Coping with NP-Completeness

T. M. Murali

May 3, 5, 2021
Examples of Hard Computational Problems
(from Kevin Wayne’s slides at Princeton University)

- Aerospace engineering: optimal mesh partitioning for finite elements.
- Biology: protein folding.
- Chemical engineering: heat exchanger network synthesis.
- Civil engineering: equilibrium of urban traffic flow.
- Economics: computation of arbitrage in financial markets with friction.
- Electrical engineering: VLSI layout.
- Environmental engineering: optimal placement of contaminant sensors.
- Financial engineering: find minimum risk portfolio of given return.
- Game theory: find Nash equilibrium that maximizes social welfare.
- Genomics: phylogeny reconstruction.
- Mechanical engineering: structure of turbulence in sheared flows.
- Medicine: reconstructing 3-D shape from biplane angiocardiogram.
- Operations research: optimal resource allocation.
- Physics: partition function of 3-D Ising model in statistical mechanics.
- Politics: Shapley-Shubik voting power.
- Pop culture: Minesweeper consistency.
- Statistics: optimal experimental design.
How Do We Tackle an $NP$-Complete Problem?

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

(Garey and Johnson, *Computers and Intractability*)
How Do We Tackle an \(\mathcal{NP}\)-Complete Problem?

- These problems come up in real life.
How Do We Tackle an $\mathcal{NP}$-Complete Problem?

My Hobby:

Embedding NP-Complete Problems in Restaurant Orders

Chotchkie's Restaurant

<table>
<thead>
<tr>
<th>Appetizers</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Mixed Fruit</td>
<td>$2.15$</td>
</tr>
<tr>
<td>French Fries</td>
<td>$2.75$</td>
</tr>
<tr>
<td>Side Salad</td>
<td>$3.35$</td>
</tr>
<tr>
<td>Hot Wings</td>
<td>$3.55$</td>
</tr>
<tr>
<td>Mozzarella Sticks</td>
<td>$4.20$</td>
</tr>
<tr>
<td>Sampler Platter</td>
<td>$5.80$</td>
</tr>
</tbody>
</table>

Sandwiches

Barbecue $6.55$

We'd like exactly $15.05$ worth of appetizers, please.

... Exactly? Uh...

Here, these papers on the knapsack problem might help you out.

Listen, I have six other tables to get to—

As fast as possible, of course. Want something on traveling salesman?
How Do We Tackle an $\mathcal{NP}$-Complete Problem?

- These problems come up in real life.
- $\mathcal{NP}$-Complete means that a problem is hard to solve in the worst case. Can we come up with better solutions at least in some cases?
How Do We Tackle an $NP$-Complete Problem?

- These problems come up in real life.
- $NP$-Complete means that a problem is hard to solve in the worst case. Can we come up with better solutions at least in some cases?
How Do We Tackle an $\mathcal{NP}$-Complete Problem?

- These problems come up in real life.
- $\mathcal{NP}$-Complete means that a problem is hard to solve in the worst case. Can we come up with better solutions at least in some cases?
  - Develop algorithms that are exponential in one parameter in the problem.
  - Consider special cases of the input, e.g., graphs that “look like” trees.
  - Develop algorithms that can provably compute a solution close to the optimal.
Vertex Cover Problem

**Vertex cover**

**INSTANCE:** Undirected graph $G$ and an integer $k$

**QUESTION:** Does $G$ contain a vertex cover of size at most $k$?

- The problem has two parameters: $k$ and $n$, the number of nodes in $G$.
- What is the running time of a brute-force algorithm?
Vertex Cover Problem

INSTANCE: Undirected graph $G$ and an integer $k$

QUESTION: Does $G$ contain a vertex cover of size at most $k$?

- The problem has two parameters: $k$ and $n$, the number of nodes in $G$.
- What is the running time of a brute-force algorithm? $O(kn\binom{n}{k}) = O(kn^{k+1})$. 
Vertex Cover Problem

**INSTANCE:** Undirected graph $G$ and an integer $k$

**QUESTION:** Does $G$ contain a vertex cover of size at most $k$?

- The problem has two parameters: $k$ and $n$, the number of nodes in $G$.
- What is the running time of a brute-force algorithm? $O(kn\binom{n}{k}) = O(kn^{k+1})$.
- Can we devise an algorithm whose running time is exponential in $k$ but polynomial in $n$, e.g., $O(2^kn)$?
Designing the Vertex Cover Algorithm

- Intuition: if a graph has a small vertex cover, it cannot have too many edges.
Solving $\text{NP}$-Complete Problems

Small Vertex Covers

Trees

Approx. Vertex Cover

Load Balancing

Knapsack

Set Cover

---

**Designing the Vertex Cover Algorithm**

- **Intuition:** if a graph has a small vertex cover, it cannot have too many edges.
- **Claim:** If $G$ has $n$ nodes and $G$ has a vertex cover of size at most $k$, then $G$ has at most $kn$ edges.
Designing the Vertex Cover Algorithm

- Intuition: if a graph has a small vertex cover, it cannot have too many edges.
- Claim: If $G$ has $n$ nodes and $G$ has a vertex cover of size at most $k$, then $G$ has at most $kn$ edges.
- Easy part of algorithm: Return no if $G$ has more than $kn$ edges.
Designing the Vertex Cover Algorithm

- Intuition: if a graph has a small vertex cover, it cannot have too many edges.
- Claim: If $G$ has $n$ nodes and $G$ has a vertex cover of size at most $k$, then $G$ has at most $kn$ edges.
- Easy part of algorithm: Return 'no' if $G$ has more than $kn$ edges.
- $G - \{u\}$ is the graph $G$ without node $u$ and the edges incident on $u$.
Designing the Vertex Cover Algorithm

- Intuition: if a graph has a small vertex cover, it cannot have too many edges.
- Claim: If $G$ has $n$ nodes and $G$ has a vertex cover of size at most $k$, then $G$ has at most $kn$ edges.
- Easy part of algorithm: Return no if $G$ has more than $kn$ edges.
- $G - \{u\}$ is the graph $G$ without node $u$ and the edges incident on $u$.
- Consider an edge $(u, v)$. Either $u$ or $v$ must be in the vertex cover.
Designing the Vertex Cover Algorithm

- Intuition: if a graph has a small vertex cover, it cannot have too many edges.
- Claim: If $G$ has $n$ nodes and $G$ has a vertex cover of size at most $k$, then $G$ has at most $kn$ edges.
- Easy part of algorithm: Return $\text{no}$ if $G$ has more than $kn$ edges.
- $G - \{u\}$ is the graph $G$ without node $u$ and the edges incident on $u$.
- Consider an edge $(u, v)$. Either $u$ or $v$ must be in the vertex cover.
- Claim: $G$ has a vertex cover of size at most $k$ iff for any edge $(u, v)$ either $G - \{u\}$ or $G - \{v\}$ has a vertex cover of size at most $k - 1$. 

![Graph 1](attachment:image1.png) ![Graph 2](attachment:image2.png)
Vertex Cover Algorithm

To search for a $k$-node vertex cover in $G$:

If $G$ contains no edges, then the empty set is a vertex cover.

If $G$ contains $> k |V|$ edges, then it has no $k$-node vertex cover.

Else let $e = (u, v)$ be an edge of $G$.

Recursively check if either of $G \setminus \{u\}$ or $G \setminus \{v\}$ has a vertex cover of size $k-1$.

If neither of them does, then $G$ has no $k$-node vertex cover.

Else, one of them (say, $G \setminus \{u\}$) has a $(k-1)$-node vertex cover $T$.

In this case, $T \cup \{u\}$ is a $k$-node vertex cover of $G$.

Endif

Endif
Analysing the Vertex Cover Algorithm

- Develop a recurrence relation for the algorithm with parameters...
Analysing the Vertex Cover Algorithm

- Develop a recurrence relation for the algorithm with parameters $n$ and $k$.
- Let $T(n, k)$ denote the worst-case running time of the algorithm on an instance of Vertex Cover with parameters $n$ and $k$. 

We need $O(2^k n)$ time to count the number of edges.
Analysing the Vertex Cover Algorithm

- Develop a recurrence relation for the algorithm with parameters $n$ and $k$.
- Let $T(n, k)$ denote the worst-case running time of the algorithm on an instance of VERTEX COVER with parameters $n$ and $k$.
- $T(n, 1) \leq cn$. 
Analysing the Vertex Cover Algorithm

- Develop a recurrence relation for the algorithm with parameters $n$ and $k$.
- Let $T(n, k)$ denote the worst-case running time of the algorithm on an instance of VERTEX COVER with parameters $n$ and $k$.
- $T(n, 1) \leq cn$.
- $T(n, k) \leq 2T(n, k - 1) + ckn$.
  - We need $O(kn)$ time to count the number of edges.
Analysing the Vertex Cover Algorithm

- Develop a recurrence relation for the algorithm with parameters $n$ and $k$.
- Let $T(n, k)$ denote the worst-case running time of the algorithm on an instance of **Vertex Cover** with parameters $n$ and $k$.
- $T(n, 1) \leq cn$.
- $T(n, k) \leq 2T(n, k - 1) + ckn$.
  - We need $O(kn)$ time to count the number of edges.
- Claim: $T(n, k) = O(2^k kn)$.
Solving $\mathcal{NP}$-Hard Problems on Trees

- "$\mathcal{NP}$-Hard": at least as hard as $\mathcal{NP}$-Complete. We will use $\mathcal{NP}$-Hard to refer to optimisation versions of decision problems.
Solving \( \mathcal{NP} \)-Hard Problems on Trees

- "\( \mathcal{NP} \)-Hard": at least as hard as \( \mathcal{NP} \)-Complete. We will use \( \mathcal{NP} \)-Hard to refer to optimisation versions of decision problems.
- Many \( \mathcal{NP} \)-Hard problems can be solved efficiently on trees.
- Intuition: subtree rooted at any node \( v \) of the tree "interacts" with the rest of tree only through \( v \). Therefore, depending on whether we include \( v \) in the solution or not, we can decouple solving the problem in \( v \)'s subtree from the rest of the tree.
Designing Greedy Algorithm for Independent Set

- Optimisation problem: Find the largest independent set in a tree.
Designing Greedy Algorithm for Independent Set

- Optimisation problem: Find the largest independent set in a tree.
- Claim: Every tree $T(V, E)$ has a leaf, a node with degree 1.
- Claim: If a tree $T$ has a leaf $v$, then there exists a maximum-size independent set in $T$ that contains $v$. 

 CLAIM: If a tree $T$ has a leaf $v$, then there exists a maximum-size independent set in $T$ that contains $v$.

