Approximation Algorithms: Basic Concepts

Stefano Leonardi

Sapienza University of Rome

Thanks to Lap-Chi Lau
NP-completeness

There are many polynomial time solvable optimization problems. e.g. maximum matching, min-cost flow, minimum cut, etc.

However, there are much more optimization problems that are NP-complete: we do not know how to solve in polynomial time. e.g. traveling salesman, graph colorings, maximum independent set, set cover, maximum clique, maximum cut, minimum Steiner tree, satisfiability, etc.
**Vertex Cover**

**Vertex cover**: a subset of vertices which "covers" every edge. An edge is **covered** if one of its endpoint is chosen.

**The Minimum Vertex Cover Problem**: Given a graph find a vertex cover with minimum number of vertices. (optimization problem)
Some Alternatives

- Special graph classes
  
  e.g. vertex cover in bipartite graphs, perfect graphs.

- Fixed parameter algorithms
  
  find a vertex cover of size $k$ efficiently for small $k$.

- Average case analysis
  
  find an algorithm which works well on average.

- Approximation algorithms
  
  find an algorithm which return solutions that are guaranteed to be close to an optimal solution.
Approximation Algorithms

Key: provably close to optimal.

Let OPT be the value of an optimal solution, and let SOL be the value of the solution that our algorithm returned.

**Additive approximation algorithms**: SOL $\leq$ OPT + c for some constant c.
Very few examples known:
edge coloring, minimum maximum-degree spanning tree, bin packing

**Constant factor approximation algorithms**: SOL $\leq$ c OPT for some constant c.
Many more examples known.
Different approximation factors

**Constant approximation algorithms**: SOL \( \leq c \) OPT

- e.g.: Vertex Cover, TSP, Max SAT, Steiner Tree, Facility Location, Max Cut

**Logarithmic approximation algorithms**: SOL \( = \Omega(\log n)^c \) OPT

- e.g.: Set Cover, Multi-Cut, Dominating Set,..

**Polynomial approximation**: SOL \( = \Omega(n^c) \) OPT, \( c \leq 1 \)

- e.g: Max Clique, Independent Set, Coloring

**Polynomial Approximation Schemes**: SOL \( \leq (1+\varepsilon) \) OPT, for each \( \varepsilon > 0 \)

- Fully Polynomial Approximation schemes if running time \( O(poly(1/\varepsilon)) \)

- e.g.: Knapsack, Scheduling, Budgeted MST, Euclidean TSP, Bin packing
Vertex Cover: 2 apx algorithm

Fix a maximum matching. Call the vertices involved black.
Since the matching is maximum, every edge must have a black endpoint.
So, by choosing all the black vertices, we have a vertex cover.

SOL <= 2 * size of a maximum matching
Vertex Cover: 2-apx algorithm

What about an optimal solution?

Each edge in the matching has to be covered by a different vertex!

Lower bound to the optimal solution:
OPT \geq \text{size of a maximum matching}

So, SOL \leq 2 \text{ OPT}, and we have a 2-approximation algor.
**Vertex Cover**

**Approximate min-max theorem:**
Maximum matching $\leq$ minimum vertex cover $\leq 2 \times$ maximum matching

**Is the analysis tight? Yes.** Consider for instance a bipartite graph.

**Major open question:**
Can we obtain a $c$-approximation algorithm, $c < 2$?

**Hardness result:**
It is NP-complete even to *approximate* within a factor of 1.36!!
Approximation Algorithms for (Min) Traveling Salesman Problem (TSP)
The Hamiltonian Path Problem

**Hamiltonian Path Problem:**
Given an undirected graph, find a cycle visiting every vertex exactly once.

**Eulerian Path Problem:**
Given an undirected graph, find a walk visiting every edge exactly once. Notice that in a walk some vertices may have been visited more than once.

The Eulerian Path problem is polynomial time solvable.
A graph has an Eulerian path if and only if every vertex has an even degree.

The Hamiltonian Path problem is NP-complete.
The Traveling Salesman Problem

Traveling Salesman Problem (TSP):
Given a complete graph with nonnegative edge costs,
Find a minimum cost cycle visiting every vertex exactly once.

Given a number of cities and the costs of traveling from any city to any other city, what is the cheapest round-trip route that visits each city exactly once and then returns to the starting city?

