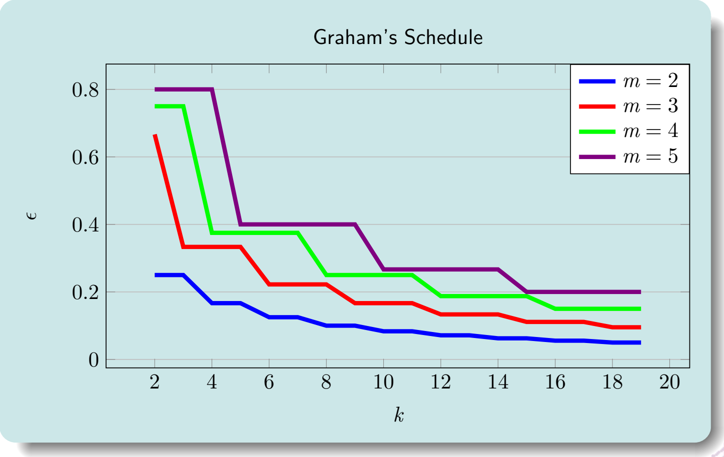
$$\frac{|F^*(I) - \hat{F}(I)|}{F^*(I)} \leq \frac{1 - 1/m}{1 + \lfloor k/m \rfloor}. \qquad (10.1.6)$$

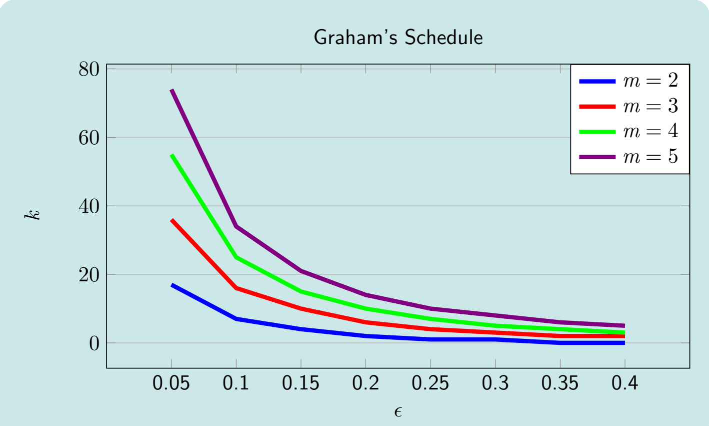
- Proof please see textbook [Horowitz] pp. 598-599.
- Given any $\epsilon$, one can find

$$k \geq \frac{m-1}{\epsilon} - m \qquad (10.1.7)$$

then the schedule generated is $\epsilon \cdot F^*(I)$.

# Polynomial Time Approximation Schemes, III





- In the Graham's algorithm $\epsilon$ can be made small, but then $k$ can be large.
- The first part of the Graham's algorithm, line 4, can take $\mathcal{O}(m^k)$ time.
- Before applying `Graham`'s algorithm, the input needs to be sorted, time complexity $\mathcal{O}(n \lg n)$.
- Thus, the total time complexity is $\mathcal{O}(n \lg n + m^k)$.
  - This is not exactly a polynomial time algorithm for large $k$.

# Solving $\mathcal{NP}$-complete Problems

- Finding solutions for $\mathcal{NP}$-complete or $\mathcal{NP}$-hard problems can take formidable amount of time.
- Approximation algorithms do not attempt to find the optimal solution but to find a feasible solution close to the optimal one.
  - The bound, if can be derived, is of great value.
- Basic methods for approximate algorithms are the ones we have studied
  - Divide-and-conquer
  - Greedy method
  - Dynamic programming
  - Local search instead of all space search
  - The key is the bounding function.
- Other heuristic approaches have been developed
  - Construction heuristics
  - Local search heuristics
  - Simulated annealing
  - Genetic algorithms
  - Tabu search

# Summary

- Approximation algorithms.
- Absolution approximations.
    - Planar graph coloring problem.
    - Maximum programs stored problem.
    - $\mathcal{NP}$-hardness.
- $\epsilon$-approximations.
    - Scheduling problem.
    - Bin packing problem.
    - $\mathcal{NP}$-hardness.
- Polynomial time approximation scheme.
    - Graham's algorithm.