Prove by exchange argument.

1. Let $S$ be a maximum-size independent set that does not contain $v$.
2. Let $v$ be connected to $u$.
3. $u$ must be in $S$; otherwise, we can add $v$ to $S$, which means $S$ is not maximum size.
4. Since $u$ is in $S$, we can swap $u$ and $v$.

Claim: If a tree $T$ has a leaf $v$, then a maximum-size independent set in $T$ is $v$ and a maximum-size independent set in $T - \{v\}$.
Designing Greedy Algorithm for Independent Set

Optimisation problem: Find the largest independent set in a tree.

Claim: Every tree $T(V, E)$ has a leaf, a node with degree 1.

Claim: If a tree $T$ has a leaf $v$, then there exists a maximum-size independent set in $T$ that contains $v$. Prove by exchange argument.

- Let $S$ be a maximum-size independent set that does not contain $v$.
- Let $v$ be connected to $u$.
- $u$ must be in $S$; otherwise, we can add $v$ to $S$, which means $S$ is not maximum size.
- Since $u$ is in $S$, we can swap $u$ and $v$. 
Optimisation problem: Find the largest independent set in a tree.

Claim: Every tree $T(V, E)$ has a leaf, a node with degree 1.

Claim: If a tree $T$ has a leaf $v$, then there exists a maximum-size independent set in $T$ that contains $v$. Prove by exchange argument.

- Let $S$ be a maximum-size independent set that does not contain $v$.
- Let $v$ be connected to $u$.
- $u$ must be in $S$; otherwise, we can add $v$ to $S$, which means $S$ is not maximum size.
- Since $u$ is in $S$, we can swap $u$ and $v$.

Claim: If a tree $T$ has a a leaf $v$, then a maximum-size independent set in $T$ is $v$ and a maximum-size independent set in $T - \{v\}$.
Greedy Algorithm for Independent Set

- A \textit{forest} is a graph where every connected component is a tree.

---

To find a maximum-size independent set in a forest $F$:

Let $S$ be the independent set to be constructed (initially empty)

While $F$ has at least one edge

Let $e = (u, v)$ be an edge of $F$ such that $v$ is a leaf

Add $v$ to $S$

Delete from $F$ nodes $u$ and $v$, and all edges incident to them

Endwhile

Return $S$
Greedy Algorithm for Independent Set

- A *forest* is a graph where every connected component is a tree.
- Running time of the algorithm is $O(n)$.

---

To find a maximum-size independent set in a forest $F$:

1. Let $S$ be the independent set to be constructed (initially empty)
2. While $F$ has at least one edge
   - Let $e = (u, v)$ be an edge of $F$ such that $v$ is a leaf
   - Add $v$ to $S$
   - Delete from $F$ nodes $u$ and $v$, and all edges incident to them
3. Endwhile
4. Return $S$
Greedy Algorithm for Independent Set

- A *forest* is a graph where every connected component is a tree.
- Running time of the algorithm is $O(n)$.
- The algorithm works correctly on any graph for which we can repeatedly find a leaf.

---

To find a maximum-size independent set in a forest $F$:

1. Let $S$ be the independent set to be constructed (initially empty)
2. While $F$ has at least one edge
   - Let $e = (u, v)$ be an edge of $F$ such that $v$ is a leaf
   - Add $v$ to $S$
   - Delete from $F$ nodes $u$ and $v$, and all edges incident to them
3. Endwhile
4. Return $S$
Maximum Weight Independent Set

- Consider the **INDEPENDENT SET** problem but with a weight $w_v$ on every node $v$.
- Goal is to find an independent set $S$ such that $\sum_{v \in S} w_v$ is as large as possible.
Consider the **Independent Set** problem but with a weight $w_v$ on every node $v$.

Goal is to find an independent set $S$ such that $\sum_{v \in S} w_v$ is as large as possible.

Can we extend the greedy algorithm?
Consider the **INDEPENDENT SET** problem but with a weight $w_v$ on every node $v$.

Goal is to find an independent set $S$ such that $\sum_{v \in S} w_v$ is as large as possible.

Can we extend the greedy algorithm? Exchange argument fails: if $u$ is a parent of a leaf $v$, $w_u$ may be larger than $w_v$. 
Consider the **INDEPENDENT SET** problem but with a weight $w_v$ on every node $v$.

Goal is to find an independent set $S$ such that $\sum_{v \in S} w_v$ is as large as possible.

Can we extend the greedy algorithm? Exchange argument fails: if $u$ is a parent of a leaf $v$, $w_u$ may be larger than $w_v$.

But there are still only two possibilities: either include $u$ in the independent set or include all neighbours of $u$ that are leaves.
Consider the **Independent Set** problem but with a weight $w_v$ on every node $v$.

Goal is to find an independent set $S$ such that $\sum_{v \in S} w_v$ is as large as possible.

Can we extend the greedy algorithm? Exchange argument fails: if $u$ is a parent of a leaf $v$, $w_u$ may be larger than $w_v$.

But there are still only two possibilities: either include $u$ in the independent set or include all neighbours of $u$ that are leaves.

Suggests dynamic programming algorithm.
Designing Dynamic Programming Algorithm

- Dynamic programming algorithm needs a set of sub-problems, recursion to combine sub-problems, and order over sub-problems.

- What are the sub-problems?
Designing Dynamic Programming Algorithm

- Dynamic programming algorithm needs a set of sub-problems, recursion to combine sub-problems, and order over sub-problems.
- What are the sub-problems?
  - Pick a node $r$ and root tree at $r$: orient edges towards $r$.
  - parent $p(u)$ of a node $u$ is the node adjacent to $u$ along the path to $r$.
  - Sub-problems are $T_u$: subtree induced by $u$ and all its descendants.
Designing Dynamic Programming Algorithm

- Dynamic programming algorithm needs a set of sub-problems, recursion to combine sub-problems, and order over sub-problems.

- What are the sub-problems?
  - Pick a node $r$ and *root* tree at $r$: orient edges towards $r$.
  - *parent* $p(u)$ of a node $u$ is the node adjacent to $u$ along the path to $r$.
  - Sub-problems are $T_u$: subtree induced by $u$ and all its descendants.

- Ordering the sub-problems: start at leaves and work our way up to the root.
Recursion for Dynamic Programming Algorithm

Either we include $u$ in an optimal solution or exclude $u$.

- $OPT_{in}(u)$: maximum weight of an independent set in $T_u$ that includes $u$.
- $OPT_{out}(u)$: maximum weight of an independent set in $T_u$ that excludes $u$. 
Recursion for Dynamic Programming Algorithm

- Either we include $u$ in an optimal solution or exclude $u$.
  - $OPT_{in}(u)$: maximum weight of an independent set in $T_u$ that includes $u$.
  - $OPT_{out}(u)$: maximum weight of an independent set in $T_u$ that excludes $u$.
- Base cases:
Recursion for Dynamic Programming Algorithm

Either we include $u$ in an optimal solution or exclude $u$.

- $OPT_{in}(u)$: maximum weight of an independent set in $T_u$ that includes $u$.
- $OPT_{out}(u)$: maximum weight of an independent set in $T_u$ that excludes $u$.

Base cases: For a leaf $u$, $OPT_{in}(u) = w_u$ and $OPT_{out}(u) = 0$.

Recurrence: Include $u$ or exclude $u$. 

Either we include $u$ in an optimal solution or exclude $u$.

- $OPT_{in}(u)$: maximum weight of an independent set in $T_u$ that includes $u$.
- $OPT_{out}(u)$: maximum weight of an independent set in $T_u$ that excludes $u$.

Base cases: For a leaf $u$, $OPT_{in}(u) = w_u$ and $OPT_{out}(u) = 0$.

Recurrence: Include $u$ or exclude $u$. 

Either we include $u$ in an optimal solution or exclude $u$.

- $OPT_{in}(u)$: maximum weight of an independent set in $T_u$ that includes $u$.
- $OPT_{out}(u)$: maximum weight of an independent set in $T_u$ that excludes $u$.

Base cases: For a leaf $u$, $OPT_{in}(u) = w_u$ and $OPT_{out}(u) = 0$.

Recurrence: Include $u$ or exclude $u$. 

Either we include $u$ in an optimal solution or exclude $u$.

- $OPT_{in}(u)$: maximum weight of an independent set in $T_u$ that includes $u$.
- $OPT_{out}(u)$: maximum weight of an independent set in $T_u$ that excludes $u$.

Base cases: For a leaf $u$, $OPT_{in}(u) = w_u$ and $OPT_{out}(u) = 0$.

Recurrence: Include $u$ or exclude $u$. 

Either we include $u$ in an optimal solution or exclude $u$.

- $OPT_{in}(u)$: maximum weight of an independent set in $T_u$ that includes $u$.
- $OPT_{out}(u)$: maximum weight of an independent set in $T_u$ that excludes $u$.

Base cases: For a leaf $u$, $OPT_{in}(u) = w_u$ and $OPT_{out}(u) = 0$.

Recurrence: Include $u$ or exclude $u$. 

Either we include $u$ in an optimal solution or exclude $u$.

- $OPT_{in}(u)$: maximum weight of an independent set in $T_u$ that includes $u$.
- $OPT_{out}(u)$: maximum weight of an independent set in $T_u$ that excludes $u$.

Base cases: For a leaf $u$, $OPT_{in}(u) = w_u$ and $OPT_{out}(u) = 0$.

Recurrence: Include $u$ or exclude $u$. 

Either we include $u$ in an optimal solution or exclude $u$.

- $OPT_{in}(u)$: maximum weight of an independent set in $T_u$ that includes $u$.
- $OPT_{out}(u)$: maximum weight of an independent set in $T_u$ that excludes $u$.

Base cases: For a leaf $u$, $OPT_{in}(u) = w_u$ and $OPT_{out}(u) = 0$.

Recurrence: Include $u$ or exclude $u$. 

Either we include $u$ in an optimal solution or exclude $u$.

- $OPT_{in}(u)$: maximum weight of an independent set in $T_u$ that includes $u$.
- $OPT_{out}(u)$: maximum weight of an independent set in $T_u$ that excludes $u$.

Base cases: For a leaf $u$, $OPT_{in}(u) = w_u$ and $OPT_{out}(u) = 0$.

Recurrence: Include $u$ or exclude $u$. 

Either we include $u$ in an optimal solution or exclude $u$.

- $OPT_{in}(u)$: maximum weight of an independent set in $T_u$ that includes $u$.
- $OPT_{out}(u)$: maximum weight of an independent set in $T_u$ that excludes $u$.