One of the most well-studied problem in combinatorial optimization.
Inapproximability of Traveling Salesman Problem

**Theorem:** There is no constant factor approximation algorithm for TSP, unless P=NP.

Idea: Use the Hamiltonian path problem.

- For each edge, we add an edge of cost 1.
- For each non-edge, we add an edge of cost \( nk \).

- If there is a Hamiltonian path, then there is a cycle of cost \( n \).
- If there is no Hamiltonian path, then every cycle has cost greater than \( nk \).

So, if you have a k-approximation algorithm for TSP, one just needs to check if the returned solution is at most \( nk \).

- If yes, then the original graph has a Hamiltonian path.
- Otherwise, the original graph has no Hamiltonian path.
Inapproximability of Traveling Salesman Problem

**Theorem:** There is no constant factor approximation algorithm for TSP, unless P=NP.

This type of theorem is called “hardness result” in the literature. Just like their names, usually they are very hard to obtain.

- If there is a Hamiltonian path, then there is a cycle of cost $n$.
- If there is no Hamiltonian path, then every cycle has cost greater than $nk$.

The strategy is usually like this.
This creates a gap between yes and no instances.
The bigger the gap, the problem is harder to approximate.
Approximation Algorithm for Metric TSP

**Metric Traveling Salesman Problem (metric TSP):**
Given a complete graph with edge costs satisfying triangle inequalities, Find a minimum cost cycle visiting every vertex exactly once.

For example, think of cost of an edge as the distance between two points.

How could triangle inequalities help in finding approximation algorithm?
Lower Bounds for TSP

What can be a good lower bound to the cost of TSP?

A tour contains a matching.

Let OPT be the cost of an optimal tour, since a tour contains two matchings, the cost of a minimum weight perfect matching is at most OPT/2.

So, the cost of a minimum spanning tree is at most OPT.
Spanning Tree and TSP

Let the thick edges have cost 1, 
And all other edges have cost greater than 1.

So the thick edges form a minimum spanning tree.

But it doesn’t look like a Hamiltonian cycle at all!

Consider a Hamiltonian cycle. 
The costs of the edges which are not in the 
minimum spanning tree might have very high costs.

Not really! Each such edge has cost at most 2 
because of the triangle inequality.
Spanning Tree and TSP

How to formalize the idea of “following” a minimum spanning tree?
Spanning Tree and TSP

How to formalize the idea of “following” a minimum spanning tree?

Key idea: double all the edges and find an Eulerian tour.

This graph has cost $2\text{MST}$. 
Spanning Tree and TSP

How to formalize the idea of “following” a minimum spanning tree?

**Key idea:** double all the edges and find an Eulerian tour.

This graph has cost $2\text{MST}$. 
Strategy: shortcut this Eulerian tour.
By triangle inequalities, the shortcut tour is not longer than the Eulerian tour.

Each directed edge is used exactly once in the shortcut tour.
A 2-Approximation Algorithm for Metric TSP

(Metric TSP – Factor 2)
1. Find an MST, T, of G.
2. Double every edge of the MST to obtain an Eulerian graph.
3. Find an Eulerian tour, T*, on this graph.
4. Output the tour that visits vertices of G in the order of their first appearance in T*. Let C be this tour.
   (That is, shortcut T*)

Analysis:
1. \(\text{cost}(T) \leq \text{OPT}\) (because MST is a lower bound of TSP)
2. \(\text{cost}(T^*) = 2\text{cost}(T)\) (because every edge appears twice)
3. \(\text{cost}(C) \leq \text{cost}(T^*)\) (because of triangle inequalities, shortcutting)
4. So, \(\text{cost}(C) \leq 2\text{OPT}\)
Better approximation?

There is a 1.5 approximation algorithm for metric TSP.

**Major open problem:** Improve this to 4/3?

**An aside:** hardness result is not an excuse to stop working, but to guide us to identify interesting cases.
Matching

**Minimum Weight Perfect Matching**
Given a complete graph with $n$ vertices and edge costs, Find a perfect matching of minimum total cost

Matching is solvable in polynomial time

Compare with vertex cover

How could matching helps in finding approximation algorithm?