Base cases: For a leaf $u$, $OPT_{in}(u) = w_u$ and $OPT_{out}(u) = 0$.

Recurrence: Include $u$ or exclude $u$. 

Either we include $u$ in an optimal solution or exclude $u$.

- $OPT_{in}(u)$: maximum weight of an independent set in $T_u$ that includes $u$.
- $OPT_{out}(u)$: maximum weight of an independent set in $T_u$ that excludes $u$.

Base cases: For a leaf $u$, $OPT_{in}(u) = w_u$ and $OPT_{out}(u) = 0$.

Recurrence: Include $u$ or exclude $u$. 

Either we include $u$ in an optimal solution or exclude $u$.

- $OPT_{in}(u)$: maximum weight of an independent set in $T_u$ that includes $u$.
- $OPT_{out}(u)$: maximum weight of an independent set in $T_u$ that excludes $u$.

Base cases: For a leaf $u$, $OPT_{in}(u) = w_u$ and $OPT_{out}(u) = 0$.

Recurrence: Include $u$ or exclude $u$. 

Either we include $u$ in an optimal solution or exclude $u$.

- $OPT_{in}(u)$: maximum weight of an independent set in $T_u$ that includes $u$.
- $OPT_{out}(u)$: maximum weight of an independent set in $T_u$ that excludes $u$.

Base cases: For a leaf $u$, $OPT_{in}(u) = w_u$ and $OPT_{out}(u) = 0$.

Recurrence: Include $u$ or exclude $u$. 

Either we include $u$ in an optimal solution or exclude $u$.

- $OPT_{in}(u)$: maximum weight of an independent set in $T_u$ that includes $u$.
- $OPT_{out}(u)$: maximum weight of an independent set in $T_u$ that excludes $u$.

Base cases: For a leaf $u$, $OPT_{in}(u) = w_u$ and $OPT_{out}(u) = 0$.

Recurrence: Include $u$ or exclude $u$. 

Either we include $u$ in an optimal solution or exclude $u$.

- $OPT_{in}(u)$: maximum weight of an independent set in $T_u$ that includes $u$.
- $OPT_{out}(u)$: maximum weight of an independent set in $T_u$ that excludes $u$.

Base cases: For a leaf $u$, $OPT_{in}(u) = w_u$ and $OPT_{out}(u) = 0$.

Recurrence: Include $u$ or exclude $u$. 

Either we include $u$ in an optimal solution or exclude $u$.

- $OPT_{in}(u)$: maximum weight of an independent set in $T_u$ that includes $u$.
- $OPT_{out}(u)$: maximum weight of an independent set in $T_u$ that excludes $u$.

Base cases: For a leaf $u$, $OPT_{in}(u) = w_u$ and $OPT_{out}(u) = 0$.

Recurrence: Include $u$ or exclude $u$. 

Either we include $u$ in an optimal solution or exclude $u$.

- $OPT_{in}(u)$: maximum weight of an independent set in $T_u$ that includes $u$.
- $OPT_{out}(u)$: maximum weight of an independent set in $T_u$ that excludes $u$.

Base cases: For a leaf $u$, $OPT_{in}(u) = w_u$ and $OPT_{out}(u) = 0$.

Recurrence: Include $u$ or exclude $u$. 

Either we include $u$ in an optimal solution or exclude $u$.

- $OPT_{in}(u)$: maximum weight of an independent set in $T_u$ that includes $u$.
- $OPT_{out}(u)$: maximum weight of an independent set in $T_u$ that excludes $u$.

Base cases: For a leaf $u$, $OPT_{in}(u) = w_u$ and $OPT_{out}(u) = 0$.

Recurrence: Include $u$ or exclude $u$. 

Either we include $u$ in an optimal solution or exclude $u$.

- $OPT_{in}(u)$: maximum weight of an independent set in $T_u$ that includes $u$.
- $OPT_{out}(u)$: maximum weight of an independent set in $T_u$ that excludes $u$.

Base cases: For a leaf $u$, $OPT_{in}(u) = w_u$ and $OPT_{out}(u) = 0$.

Recurrence: Include $u$ or exclude $u$. 

Either we include $u$ in an optimal solution or exclude $u$.

- $OPT_{in}(u)$: maximum weight of an independent set in $T_u$ that includes $u$.
- $OPT_{out}(u)$: maximum weight of an independent set in $T_u$ that excludes $u$.

Base cases: For a leaf $u$, $OPT_{in}(u) = w_u$ and $OPT_{out}(u) = 0$.

Recurrence: Include $u$ or exclude $u$.
Recursion for Dynamic Programming Algorithm

Either we include \( u \) in an optimal solution or exclude \( u \).

- \( OPT_{in}(u) \): maximum weight of an independent set in \( T_u \) that includes \( u \).
- \( OPT_{out}(u) \): maximum weight of an independent set in \( T_u \) that excludes \( u \).

Base cases: For a leaf \( u \), \( OPT_{in}(u) = w_u \) and \( OPT_{out}(u) = 0 \).

Recurrence: Include \( u \) or exclude \( u \).

- If we include \( u \), all children must be excluded.
  \[ OPT_{in}(u) = w_u + \sum_{v \in \text{children}(u)} OPT_{out}(v) \]
Either we include \( u \) in an optimal solution or exclude \( u \).

- \( \text{OPT}_{\text{in}}(u) \): maximum weight of an independent set in \( T_u \) that includes \( u \).
- \( \text{OPT}_{\text{out}}(u) \): maximum weight of an independent set in \( T_u \) that excludes \( u \).

Base cases: For a leaf \( u \), \( \text{OPT}_{\text{in}}(u) = w_u \) and \( \text{OPT}_{\text{out}}(u) = 0 \).

Recurrence: Include \( u \) or exclude \( u \).

1. If we include \( u \), all children must be excluded.
   \[
   \text{OPT}_{\text{in}}(u) = w_u + \sum_{v \in \text{children}(u)} \text{OPT}_{\text{out}}(v)
   \]

2. If we exclude \( u \), a child may or may not be excluded.
   \[
   \text{OPT}_{\text{out}}(u) = \sum_{v \in \text{children}(u)} \max (\text{OPT}_{\text{in}}(v), \text{OPT}_{\text{out}}(v))
   \]
Dynamic Programming Algorithm

To find a maximum-weight independent set of a tree $T$:

Root the tree at a node $r$

For all nodes $u$ of $T$ in post-order

If $u$ is a leaf then set the values:

$M_{out}[u] = 0$
$M_{in}[u] = w_u$

Else set the values:

$M_{out}[u] = \sum_{v \in \text{children}(u)} \max(M_{out}[v], M_{in}[v])$

$M_{in}[u] = w_u + \sum_{v \in \text{children}(u)} M_{out}[u].$

Endif
Endfor

Return $\max(M_{out}[r], M_{in}[r])$
Dynamic Programming Algorithm

To find a maximum-weight independent set of a tree $T$:

Root the tree at a node $r$

For all nodes $u$ of $T$ in post-order

If $u$ is a leaf then set the values:

\[ M_{out}[u] = 0 \]
\[ M_{in}[u] = w_u \]

Else set the values:

\[ M_{out}[u] = \sum_{v \in \text{children}(u)} \max(M_{out}[v], M_{in}[v]) \]
\[ M_{in}[u] = w_u + \sum_{v \in \text{children}(u)} M_{out}[u] \]

Endif
Endfor

Return $\max(M_{out}[r], M_{in}[r])$

- Running time of the algorithm is $O(n)$. 

T. M. Murali  May 3, 5, 2021  Coping with NP-Completeness
Approximation Algorithms

- Methods for optimisation versions of \( \mathcal{NP} \)-Complete problems.
- Run in polynomial time.
- Solution returned is guaranteed to be within a small factor of the optimal solution
Approximation Algorithm for VertexCover

EASYVERTEXCOVER(G) (Gavril, 1974; Yannakakis)

1: \( C \leftarrow \emptyset \) \{C will be the vertex cover\}
2: while G has at least one edge do
3: \( (u, v) \) be any edge in G
4: \hspace{2cm} \{Update C using u and/or v\}
5: \hspace{2cm} \{Update G using u and/or v\}
6: 
7: end while
8: return \( C \)

```
T. M. Murali May 3, 5, 2021 Coping with NP-Completeness
```
**Approximation Algorithm for VertexCover**

**EasyVertexCover(G)** (Gavril, 1974; Yannakakis)

1. $C \leftarrow \emptyset$ \{C will be the vertex cover\}
2. while $G$ has at least one edge do
3. Let $(u, v)$ be any edge in $G$
4. Add $u$ and $v$ to $C$
5. \{Update $G$ using $u$ and/or $v$\}
6. 
7. end while
8. return $C$
Approximation Algorithm for VertexCover

**EasyVertexCover**\((G)\) *(Gavril, 1974; Yannakakis)*

1: \(C \leftarrow \emptyset, \ E' \leftarrow \emptyset\) \{\(C\) will be the vertex cover\}
2: while \(G\) has at least one edge do
3: \(\text{Let } (u, v)\) be any edge in \(G\)
4: \(\text{Add } u \text{ and } v \text{ to } C\)
5: \(G \leftarrow G - \{u, v\}\) \{Delete \(u, v\), and all incident edges from \(G\).\}
6: \(\text{Add } (u, v) \text{ to } E'\) \{Keep track of edges for bookkeeping.\}
7: end while
8: return \(C\)
Approximation Algorithm for VertexCover

**EasyVertexCover(G)** (Gavril, 1974; Yannakakis)

1: $C \leftarrow \emptyset$, $E' \leftarrow \emptyset$ \{C will be the vertex cover\}
2: while $G$ has at least one edge do
3: Let $(u, v)$ be any edge in $G$
4: Add $u$ and $v$ to $C$
5: $G \leftarrow G - \{u, v\}$ \{Delete $u$, $v$, and all incident edges from $G$.\}
6: Add $(u, v)$ to $E'$ \{Keep track of edges for bookkeeping.\}
7: end while
8: return $C$
Approximation Algorithm for VertexCover

\textsc{EasyVertexCover}(G) (Gavril, 1974; Yannakakis)

1: \( C \leftarrow \emptyset, E' \leftarrow \emptyset \) \{\( C \) will be the vertex cover\}
2: \textbf{while} \( G \) has at least one edge \textbf{do}
3: \quad Let \((u, v)\) be any edge in \( G \)
4: \quad Add \( u \) and \( v \) to \( C \)
5: \quad \( G \leftarrow G - \{u, v\} \) \{Delete \( u, v, \) and all incident edges from \( G.\)\}
6: \quad Add \((u, v)\) to \( E' \) \{Keep track of edges for bookkeeping.\}
7: \textbf{end while}
8: \textbf{return} \( C \)
Approximation Algorithm for VertexCover

EASYVERTEXCOVER(G) (Gavril, 1974; Yannakakis)

1: $C \leftarrow \emptyset$, $E' \leftarrow \emptyset$ \{C will be the vertex cover\}
2: while $G$ has at least one edge do
3: Let $(u, v)$ be any edge in $G$
4: Add $u$ and $v$ to $C$
5: $G \leftarrow G - \{u, v\}$ \{Delete $u$, $v$, and all incident edges from $G$.\}
6: Add $(u, v)$ to $E'$ \{Keep track of edges for bookkeeping.\}
7: end while
8: return $C$
Analysis of EasyVertexCover

**EasyVertexCover(G)**

1. \( C \leftarrow \emptyset, E' \leftarrow \emptyset \)
2. while \( G \) has at least one edge do
3. \( \text{Let (} u, v \text{) be any edge in } G \)
4. \( \text{Add } u \text{ and } v \text{ to } C \)
5. \( G \leftarrow G - \{u, v\} \)
6. \( \text{Add (} u, v \text{) to } E' \)
7. end while
8. return \( C \)

- Running time is

Claim: \( C \) is a vertex cover.

Claim: No two edges in \( E' \) can be covered by the same node.

Claim: The size \( c^* \) of the smallest vertex cover is at least \( |E'| \).

Claim: \( |C| = 2|E'| \leq 2c^* \)

No approximation algorithm with a factor better than 1.3606 is possible unless \( P = NP \) (Dinur and Safra, 2005).

No approximation algorithm with a factor better than 2 is possible if the "unique games conjecture" is true (Khot and Regev, 2008).
Analysis of EasyVertexCover

**EasyVertexCover**$(G)$

1: $C \leftarrow \emptyset$, $E' \leftarrow \emptyset$
2: **while** $G$ has at least one edge **do**
3: Let $(u, v)$ be any edge in $G$
4: Add $u$ and $v$ to $C$
5: $G \leftarrow G - \{u, v\}$
6: Add $(u, v)$ to $E'$
7: **end while**
8: **return** $C$

- Running time is linear in the size of the graph.
Analysis of EasyVertexCover

**EasyVertexCover**(*G*)

1: \( C \leftarrow \emptyset, \quad E' \leftarrow \emptyset \)
2: while \( G \) has at least one edge do
3: \( \text{Let } (u, v) \text{ be any edge in } G \)
4: \( \text{Add } u \text{ and } v \text{ to } C \)
5: \( G \leftarrow G - \{u, v\} \)
6: \( \text{Add } (u, v) \text{ to } E' \)
7: end while
8: return \( C \)

- Running time is linear in the size of the graph.
- Claim: \( C \) is a vertex cover.
Analysis of EasyVertexCover

**EasyVertexCover**\((G)\)

1: \(C \leftarrow \emptyset, E' \leftarrow \emptyset\)
2: **while** \(G\) has at least one edge **do**
3: \(\text{Let } (u, v) \text{ be any edge in } G\)
4: \(\text{Add } u \text{ and } v \text{ to } C\)
5: \(G \leftarrow G - \{u, v\}\)
6: \(\text{Add } (u, v) \text{ to } E'\)
7: **end while**
8: **return** \(C\)

- Running time is linear in the size of the graph.
- Claim: \(C\) is a vertex cover.
- Claim: No two edges in \(E'\) can be covered by the same node.
**EasyVertexCover**

1. Let \( C \leftarrow \emptyset \), \( E' \leftarrow \emptyset \)
2. while \( G \) has at least one edge do
   3. Let \((u, v)\) be any edge in \( G \)
   4. Add \( u \) and \( v \) to \( C \)
   5. \( G \leftarrow G - \{u, v\} \)
   6. Add \((u, v)\) to \( E' \)
3. end while
4. return \( C \)

- Running time is linear in the size of the graph.
- Claim: \( C \) is a vertex cover.
- Claim: No two edges in \( E' \) can be covered by the same node.
- Claim: The size \( c^* \) of the smallest vertex cover is at least \(|E'|\).
### Analysis of EasyVertexCover

**EasyVertexCover(G)**

1: \( C \leftarrow \emptyset, \ E' \leftarrow \emptyset \)
2: **while** \( G \) has at least one edge **do**
3: \( \) Let \( (u, v) \) be any edge in \( G \)
4: \( \) Add \( u \) and \( v \) to \( C \)
5: \( G \leftarrow G - \{u, v\} \)
6: \( \) Add \( (u, v) \) to \( E' \)
7: **end while**
8: **return** \( C \)

- Running time is linear in the size of the graph.
- **Claim:** \( C \) is a vertex cover.
- **Claim:** No two edges in \( E' \) can be covered by the same node.
- **Claim:** The size \( c^* \) of the smallest vertex cover is at least \( |E'| \).
- **Claim:** \( |C| = 2|E'| \leq 2c^* \)
Analysis of EasyVertexCover

EasyVertexCover(G)

1: \( C \leftarrow \emptyset, E' \leftarrow \emptyset \)
2: while \( G \) has at least one edge do
3: Let \((u, v)\) be any edge in \( G \)
4: Add \( u \) and \( v \) to \( C \)
5: \( G \leftarrow G - \{u, v\} \)
6: Add \((u, v)\) to \( E'\)
7: end while
8: return \( C \)

- Running time is linear in the size of the graph.
- Claim: \( C \) is a vertex cover.
- Claim: No two edges in \( E' \) can be covered by the same node.
- Claim: The size \( c^* \) of the smallest vertex cover is at least \(|E'|\).
- Claim: \(|C| = 2|E'| \leq 2c^*\)
- No approximation algorithm with a factor better than 1.3606 is possible unless \( P = NP \) (Dinur and Safra, 2005).
Analysis of EasyVertexCover

**EasyVertexCover(G)**

1: \( C \leftarrow \emptyset, \ E' \leftarrow \emptyset \)
2: \textbf{while} \( G \) has at least one edge \textbf{do}
3: \hspace{1em} Let \((u, v)\) be any edge in \( G \)
4: \hspace{1em} Add \( u \) and \( v \) to \( C \)
5: \hspace{1em} \( G \leftarrow G - \{u, v\} \)
6: \hspace{1em} Add \((u, v)\) to \( E' \)
7: \textbf{end while}
8: \textbf{return} \( C \)

- Running time is linear in the size of the graph.
- Claim: \( C \) is a vertex cover.
- Claim: No two edges in \( E' \) can be covered by the same node.
- Claim: The size \( c^* \) of the smallest vertex cover is at least \(|E'|\).
- Claim: \(|C| = 2|E'| \leq 2c^* \)
- No approximation algorithm with a factor better than 1.3606 is possible unless \( P = NP \) (Dinur and Safra, 2005).
- No approximation algorithm with a factor better than 2 is possible if the “unique games conjecture” is true (Khot and Regev, 2008).
Given set of $m$ machines $M_1, M_2, \ldots M_m$.

Given a set of $n$ jobs: job $j$ has processing time $t_j$.

Assign each job to one machine so that the total time spent is minimised.
Load Balancing Problem

- Given set of \( m \) machines \( M_1, M_2, \ldots M_m \).
- Given a set of \( n \) jobs: job \( j \) has processing time \( t_j \).
- Assign each job to one machine so that the total time spent is minimised.
- Let \( A(i) \) be the set of jobs assigned to machine \( M_i \).
- Total time spent on machine \( i \) is \( T_i = \sum_{k \in A(i)} t_k \).
- Minimise *makespan* \( T = \max_i T_i \), the largest load on any machine.
Given set of $m$ machines $M_1, M_2, \ldots M_m$.

Given a set of $n$ jobs: job $j$ has processing time $t_j$.

Assign each job to one machine so that the total time spent is minimised.

Let $A(i)$ be the set of jobs assigned to machine $M_i$.

Total time spent on machine $i$ is $T_i = \sum_{k \in A(i)} t_k$.

Minimise makespan $T = \max_i T_i$, the largest load on any machine.

Minimising makespan is $\mathcal{NP}$-Complete.
Greedy-Balance Algorithm

- Adopt a greedy approach *(Graham, 1966)*.
- Process jobs in *any* order.
- Assign next job to the processor that has smallest total load so far.

**Greedy-Balance:**

Start with no jobs assigned

Set $T_i = 0$ and $A(i) = \emptyset$ for all machines $M_i$

For $j = 1, \ldots, n$

   Let $M_i$ be a machine that achieves the minimum $\min_k T_k$

   Assign job $j$ to machine $M_i$

   Set $A(i) \leftarrow A(i) \cup \{j\}$

   Set $T_i \leftarrow T_i + t_j$

EndFor
Example of Greedy-Balance Algorithm

Jobs

3

2

Job index

Job time

Machines

\( T = T_2 \)

\( T_1, T_3 \)

Jobs

1 2 3 4 5 6 7 8 9 10

Machines

1 2 3 4 5 6 7 8 9 10

M_1 M_2 M_3

T. M. Murali May 3, 5, 2021 Coping with NP-Completeness
Lower Bounds on the Optimal Makespan

- We need a lower bound on the optimum makespan $T^*$. 
Lower Bounds on the Optimal Makespan

- We need a lower bound on the optimum makespan $T^*$. 
- The two bounds below will suffice:

\[ T^* \geq \frac{1}{m} \sum_{j} t_j \]
\[ T^* \geq \max_{j} t_j \]
Claim: Computed makespan $T \leq 2T^*$. 
Claim: Computed makespan $T \leq 2T^*$.  
Let $M_i$ be the machine whose load is $T$  
and $j$ be the last job placed on $M_i$.  
What was the situation just before placing this job?
Claim: Computed makespan $T \leq 2T^*$. 
Let $M_i$ be the machine whose load is $T$ and $j$ be the last job placed on $M_i$. 
What was the situation just before placing this job? 
$M_i$ had the smallest load and its load was $T - t_j$. 
For every machine $M_k$, load $T_k \geq T - t_j$. 
Claim: Computed makespan $T \leq 2T^*$.

Let $M_i$ be the machine whose load is $T$ and $j$ be the last job placed on $M_i$.

What was the situation just before placing this job?

$M_i$ had the smallest load and its load was $T - t_j$.