Note: if number of vertices odd then there is at least one unmatched vertex
Lower Bounds for TSP

What can be a good lower bound to the cost of TSP?

A tour contains a matching.

Let OPT be the cost of an optimal tour, since a tour contains two matchings, the cost of a minimum weight perfect matching is at most $\frac{OPT}{2}$.

A tour contains a spanning tree.

So, the cost of a minimum spanning tree is at most OPT.
Spanning Tree matching and TSP

How to generalize the idea of “following” a minimum spanning tree?
Spanning Tree matching and TSP

How to generalize the idea of “following” a matching and a minimum spanning tree?

IDEA: obtain an Eulerian graph using a spanning tree and a matching
Spanning Tree matching and TSP

Eulerian graph: each vertex has even degree

Christofides: apply matching to vertices with odd degree in the minimum spanning tree!
**A Christofides’ Algorithm for Metric TSP**

*(Metric TSP – Factor 3/2)*

1. Find an MST, $T$, of $G$.
2. Find a matching $M$ among **odd** degree vertices.
3. Find an Eulerian tour $E$, on the graph with edges from $T$ and $M$.
4. Output the tour $C$ that visits vertices of $G$ in the order of their first appearance. Let $C$ be this tour.

*(That is, shortcut $T^*$)*

**Analysis:**

1. $\text{cost}(T) \leq \text{OPT}$ (because MST is a lower bound of TSP)
2. $\text{cost}(M) \leq 0.5 \text{OPT}$ (because every edge appears twice)
3. $\text{cost}(C) \leq \text{cost}(T) + \text{cost}(M)$ (because of triangle inequalities, shortcutting)
4. So, $\text{cost}(C) \leq 1.5 \text{OPT}$
Better approximation?

Christofides: there is a 1.5 approximation algorithm for metric TSP.
Approximation criteria

An algorithm is a $c$-approximation algorithm if

- $\text{SOL} \leq c \text{OPT}$ for a minimization problem
- $\text{SOL} \geq c \text{OPT}$ for a maximization problem

An algorithm $A$ is an **approximation scheme** if for every $\epsilon > 0$, $A$ runs in polynomial time (which may depend on $\epsilon$) and return a solution:

- $\text{SOL} \leq (1+\epsilon)\text{OPT}$ for a minimization problem
- $\text{SOL} \geq (1-\epsilon)\text{OPT}$ for a maximization problem
Knapsack

Given $n$ objects with
size($a_i$) and profit($a_i$), $i=1,2,…,n$

Knapsack can only hold a
• total weight of $B$

Goal: to pick a subset which can fit into the knapsack
and maximize the value of this subset.
Knapsack

Given a set $S = \{a_1, \ldots, a_n\}$ of objects,
with specified sizes and profits, $\text{size}(a_i)$ and $\text{profit}(a_i)$,
and a knapsack capacity $B$, find a subset of objects whose total size is bounded by $B$ and total profit is maximized.

Assume $\text{size}(a_i)$, $\text{profit}(a_i)$, and $B$ are all integers.

Integer programming formulation

Max $\sum_i \text{profit}(a_i)x_i$

s.t. $\sum_i \text{size}(a_i)x_i \leq B$, $x_i \in \{0,1\}$
Greedy methods

General greedy method:
Sort the objects by some rule,
and then put the objects into the knapsack according to this order

Sort
• G1: Sort by object size in non-decreasing order
• G2: Sort by profit in non-increasing order
• G3: Sort by profit/object size in non-increasing order

Fact: All previous ordering do not work to find a good approximate solution

Theorem: For all instances G3 returns a solution SOL-G3 such that

\[ \max(\text{profit}(a_i), \text{SOL-G3}) \geq 1/2 \text{ OPT} \]

Exercise: prove the Fact and the Theorem
Dynamic Programming

Dynamic programming is just exhaustive search with polynomial number of subproblems.

We only need to compute each subproblem once, and each subproblem is looked up at most a polynomial number of times, and so the total running time is at most a polynomial.
Dynamic Programming for Knapsack

Suppose we have considered object 1 to object i. We want to remember what profits are achievable. For each achievable profit, we want to minimize the size.

Let $S(i,p)$ denote a subset of $\{a_1, \ldots, a_i\}$ whose total profit is exactly $p$ and total size is minimized. Let $A(i,p)$ denote the size of the set $S(i,p)$.