For every machine $M_k$, load $T_k \geq T - t_j$.

\[ \sum_k T_k \geq m(T - t_j), \text{ where } k \text{ ranges over all machines} \]

\[ \sum_j t_j \geq m(T - t_j), \text{ where } j \text{ ranges over all jobs} \]

\[ T - t_j \leq \frac{1}{m} \sum_j t_j \leq T^* \]

\[ T \leq 2T^*, \text{ since } t_j \leq T^* \]
It is easy to construct an example for which the greedy algorithm produces a solution close to a factor of 2 away from optimal.
Improving the Bound

- It is easy to construct an example for which the greedy algorithm produces a solution close to a factor of 2 away from optimal.
- How can we improve the algorithm?
Improving the Bound

- It is easy to construct an example for which the greedy algorithm produces a solution close to a factor of 2 away from optimal.
- How can we improve the algorithm?
- What if we process the jobs in decreasing order of processing time? (Graham, 1969)
Sorted-Balance Algorithm

Sorted-Balance:
Start with no jobs assigned
Set $T_i = 0$ and $A(i) = \emptyset$ for all machines $M_i$
Sort jobs in decreasing order of processing times $t_j$
Assume that $t_1 \geq t_2 \geq \ldots \geq t_n$
For $j = 1, \ldots, n$
    Let $M_i$ be the machine that achieves the minimum $\min_k T_k$
    Assign job $j$ to machine $M_i$
    Set $A(i) \leftarrow A(i) \cup \{j\}$
    Set $T_i \leftarrow T_i + t_j$
EndFor
Sorted-Balance Algorithm

**Sorted-Balance:**

Start with no jobs assigned

Set $T_i = 0$ and $A(i) = \emptyset$ for all machines $M_i$

Sort jobs in decreasing order of processing times $t_j$

Assume that $t_1 \geq t_2 \geq \ldots \geq t_n$

For $j = 1, \ldots, n$

- Let $M_i$ be the machine that achieves the minimum $\min_k T_k$
- Assign job $j$ to machine $M_i$
- Set $A(i) \leftarrow A(i) \cup \{j\}$
- Set $T_i \leftarrow T_i + t_j$

EndFor

- This algorithm assigns the first $m$ jobs to $m$ distinct machines.
Example of Sorted-Balance Algorithm

Job time

Jobs

Job index

Machines

$T = T_1$

$T_2, T_3$

$M_1$

$M_2$

$M_3$
Analyzing Sorted-Balance

- Claim: if there are fewer than $m$ jobs, algorithm is optimal.
- Claim: if there are more than $m$ jobs, then $T^* \geq 2t_{m+1}$.
Analyzing Sorted-Balance

- Claim: if there are fewer than \( m \) jobs, algorithm is optimal.
- Claim: if there are more than \( m \) jobs, then \( T^* \geq 2t_{m+1} \).
  - Consider only the first \( m + 1 \) jobs in sorted order.
  - Consider \textit{any} assignment of these \( m + 1 \) jobs to machines.
  - Some machine must be assigned two jobs, each with processing time \( \geq t_{m+1} \).
  - This machine will have load at least \( 2t_{m+1} \).
Analyzing Sorted-Balance

- Claim: if there are fewer than \( m \) jobs, algorithm is optimal.
- Claim: if there are more than \( m \) jobs, then \( T^* \geq 2t_{m+1} \).
  - Consider only the first \( m + 1 \) jobs in sorted order.
  - Consider any assignment of these \( m + 1 \) jobs to machines.
  - Some machine must be assigned two jobs, each with processing time \( \geq t_{m+1} \).
  - This machine will have load at least \( 2t_{m+1} \).
- Claim: \( T \leq 3T^*/2 \).
Analyzing Sorted-Balance

- Claim: if there are fewer than \( m \) jobs, algorithm is optimal.
- Claim: if there are more than \( m \) jobs, then \( T^* \geq 2t_{m+1} \).
  - Consider only the first \( m + 1 \) jobs in sorted order.
  - Consider any assignment of these \( m + 1 \) jobs to machines.
  - Some machine must be assigned two jobs, each with processing time \( \geq t_{m+1} \).
  - This machine will have load at least \( 2t_{m+1} \).

- Claim: \( T \leq 3T^*/2 \).
- Let \( M_i \) be the machine whose load is \( T \) and \( j \) be the last job placed on \( M_i \). (\( M_i \) has at least two jobs.)
Analyzing Sorted-Balance

- Claim: if there are fewer than $m$ jobs, algorithm is optimal.
- Claim: if there are more than $m$ jobs, then $T^* \geq 2t_{m+1}$.
  - Consider only the first $m + 1$ jobs in sorted order.
  - Consider any assignment of these $m + 1$ jobs to machines.
  - Some machine must be assigned two jobs, each with processing time $\geq t_{m+1}$.
  - This machine will have load at least $2t_{m+1}$.

- Claim: $T \leq 3T^*/2$.
- Let $M_i$ be the machine whose load is $T$ and $j$ be the last job placed on $M_i$. ($M_i$ has at least two jobs.)

$$t_j \leq t_{m+1} \leq T^*/2, \text{ since } j \geq m + 1$$

$$T - t_j \leq T^*, \text{ Greedy-Balance proof}$$

$$T \leq 3T^*/2$$
Analyzing Sorted-Balance

- Claim: if there are fewer than $m$ jobs, algorithm is optimal.
- Claim: if there are more than $m$ jobs, then $T^* \geq 2t_{m+1}$.
  - Consider only the first $m + 1$ jobs in sorted order.
  - Consider any assignment of these $m + 1$ jobs to machines.
  - Some machine must be assigned two jobs, each with processing time $\geq t_{m+1}$.
  - This machine will have load at least $2t_{m+1}$.
- Claim: $T \leq 3T^*/2$.
- Let $M_i$ be the machine whose load is $T$ and $j$ be the last job placed on $M_i$. ($M_i$ has at least two jobs.)

$t_j \leq t_{m+1} \leq T^*/2$, since $j \geq m + 1$

$T - t_j \leq T^*$, Greedy-Balance proof

$T \leq 3T^*/2$

- Better bound: $T \leq 4T^*/3$ (Graham, 1969).
Analyzing Sorted-Balance

- Claim: if there are fewer than $m$ jobs, algorithm is optimal.
- Claim: if there are more than $m$ jobs, then $T^* \geq 2t_{m+1}$.
  - Consider only the first $m + 1$ jobs in sorted order.
  - Consider any assignment of these $m + 1$ jobs to machines.
  - Some machine must be assigned two jobs, each with processing time $\geq t_{m+1}$.
  - This machine will have load at least $2t_{m+1}$.
- Claim: $T \leq 3T^*/2$.
- Let $M_i$ be the machine whose load is $T$ and $j$ be the last job placed on $M_i$. ($M_i$ has at least two jobs.)

$$t_j \leq t_{m+1} \leq T^*/2,$$ since $j \geq m + 1$

$$T - t_j \leq T^*,$$ Greedy-Balance proof

$$T \leq 3T^*/2$$

Better bound: $T \leq 4T^*/3$ (Graham, 1969).

**Polynomial-time approximation scheme:** for every $\varepsilon > 0$, compute solution with makespan $T \leq (1 + \varepsilon)T^*$ in $O((n/\varepsilon)^{(1/\varepsilon^2)})$ time (Hochbaum and Shmoys, 1987).
The Knapsack Problem

Partition

**INSTANCE:** A set of $n$ natural numbers $w_1, w_2, \ldots, w_n$.

**SOLUTION:** A subset $S$ of numbers such that $\sum_{i \in S} w_i = \sum_{i \notin S} w_i$. 
The Knapsack Problem

**Partition**

**INSTANCE:** A set of $n$ natural numbers $w_1, w_2, \ldots, w_n$.

**SOLUTION:** A subset $S$ of numbers such that $\sum_{i \in S} w_i = \sum_{i \notin S} w_i$.

**Subset Sum**

**INSTANCE:** A set of $n$ natural numbers $w_1, w_2, \ldots, w_n$ and a target $W$.

**SOLUTION:** A subset $S$ of numbers such that $\sum_{i \in S} w_i$ is maximised subject to the constraint $\sum_{i \in S} w_i \leq W$. 
The Knapsack Problem

Partition

INSTANCE: A set of $n$ natural numbers $w_1, w_2, \ldots, w_n$.

SOLUTION: A subset $S$ of numbers such that $\sum_{i \in S} w_i = \sum_{i \notin S} w_i$.

Subset Sum

INSTANCE: A set of $n$ natural numbers $w_1, w_2, \ldots, w_n$ and a target $W$.

SOLUTION: A subset $S$ of numbers such that $\sum_{i \in S} w_i$ is maximised subject to the constraint $\sum_{i \in S} w_i \leq W$.

Knapsack

INSTANCE: A set of $n$ elements, with each element $i$ having a weight $w_i$ and a value $v_i$, and a knapsack capacity $W$.

SOLUTION: A subset $S$ of items such that $\sum_{i \in S} v_i$ is maximised subject to the constraint $\sum_{i \in S} w_i \leq W$.
The Knapsack Problem

Partition

INSTANCE: A set of \( n \) natural numbers \( w_1, w_2, \ldots, w_n \).

SOLUTION: A subset \( S \) of numbers such that \( \sum_{i \in S} w_i = \sum_{i \notin S} w_i \).

Subset Sum

INSTANCE: A set of \( n \) natural numbers \( w_1, w_2, \ldots, w_n \) and a target \( W \).

SOLUTION: A subset \( S \) of numbers such that \( \sum_{i \in S} w_i \) is maximised subject to the constraint \( \sum_{i \in S} w_i \leq W \).

Knapsack

INSTANCE: A set of \( n \) elements, with each element \( i \) having a weight \( w_i \) and a value \( v_i \), and a knapsack capacity \( W \).

SOLUTION: A subset \( S \) of items such that \( \sum_{i \in S} v_i \) is maximised subject to the constraint \( \sum_{i \in S} w_i \leq W \).

- 3D Matching \( \leq_P \) Partition \( \leq_P \) Subset Sum \( \leq_P \) Knapsack
- All problems have dynamic programming algorithms with pseudo-polynomial running times.
Dynamic Programming for Subset Sum

Subset Sum

INSTANCE: A set of $n$ natural numbers $w_1, w_2, \ldots, w_n$ and a target $W$.

SOLUTION: A subset $S$ of numbers such that $\sum_{i \in S} w_i$ is maximised subject to the constraint $\sum_{i \in S} w_i \leq W$. 

Worked it out on board in class a few weeks ago. Running time is $O(nW)$. 

$OPT(i, w)$ is the largest sum possible using only the first $i$ numbers with target $w$. 