For example, $A(1,p) = \text{size}(a_1)$ if $p = \text{profit}(a_1)$.
Otherwise $A(1,p) = \infty$ (if $p \neq \text{profit}(a_1)$).
Recurrence

Remember: $A(i,p)$ denote the minimum size to achieve profit $p$ using objects from 1 to $i$.

Goal: we know $A(i,q)$ for all $q$ and we want to compute $A(i+1,p)$

Two possibilities:

If we do not choose object $i+1$:
then $A(i+1,p) = A(i,p)$.

If we choose object $i+1$:
then $A(i+1,p) = \text{size}(a_{i+1}) + A(i,p-\text{profit}(a_{i+1}))$ if $p > \text{profit}(a_{i+1})$

$A(i+1,p) = \text{minimum of these two values.}$
An Example

Remember: \( A(i,p) \) denote the minimize size to achieve profit \( p \) using objects from 1 to \( i \).

Optimal Solution: \( \max \{ p \mid A(n,p) \leq B \} \) where \( B \) is the size of the knapsack.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>2</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
</tbody>
</table>

\( \text{size(a1)} = \text{2}, \text{profit(a1)} = \text{4}; \text{size(a2)} = \text{3}, \text{profit(a2)} = \text{5}; \text{size(a3)} = \text{2}, \text{profit(a3)} = \text{3}; \text{size(a4)} = \text{1}, \text{profit(a4)} = \text{2} \)
**An Example**

\[
A(i+1,p) = \min\{A(i,p), \text{size}(a_{i+1}) + A(i,p-\text{profit}(a_{i+1}))\}.
\]

\[
A(2,p) = \min\{A(1,p), A(1,p-5) + 3\}.
\]

Size and profit of items:
- size\((a_1) = 2\), profit\((a_1) = 4\);
- size\((a_2) = 3\), profit\((a_2) = 5\);
- size\((a_3) = 2\), profit\((a_3) = 3\);
- size\((a_4) = 1\), profit\((a_4) = 2\);

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>2</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
</tbody>
</table>
An Example

\[ A(i+1,p) = \min\{A(i,p), \text{size}(a_{i+1}) + A(i,p-\text{profit}(a_{i+1}))\}. \]

\[ A(3,p) = \min\{A(2,p), A(2,p-3)+2\}. \]

size(a1)=2, profit(a1)=4; size(a2)=3, profit(a2)=5; size(a3)=2, profit(a3)=3; size(a4)=1, profit(a4)=2

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>2</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>2</td>
<td>3</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>5</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
</tbody>
</table>
An Example

A(\(i+1,p\)) = \min\{A(i,p), size(a_{i+1}) + A(i,p-profit(a_{i+1}))\}.

A(4,p) = \min\{A(3,p), A(3,p-2)+1\}.

size(a1)=2, profit(a1)=4; size(a2)=3, profit(a2)=5; size(a3)=2, profit(a3)=3; size(a4)=1, profit(a4)=2

\[
\begin{array}{cccccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\
\hline
1 & 0 & \infty & \infty & \infty & \infty & 2 & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty \\
2 & 0 & \infty & \infty & \infty & 2 & 3 & \infty & \infty & \infty & 5 & \infty & \infty & \infty & \infty \\
3 & 0 & \infty & \infty & 2 & 2 & 3 & \infty & 4 & 5 & 5 & \infty & \infty & 7 & \infty \\
4 & 0 & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty \end{array}
\]
An Example

\[ A(i+1,p) = \min\{A(i,p), \text{size}(a_{i+1}) + A(i,p-\text{profit}(a_{i+1}))\}. \]

\[ A(4,p) = \min\{A(3,p), A(3,p-2)+1\}. \]

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>2</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>2</td>
<td>3</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>5</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>∞</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>∞</td>
<td>∞</td>
<td>7</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>∞</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>∞</td>
<td>8</td>
</tr>
</tbody>
</table>

\(\text{size}(a1)=2, \text{profit}(a1)=4; \ \text{size}(a2)=3, \text{profit}(a2)=5; \ \text{size}(a3)=2, \text{profit}(a3)=3; \ \text{size}(a4)=1, \text{profit}(a4)=2\)
An Example

Remember: $A(i,p)$ denote the minimize size to achieve profit $p$ using objects from 1 to $i$.