$OPT(i, w) = OPT(i-1, w)$, $i > 0$, $w_i > w$

$OPT(i, w) = \max\left( OPT(i-1, w), w_i + OPT(i-1, w-w_i) \right)$, $i > 0$, $w_i \leq w$

$OPT(0, w) = 0$
Dynamic Programming for Subset Sum

**Subset Sum**

**INSTANCE:** A set of $n$ natural numbers $w_1, w_2, \ldots, w_n$ and a target $W$.

**SOLUTION:** A subset $S$ of numbers such that $\sum_{i \in S} w_i$ is maximised subject to the constraint $\sum_{i \in S} w_i \leq W$.

- Worked it out on board in class a few weeks ago. Running time is $O(nW)$.
- $OPT(i, w)$ is the largest sum possible using only the first $i$ numbers with target $w$. 
Dynamic Programming for Subset Sum

**Subset Sum**

**INSTANCE:** A set of \( n \) natural numbers \( w_1, w_2, \ldots, w_n \) and a target \( W \).

**SOLUTION:** A subset \( S \) of numbers such that \( \sum_{i \in S} w_i \) is maximised subject to the constraint \( \sum_{i \in S} w_i \leq W \).

- Worked it out on board in class a few weeks ago. Running time is \( O(nW) \).
- \( OPT(i, w) \) is the largest sum possible using only the first \( i \) numbers with target \( w \).

\[
OPT(i, w) = OPT(i - 1, w), \quad i > 0, w_i > w
\]
\[
OPT(i, w) = \max \left( OPT(i - 1, w), w_i + OPT(i - 1, w - w_i) \right), \quad i > 0, w_i \leq w
\]
\[
OPT(0, w) = 0
\]
Dynamic Programming for Knapsack

**KNAPSACK**

**INSTANCE:** A set of $n$ elements, with each element $i$ having a weight $w_i$ and a value $v_i$, and a knapsack capacity $W$.

**SOLUTION:** A subset $S$ of items such that $\sum_{i \in S} v_i$ is maximised subject to the constraint $\sum_{i \in S} w_i \leq W$. 

Can generalize the dynamic program for Subset Sum. But we will develop a different dynamic program that will be useful later.

$OPT(i, v)$ is the smallest knapsack weight so that there is a solution using only the first $i$ items with total value $\geq v$.

What are the ranges of $i$ and $v$?

▶ $i$ ranges between 0 and $n$, the number of items.
▶ Given $i$, $v$ ranges between 0 and $\sum_{1 \leq j \leq i} v_j$.
▶ Largest value of $v$ is $\sum_{1 \leq j \leq n} v_j \leq n v^*$, where $v^* = \max_i v_i$.

The solution we want is the largest value $v$ such that $OPT(n, v) \leq W$.

$OPT(i, 0) = 0$ for every $i \geq 1$.

$OPT(i, v) = \max(OPT(i-1, v), w_i + OPT(i-1, v-w_i))$, otherwise.

Can find items in the solution by tracing back.

Running time is $O(n^2 v^*)$, which is pseudo-polynomial in the input size.
Dynamic Programming for Knapsack

**Knapsack**

**INSTANCE:** A set of $n$ elements, with each element $i$ having a weight $w_i$ and a value $v_i$, and a knapsack capacity $W$.

**SOLUTION:** A subset $S$ of items such that $\sum_{i \in S} v_i$ is maximised subject to the constraint $\sum_{i \in S} w_i \leq W$.

- Can generalize the dynamic program for *Subset Sum*.
- But we will develop a different dynamic program that will be useful later.
- $OPT(i, v)$ is the smallest knapsack weight so that there is a solution using only the first $i$ items with total value $\geq v$.
- What are the ranges of $i$ and $v$?
Dynamic Programming for Knapsack

**Knapsack**

**INSTANCE:** A set of \( n \) elements, with each element \( i \) having a weight \( w_i \) and a value \( v_i \), and a knapsack capacity \( W \).

**SOLUTION:** A subset \( S \) of items such that \( \sum_{i \in S} v_i \) is maximised subject to the constraint \( \sum_{i \in S} w_i \leq W \).

- Can generalize the dynamic program for **Subset Sum**.
- But we will develop a different dynamic program that will be useful later.
- \( OPT(i, v) \) is the smallest knapsack weight so that there is a solution using only the first \( i \) items with total value \( \geq v \).
- What are the ranges of \( i \) and \( v \)?
  - \( i \) ranges between 0 and \( n \), the number of items.
  - Given \( i, v \) ranges between 0 and \( \sum_{1 \leq j \leq i} v_j \).
  - Largest value of \( v \) is \( \sum_{1 \leq j \leq n} v_j \leq nv^* \), where \( v^* = \max_i v_i \).
- The solution we want is

Dynamic Programming for Knapsack

Knapsack

Instance: A set of $n$ elements, with each element $i$ having a weight $w_i$ and a value $v_i$, and a knapsack capacity $W$.

Solution: A subset $S$ of items such that $\sum_{i \in S} v_i$ is maximised subject to the constraint $\sum_{i \in S} w_i \leq W$.

- Can generalize the dynamic program for Subset Sum.
- But we will develop a different dynamic program that will be useful later.
- $OPT(i, v)$ is the smallest knapsack weight so that there is a solution using only the first $i$ items with total value $\geq v$.

What are the ranges of $i$ and $v$?

- $i$ ranges between 0 and $n$, the number of items.
- Given $i$, $v$ ranges between 0 and $\sum_{1 \leq j \leq i} v_j$.
- Largest value of $v$ is $\sum_{1 \leq j \leq n} v_j \leq n v^*$, where $v^* = \max_i v_i$.

The solution we want is the largest value $v$ such that $OPT(n, v) \leq W$. 
Dynamic Programming for Knapsack

Knapsack

Instance: A set of \( n \) elements, with each element \( i \) having a weight \( w_i \) and a value \( v_i \), and a knapsack capacity \( W \).

Solution: A subset \( S \) of items such that \( \sum_{i \in S} v_i \) is maximised subject to the constraint \( \sum_{i \in S} w_i \leq W \).

- Can generalize the dynamic program for Subset Sum.
- But we will develop a different dynamic program that will be useful later.
- \( \text{OPT}(i, v) \) is the smallest knapsack weight so that there is a solution using only the first \( i \) items with total value \( \geq v \).

What are the ranges of \( i \) and \( v \)?

- \( i \) ranges between 0 and \( n \), the number of items.
- Given \( i \), \( v \) ranges between 0 and \( \sum_{1 \leq j \leq i} v_j \).
- Largest value of \( v \) is \( \sum_{1 \leq j \leq n} v_j \leq nv^*, \) where \( v^* = \max_i v_i \).

The solution we want is the largest value \( v \) such that \( \text{OPT}(n, v) \leq W \).

\[
\begin{align*}
\text{OPT}(i, 0) &= 0 \quad \text{for every } i \geq 1 \\
\text{OPT}(i, v) &= \max \left( \text{OPT}(i - 1, v), w_i + \text{OPT}(i - 1, v - v_i) \right), \quad \text{otherwise}
\end{align*}
\]
Dynamic Programming for Knapsack

Knapsack

Instance: A set of \( n \) elements, with each element \( i \) having a weight \( w_i \) and a value \( v_i \), and a knapsack capacity \( W \).

Solution: A subset \( S \) of items such that \( \sum_{i \in S} v_i \) is maximised subject to the constraint \( \sum_{i \in S} w_i \leq W \).

- Can generalize the dynamic program for Subset Sum.
- But we will develop a different dynamic program that will be useful later.
- \( OPT(i, v) \) is the smallest knapsack weight so that there is a solution using only the first \( i \) items with total value \( \geq v \).

What are the ranges of \( i \) and \( v \)?

- \( i \) ranges between 0 and \( n \), the number of items.
- Given \( i \), \( v \) ranges between 0 and \( \sum_{1 \leq j \leq i} v_j \).
- Largest value of \( v \) is \( \sum_{1 \leq j \leq n} v_j \leq nv^*, \) where \( v^* = \max_i v_i \).

The solution we want is the largest value \( v \) such that \( OPT(n, v) \leq W \).

\[
\begin{align*}
OPT(i, 0) &= 0 \quad \text{for every } i \geq 1 \\
OPT(i, v) &= \max \left( OPT(i - 1, v), w_i + OPT(i - 1, v - v_i) \right), \quad \text{otherwise}
\end{align*}
\]

- Can find items in the solution by tracing back.
- Running time is \( O(n^2 v^*) \), which is pseudo-polynomial in the input size.
Intuition Underlying Approximation Algorithm

What is the running time if all values are the same?
Intuition Underlying Approximation Algorithm

- What is the running time if all values are the same? Polynomial.
- What is the running time if all values are small integers?
Intuition Underlying Approximation Algorithm

- What is the running time if all values are the same? Polynomial.
- What is the running time if all values are small integers? Also polynomial.
- Idea:
  - Round and scale all the values to lie in a smaller range.
  - Run the dynamic programming algorithm with the modified new values.
  - Return the items in this optimal solution.
  - Prove that the value of this solution is not much smaller than the true optimum.
Polynomial-Time Approximation Scheme for Knapsack

- 0 $< \varepsilon < 1$ is a “precision” parameter; assume that $1/\varepsilon$ is an integer.
- Scaling factor $\theta = \frac{\varepsilon v^*}{2n}$.
- For every item $i$, set

$$\tilde{v}_i = \left\lceil \frac{v_i}{\theta} \right\rceil \theta, \quad \hat{v}_i = \left\lfloor \frac{v_i}{\theta} \right\rfloor$$
Polynomial-Time Approximation Scheme for Knapsack

- $0 < \varepsilon < 1$ is a “precision” parameter; assume that $1/\varepsilon$ is an integer.
- Scaling factor $\theta = \frac{\varepsilon v^*}{2n}$.
- For every item $i$, set
  \[ \tilde{v}_i = \left\lceil \frac{v_i}{\theta} \right\rceil \theta, \quad \hat{v}_i = \left\lfloor \frac{v_i}{\theta} \right\rfloor \theta \]

$\text{Knapsack-Approx}(\varepsilon)$
- Solve the Knapsack problem using the dynamic program with the values $\hat{v}_i$.
- Return the set $S$ of items found.

What is the running time of $\text{Knapsack-Approx}$?

\[ O(n^2 \max_i \hat{v}_i) = O(n^2 \frac{v^*}{\theta}) = O(n^3 / \varepsilon) \]
Polynomial-Time Approximation Scheme for Knapsack

- $0 < \varepsilon < 1$ is a “precision” parameter; assume that $1/\varepsilon$ is an integer.
- Scaling factor $\theta = \frac{\varepsilon v^*}{2n}$.
- For every item $i$, set
  \[
  \tilde{v}_i = \left\lfloor \frac{v_i}{\theta} \right\rfloor \theta, \quad \hat{v}_i = \left\lceil \frac{v_i}{\theta} \right\rceil
  \]

**Knapsack-Approx($\varepsilon$)**

Solve the Knapsack problem using the dynamic program with the values $\hat{v}_i$. Return the set $S$ of items found.

- What is the running time of Knapsack-Approx?
Polynomial-Time Approximation Scheme for Knapsack

0 < \varepsilon < 1 is a “precision” parameter; assume that 1/\varepsilon is an integer.
Scaling factor \theta = \frac{\varepsilon v^*}{2n}.
For every item \(i\), set
\[
\tilde{v}_i = \left\lceil \frac{v_i}{\theta} \right\rceil \theta, \quad \hat{v}_i = \left\lceil \frac{v_i}{\theta} \right\rceil
\]

Knapsack-Approx(\varepsilon)
Solve the Knapsack problem using the dynamic program with the values \(\hat{v}_i\).
Return the set \(S\) of items found.

What is the running time of Knapsack-Approx?
\[O(n^2 \max_i \hat{v}_i) = O(n^2 v^*/\theta) = O(n^3 / \varepsilon)\]

We need to show that the value of the solution returned by Knapsack-Approx is good.
Approximation Guarantee for Knapsack-Approx

1. Let $S$ be the solution computed by Knapsack-Approx.
2. Let $S^*$ be any other solution satisfying $\sum_{j \in S^*} w_j \leq W$. 

How can we do better?

Improve running time to $O(n \log \frac{1}{\epsilon} + \frac{1}{\epsilon^4})$ (Lawler, 1979).
Approximation Guarantee for Knapsack-Approx

- Let $S$ be the solution computed by Knapsack-Approx.
- Let $S^*$ be any other solution satisfying \( \sum_{j \in S^*} w_j \leq W \).
- Claim: \( \sum_{i \in S} v_i \geq \sum_{j \in S^*} v_j \). Polynomial-time approximation scheme.
Approximation Guarantee for Knapsack-Approx

- Let $S$ be the solution computed by Knapsack-Approx.
- Let $S^*$ be any other solution satisfying $\sum_{j \in S^*} w_j \leq W$.
- Claim: $(1 + \varepsilon) \sum_{i \in S} v_i \geq \sum_{j \in S^*} v_j$. Polynomial-time approximation scheme.
- Since Knapsack-Approx is optimal for the values $\tilde{v}_i$,

$$\sum_{i \in S} \tilde{v}_i \geq \sum_{j \in S^*} \tilde{v}_j$$

How can we do better?

Improve running time to $O(n \log \frac{1}{\varepsilon} + \frac{1}{\varepsilon})$ (Lawler, 1979).
Approximation Guarantee for Knapsack-Approx

- Let $S$ be the solution computed by Knapsack-Approx.
- Let $S^*$ be any other solution satisfying $\sum_{j \in S^*} w_j \leq W$.
- Claim: $(1 + \varepsilon) \sum_{i \in S} v_i \geq \sum_{j \in S^*} v_j$. Polynomial-time approximation scheme.
- Since Knapsack-Approx is optimal for the values $\tilde{v}_i$,

$$\sum_{i \in S} \tilde{v}_i \geq \sum_{j \in S^*} \tilde{v}_j$$

- Since for each $i$, $v_i \leq \tilde{v}_i \leq v_i + \theta$,

$$\sum_{j \in S^*} v_j \leq \sum_{j \in S^*} \tilde{v}_j \leq \sum_{i \in S} \tilde{v}_i \leq \sum_{i \in S} v_i + n\theta = \sum_{i \in S} v_i + \frac{\varepsilon v^*}{2}$$

How can we do better?

Improve running time to $O\left(\frac{n \log 2}{1 + \varepsilon} + 1\right)$ (Lawler, 1979).
**Approximation Guarantee for Knapsack-Approx**

- Let $S$ be the solution computed by Knapsack-Approx.
- Let $S^*$ be any other solution satisfying $\sum_{j \in S^*} w_j \leq W$.
- Claim: $(1 + \varepsilon) \sum_{i \in S} v_i \geq \sum_{j \in S^*} v_j$. Polynomial-time approximation scheme.
- Since Knapsack-Approx is optimal for the values $\tilde{v}_i$,

$$\sum_{i \in S} \tilde{v}_i \geq \sum_{j \in S^*} \tilde{v}_j$$

- Since for each $i$, $v_i \leq \tilde{v}_i \leq v_i + \theta$,

$$\sum_{j \in S^*} v_j \leq \sum_{j \in S^*} \tilde{v}_j \leq \sum_{i \in S} \tilde{v}_i \leq \sum_{i \in S} v_i + n \theta = \sum_{i \in S} v_i + \frac{\varepsilon v^*}{2}$$

- Apply argument to $S^*$ containing only the item with largest value:

$$v^* \leq \sum_{i \in S} v_i + \frac{\varepsilon v^*}{2} \leq \sum_{i \in S} v_i + \frac{v^*}{2}, \text{ i.e., } v^* \leq 2 \sum_{i \in S} v_i.$$
Approximation Guarantee for Knapsack-Approx

- Let $S$ be the solution computed by Knapsack-Approx.
- Let $S^*$ be any other solution satisfying $\sum_{j \in S^*} w_j \leq W$.
- Claim: $(1 + \epsilon) \sum_{i \in S} v_i \geq \sum_{j \in S^*} v_j$. Polynomial-time approximation scheme.
- Since Knapsack-Approx is optimal for the values $\tilde{v}_i$,

$$\sum_{i \in S} \tilde{v}_i \geq \sum_{j \in S^*} \tilde{v}_j$$

- Since for each $i$, $v_i \leq \tilde{v}_i \leq v_i + \theta$,

$$\sum_{j \in S^*} v_j \leq \sum_{j \in S^*} \tilde{v}_j \leq \sum_{i \in S} \tilde{v}_i \leq \sum_{i \in S} v_i + n\theta = \sum_{i \in S} v_i + \frac{\epsilon v^*}{2}$$

- Apply argument to $S^*$ containing only the item with largest value:

$v^* \leq \sum_{i \in S} v_i + \frac{\epsilon v^*}{2} \leq \sum_{i \in S} v_i + \frac{v^*}{2}$, i.e., $v^* \leq 2 \sum_{i \in S} v_i$.
- Therefore,

$$\sum_{j \in S^*} v_j \leq \sum_{i \in S} v_i + \frac{\epsilon v^*}{2} \leq (1 + \epsilon) \sum_{i \in S} v_i$$

How can we do better?

Improve running time to $O(n \log_2 \frac{1}{\epsilon} + \frac{1}{\epsilon^4})$ (Lawler, 1979).
Approximation Guarantee for Knapsack-Approx

- Let $S$ be the solution computed by Knapsack-Approx.
- Let $S^*$ be any other solution satisfying $\sum_{j \in S^*} w_j \leq W$.
- Claim: $(1 + \varepsilon) \sum_{i \in S} v_i \geq \sum_{j \in S^*} v_j$. Polynomial-time approximation scheme.
- Since Knapsack-Approx is optimal for the values $\tilde{v}_i$,

$$\sum_{i \in S} \tilde{v}_i \geq \sum_{j \in S^*} \tilde{v}_j$$

- Since for each $i$, $v_i \leq \tilde{v}_i \leq v_i + \theta$,

$$\sum_{j \in S^*} v_j \leq \sum_{j \in S^*} \tilde{v}_j \leq \sum_{i \in S} \tilde{v}_i \leq \sum_{i \in S} v_i + n\theta = \sum_{i \in S} v_i + \frac{\varepsilon v^*}{2}$$

- Apply argument to $S^*$ containing only the item with largest value:

$$v^* \leq \sum_{i \in S} v_i + \frac{\varepsilon v^*}{2} \leq \sum_{i \in S} v_i + \frac{v^*}{2}, \text{ i.e., } v^* \leq 2 \sum_{i \in S} v_i.$$

- Therefore,

$$\sum_{j \in S^*} v_j \leq \sum_{i \in S} v_i + \frac{\varepsilon v^*}{2} \leq (1 + \varepsilon) \sum_{i \in S} v_i$$

- How can we do better?
Approximation Guarantee for Knapsack-Approx

- Let $S$ be the solution computed by Knapsack-Approx.
- Let $S^*$ be any other solution satisfying $\sum_{j \in S^*} w_j \leq W$.
- Claim: $(1 + \varepsilon) \sum_{i \in S} v_i \geq \sum_{j \in S^*} v_j$. Polynomial-time approximation scheme.
- Since Knapsack-Approx is optimal for the values $\tilde{v}_i$,

$$\sum_{i \in S} \tilde{v}_i \geq \sum_{j \in S^*} \tilde{v}_j$$

- Since for each $i$, $v_i \leq \tilde{v}_i \leq v_i + \theta$,

$$\sum_{j \in S^*} v_j \leq \sum_{j \in S^*} \tilde{v}_j \leq \sum_{i \in S} \tilde{v}_i \leq \sum_{i \in S} v_i + n\theta = \sum_{i \in S} v_i + \frac{\varepsilon v^*}{2}$$

- Apply argument to $S^*$ containing only the item with largest value: $v^* \leq \sum_{i \in S} v_i + \frac{\varepsilon v^*}{2} \leq \sum_{i \in S} v_i + \frac{v^*}{2}$, i.e., $v^* \leq 2 \sum_{i \in S} v_i$.

- Therefore,

$$\sum_{j \in S^*} v_j \leq \sum_{i \in S} v_i + \frac{\varepsilon v^*}{2} \leq (1 + \varepsilon) \sum_{i \in S} v_i$$

- How can we do better? Improve running time to $O(n \log_2 \frac{1}{\varepsilon} + \frac{1}{\varepsilon^4})$ (Lawler, 1979).
Set Cover

**Set Cover**

**INSTANCE:** A set $U$ of $n$ elements, a collection $S_1, S_2, \ldots, S_m$ of subsets of $U$, each with an associated weight $w$.