Optimal Solution: $\max\{ p \mid A(n,p) \leq B \}$ where $B$ is the size of the knapsack.

For example, if $B=8$, OPT=14, if $B=7$, OPT=12, if $B=6$, OPT=11.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The input has 2n numbers, so the input total length is 2n.
Assume $P = \max \text{ profit}(a_i)$ for $i=1,2,...,n$ and $S = \max \text{ size}(a_i)$ for $i=1,2,...,n$.
So the input has total length $2n\log(\max(P,S))$.

For the dynamic programming algorithm,
there are $n$ rows and at most $nP$ columns.
Each entry can be computed in constant time (look up two entries).
So the total time complexity is $O(n^2P)$.

The running time is not polynomial if $P$ is very large compared to $n$.
In fact, the Knapsack problem is NP-complete and we do not expect
That there exists a polynomial time algorithm.
**Scaling Down**

**Idea:** use dynamic programming to scale down the numbers and compute the optimal solution in this modified instance

- Suppose $P \geq 1000n$, ($P$ max profit).
- Then $OPT \geq 1000n$ (each single object must enter the knapsack).
- Now scale down each element by 100 times ($profit^* := profit/100$).
- Compute the optimal solution using this new profit.
- Can’t distinguish between element of size, say 2199 and 2100.
- Each element contributes at most an error of 100.
- So total error is at most $100n$.
- This is at most $1/10$ of the optimal solution.
- However, the running time is 100 times faster.
Goal: to find a solution which is at least \((1-\epsilon)OPT\) for any \(\epsilon > 0\).

**Approximation Scheme for Knapsack**

1. Given \(\epsilon > 0\), let \(K = \epsilon P/n\), where \(P\) is the largest profit of an object.
2. For each object \(a_i\), define \(\text{profit}^*(a_i) = \left\lfloor \text{profit}(a_i)/K \right\rfloor\).
3. With these as profits of objects, using the dynamic programming algorithm, find the most profitable set, say \(S'\).
4. Output \(S'\) as the approximate solution.
Quality of Solution

**Theorem.** Let $S$ denote the set returned by the algorithm. Then,

$$\text{profit}(S) \geq (1 - \epsilon)\text{OPT}.$$ 

**Proof.** Let $O$ denote the optimal set.
For each object $a$, because of rounding down,

$$K \cdot \text{profit}^*(a)$$

can be smaller than $\text{profit}(a)$, but by not more than $K$.

Since there are at most $n$ objects in $O$,

$$\text{profit}(O) - K \cdot \text{profit}^*(O) \leq nK.$$ 

Since the algorithm return an optimal solution under the new profits,

$$\text{profit}(S) \geq K \cdot \text{profit}^*(S) \geq K \cdot \text{profit}^*(O) \geq \text{profit}(O) - nK = \text{OPT} - \epsilon P \geq (1 - \epsilon)\text{OPT}$$

because $\text{OPT} \geq P$. 
Running Time

For the dynamic programming algorithm, there are n rows and at most \( n \left\lfloor \frac{p}{k} \right\rfloor \) columns. Each entry can be computed in constant time (look up two entries). So the total time complexity is \( O(n^2 \left\lfloor \frac{p}{k} \right\rfloor) = O(n^3/ \epsilon) \).

Therefore, we have an approximation scheme for Knapsack.

Claim: Strong NP-hard problems do not admit FPAS assuming P\( \neq \)NP

Knapsack is not Strong NP-hard
Approximation Scheme

Quick Summary
1. Modify the instance by rounding the numbers.
2. Use dynamic programming to compute an optimal solution $S$ in the modified instance.
3. Output $S$ as the approximate solution.

Other examples: bin packing, Euclidean TSP.
Set covering

Given $n$ subsets $S_1, S_2, \ldots, S_m$ of $U$ with $U = 1, 2, \ldots, n$ (so $|U| = n$)

Find a minimum set $C$ of $\{1, 2, \ldots, m\}$
Such that $\bigcup_{i \in C} S_i = U$

NP-hard;

Goal: to pick a subset which cover all items
and **minimize the cardinality** of this subset.
Greedy for set covering

General greedy method:
Sol = emptyset
While not finished
    choose the set that covers most elements not yet covered

Example
U={1,2,3,4,5,6}
Sets: −S1={1,2} −S2={3,4} −S3={5,6} −S4={1,3,5}

Algorithm picks {4,1,2,3}
Not optimal!
Greedy for set covering

Notation: $C^{OPT} = \text{optimal cover let } k=|C^{OPT}|$

Fact: At any iteration of the algorithm, there exists $S_j$ which contains at least $\geq \frac{1}{k}$ fraction of yet-not-covered elements

Proof: by contradiction.
If all sets cover $<\frac{1}{k}$ fraction of yet-not-covered elements, there is no way to cover them using $k$ sets
But $C^{OPT}$ does that!
Therefore, at each iteration greedy covers $\geq\frac{1}{k}$ fraction of yet-not-covered elements
Greedy for set covering

Fact: At any iteration of the algorithm, there exists $S_j$ which contains at least $1/k$ fraction of yet-not-covered elements.

Let $C^i$ be the number of yet-not-covered elements at the end of step $i=0,1,2,...$

We have $C^{i+1} \leq C^i(1-1/k)$ with $C^0=n$

Therefore, after $t=k \ln n$ steps, we have

$C^t \leq C^0 (1-1/k)^t \leq n (1-1/k)^{k \ln n} < n 1/e^{\ln n} =1$

I.e., all elements are covered by the $k \ln n$ sets chosen by greedy algorithm.

Opt size is $k \Rightarrow$ greedy is $\ln(n)$-approximate.
The weighted set cover problem

- Assume set $S$ has cost $c(S)$.
- Find a min cost collection of sets that cover all elements $U=\{e_1, e_2, \ldots, e_n\}$
- $C_i$ be the set of elements not yet covered at the beginning of iteration $i$
- The greedy algorithm repeatedly selects the most cost-effective set, i.e. the set that maximizes $c(S_i)/(S_i \cap C_i)$
- Repeat till all elements are covered

- For an element $e_j$ covered by set $S_i$ define $price(e_j)= c(S_i)/(S_i \cap C_i)$

Claim: $C_{\text{ALG}} = \sum_j price(e_j)$
Analysis of Greedy Set Cover

• Claim: When set Si is selected there exists a set of OPT with cost effectiveness $\leq \frac{C^{OPT}}{C_{i}}$

• For ej covered by set Si it holds $\text{price}(e_j) \leq \frac{C^{OPT}}{C_{i}} \leq \frac{C^{OPT}}{n - j + 1}$

• We conclude $C^{ALG} = \sum_{j} \text{price}(e_j) \leq \sum_{n - j + 1}^{C^{OPT}} \leq H_n C^{OPT}$
Lower bound for Greedy

Optimal cost is $1 + \varepsilon$

Greedy cost is $H_n$
Better algorithm for Set covering?

It is possible to show that approximating Set covering better than 0.999.. ln(m) is NP-hard!!

APX-hard problem: a problem for which there is a constant c such that it is NP-hard to find an approximation algorithm with approximation ratio better than c

Equivalently: there exists constant c such that finding an approximation better than c is as hard as finding the optimal solution

Claim: APX hard problems do not admit PAS (Polynomial Approximation Schemes)
APX, PAS and FPAS

Diagram:
- APX
- PAS
- FPAS
- NP-hard
PAS and FPAS

Class of APX problems: problems that have a Polynomial Approximation Algorithm: for some constant $c$ running time polynomial in input length.

Class of PAS problems: problem that have a Polynomial Approximation Schemes: for any given $\epsilon$ running time polynomial in input length.

Class of FPAS problems: problems that have a Fully Polynomial Approximation Schemes: for any given $\epsilon$ running time polynomial in input length and $1/\epsilon$. 
There are problems that are hard to approximate:
For any constant: TSP in the general case, Set covering
For some constant $c$: Vertex cover, Max Sat (maximize no. of satisfied clauses)

APX hard problems do not admit PAS or FPAS
There are Strong NP-hard problems that admit a PAS
But finding a FPAS is as hard as finding the optimum:
Generalization of Knapsack with two constraints

Note: not all Strong NP-hard admit a good approximation