**SOLUTION:** A collection $\mathcal{C}$ of sets in the collection such that $\bigcup_{S_i \in \mathcal{C}} S_i = U$ and $\sum_{S_i \in \mathcal{C}} w_i$ is minimised.
Greedy Approach

1.1

1

1.1

1

1

1

1

1

1

1

1

1

1

1

1

1
Greedy Approach

1.1

1

1.1

1

1

1

1

1

1

Solving \( NP \)-Complete Problems

Small Vertex Covers

Trees

Approx. Vertex Cover

Load Balancing

Knapsack

Set Cover

T. M. Murali

May 3, 5, 2021

Coping with NP-Completeness
Greedy Approach

1.1

1

1

0.5

0.25

0.25

0.25

0.25

1

2

3

4

5

6

7

8

T. M. Murali May 3, 5, 2021 Coping with NP-Completeness
Greedy Approach

Solving \( \mathcal{NP} \)-Complete Problems
Small Vertex Covers
Trees
Approx. Vertex Cover
Load Balancing
Knapsack
Set Cover

Coping with NP-Completeness
Solving NP-Complete Problems

- Small Vertex Covers
- Trees
- Approx. Vertex Cover
- Load Balancing
- Knapsack
- Set Cover

Greedy Approach

1.1
1.1

1
1
1
1
1
1
1
1

1
1
1
1
1
1
1
1

1
1
1
1
1
1
1
1

1
1
1
1
1
1
1
1

1
1
1
1
1
1
1
1

1
1
1
1
1
1
1
1

1
1
1
1
1
1
1
1

0.25
0.25
0.25
0.25
0.25
0.25
0.25
0.25

0.5
0.5
0.5
0.5
0.5
0.5
0.5
0.5

T. M. Murali May 3, 5, 2021 Coping with NP-Completeness
Greedy-Set-Cover

To get a greedy algorithm, in what order should we process the sets?

Maintain set $R$ of uncovered elements. Process set in decreasing order of $w_i / |S_i ∩ R|$. The algorithm computes a set cover whose weight is at most $O(\log n)$ times the optimal weight (Johnson 1974, Lovász 1975, Chvatal 1979).
Greedy-Set-Cover

- To get a greedy algorithm, in what order should we process the sets?
- Maintain set $R$ of uncovered elements.
- Process set in decreasing order of $w_i/|S_i \cap R|$.

The algorithm computes a set cover whose weight is at most $O(\log n)$ times the optimal weight (Johnson 1974, Lovász 1975, Chvatal 1979).
**Greedy-Set-Cover**

- To get a greedy algorithm, in what order should we process the sets?
- Maintain set $R$ of uncovered elements.
- Process set in decreasing order of $w_i/|S_i \cap R|$.

---

**Greedy-Set-Cover:**

Start with $R = U$ and no sets selected

While $R \neq \emptyset$

- Select set $S_i$ that minimizes $w_i/|S_i \cap R|
- Delete set $S_i$ from $R$

EndWhile

Return the selected sets
Greedy-Set-Cover

- To get a greedy algorithm, in what order should we process the sets?
- Maintain set $R$ of uncovered elements.
- Process set in decreasing order of $w_i/|S_i \cap R|$.

Greedy-Set-Cover:

Start with $R = U$ and no sets selected

While $R \neq \emptyset$
  
  Select set $S_i$ that minimizes $w_i/|S_i \cap R|$
  
  Delete set $S_i$ from $R$

EndWhile

Return the selected sets

- The algorithm computes a set cover whose weight is at most $O(\log n)$ times the optimal weight (Johnson 1974, Lovász 1975, Chvatal 1979).
Add Bookkeeping to Greedy-Set-Cover

- Good lower bounds on the weight $w^*$ of the optimum set cover are not easy to obtain.
Add Bookkeeping to Greedy-Set-Cover

- Good lower bounds on the weight $w^*$ of the optimum set cover are not easy to obtain.
- Bookkeeping: record the per-element *cost* paid when selecting $S_i$. 

\[ \text{Define } c_t = \frac{w_i}{|S_i \cap R|} \text{ for all } t \in S_i \cap R. \]

As each set $S_i$ is selected, distribute its weight over the costs $c_t$ of the newly-covered elements.

Each element in the universe assigned cost exactly once.
Good lower bounds on the weight $w^*$ of the optimum set cover are not easy to obtain.

Bookkeeping: record the per-element cost paid when selecting $S_i$.

In the algorithm, after selecting $S_i$, add the line

Define $c_t = w_i / |S_i \cap R|$ for all $t \in S_i \cap R$.

As each set $S_i$ is selected, distribute its weight over the costs $c_t$ of the newly-covered elements.

Each element in the universe assigned cost exactly once.
Good lower bounds on the weight $w^*$ of the optimum set cover are not easy to obtain.

Bookkeeping: record the per-element *cost* paid when selecting $S_i$.

In the algorithm, after selecting $S_i$, add the line

Define $c_t = w_i/|S_i \cap R|$ for all $t \in S_i \cap R$.

As each set $S_i$ is selected, distribute its weight over the costs $c_t$ of the newly-covered elements.

Each element in the universe assigned cost exactly once.
Starting the Analysis of Greedy-Set-Cover

Let \( C \) be the set cover computed by \textsc{Greedy-Set-Cover}.

Claim: \( \sum_{S_i \in C} w_i = \sum_{t \in U} c_s \).

\[
\sum_{S_i \in C} w_i = \sum_{S_i \in C} \left( \sum_{t \in S_i \cap R} c_s \right), \text{ by definition of } c_s
\]

\[
= \sum_{t \in U} c_t, \text{ since each element in the universe contributes exactly once}
\]

In other words, the total weight of the solution computed by \textsc{Greedy-Set-Cover} is the sum of the costs it assigns to the elements in the universe.

Can “switch” between set-based weight of solution and element-based costs.

Note: sets have weights whereas \textsc{Greedy-Set-Cover} assigns costs to elements.
Intuition Behind the Proof

- Suppose $C^*$ is the optimal set cover: $w^* = \sum_{S_j \in C^*} w_j$.
- Goal is to relate total weight of sets in $C$ to total weight of sets in $C^*$.
Intuition Behind the Proof

- Suppose \( C^* \) is the optimal set cover: \( w^* = \sum_{S_j \in C^*} w_j \).
- Goal is to relate total weight of sets in \( C \) to total weight of sets in \( C^* \).
- What is the total cost assigned by \textsc{Greedy-Set-Cover} to the elements in the sets in the optimal cover \( C^* \)?
Intuition Behind the Proof

- Suppose \( C^* \) is the optimal set cover: \( w^* = \sum_{S_j \in C^*} w_j \).
- Goal is to relate total weight of sets in \( C \) to total weight of sets in \( C^* \).
- What is the total cost assigned by Greedy-Set-Cover to the elements in the sets in the optimal cover \( C^* \)?

- Since \( C^* \) is a set cover, \( \sum_{S_j \in C^*} \left( \sum_{t \in S_j} c_t \right) \geq \sum_{t \in U} c_t = \sum_{S_i \in C} w_i = w \).
Intuition Behind the Proof

- Suppose $C^*$ is the optimal set cover: $w^* = \sum_{S_j \in C^*} w_j$.
- Goal is to relate total weight of sets in $C$ to total weight of sets in $C^*$.
- What is the total cost assigned by \textsc{Greedy-Set-Cover} to the elements in the sets in the optimal cover $C^*$?

- Since $C^*$ is a set cover, $\sum_{S_j \in C^*} \left( \sum_{t \in S_j} c_t \right) \geq \sum_{t \in U} c_t = \sum_{S_i \in C} w_i = w$.
- In the sum on the left, $S_j$ is a set in $C^*$ (need not be a set in $C$). How large can total cost of elements in such a set be?
**Intuition Behind the Proof**

- Suppose $C^*$ is the optimal set cover: $w^* = \sum_{S_j \in C^*} w_j$.
- Goal is to relate total weight of sets in $C$ to total weight of sets in $C^*$.
- What is the total cost assigned by Greedy-Set-Cover to the elements in the sets in the optimal cover $C^*$?

Since $C^*$ is a set cover, $\sum_{S_j \in C^*} \left( \sum_{t \in S_j} c_t \right) \geq \sum_{t \in U} c_t = \sum_{S_i \in C} w_i = w$.

In the sum on the left, $S_j$ is a set in $C^*$ (need not be a set in $C$). How large can total cost of elements in such a set be?

For any set $S_k$, suppose we can prove $\sum_{t \in S_k} c_t \leq \alpha w_k$, for some fixed $\alpha > 0$, i.e., total cost assigned by Greedy-Set-Cover to the elements in $S_k$ cannot be much larger than the weight of $s_k$. 

---

T. M. Murali  May 3, 5, 2021  Coping with NP-Completeness
Intuition Behind the Proof

- Suppose \( C^* \) is the optimal set cover: \( w^* = \sum_{S_j \in C^*} w_j \).
- Goal is to relate total weight of sets in \( C \) to total weight of sets in \( C^* \).
- What is the total cost assigned by \texttt{Greedy-Set-Cover} to the elements in the sets in the optimal cover \( C^* \)?

Since \( C^* \) is a set cover, \[ \sum_{S_j \in C^*} \left( \sum_{t \in S_j} c_t \right) \geq \sum_{t \in U} c_t = \sum_{S_i \in C} w_i = w. \]

In the sum on the left, \( S_j \) is a set in \( C^* \) (need not be a set in \( C \)). How large can total cost of elements in such a set be?

For any set \( S_k \), suppose we can prove \( \sum_{t \in S_k} c_t \leq \alpha w_k \), for some fixed \( \alpha > 0 \), i.e., total cost assigned by \texttt{Greedy-Set-Cover} to the elements in \( S_k \) cannot be much larger than the weight of \( s_k \).

Then \[ w \leq \sum_{S_j \in C^*} \left( \sum_{t \in S_j} c_t \right) \leq \sum_{S_j \in C^*} \alpha w_j = \alpha w^* , \] giving an algorithm with approximation factor \( \alpha \).
**Intuition Behind the Proof**

- Suppose $C^*$ is the optimal set cover: $w^* = \sum_{S_j \in C^*} w_j$.
- Goal is to relate total weight of sets in $C$ to total weight of sets in $C^*$.
- What is the total cost assigned by $\textsc{Greedy-Set-Cover}$ to the elements in the sets in the optimal cover $C^*$?

Since $C^*$ is a set cover, $\sum_{S_j \in C^*} \left( \sum_{t \in S_j} c_t \right) \geq \sum_{t \in U} c_t = \sum_{S_i \in C} w_i = w$.

In the sum on the left, $S_j$ is a set in $C^*$ (need not be a set in $C$). How large can total cost of elements in such a set be?

For any set $S_k$, suppose we can prove $\sum_{t \in S_k} c_t \leq \alpha w_k$, for some fixed $\alpha > 0$, i.e., total cost assigned by $\textsc{Greedy-Set-Cover}$ to the elements in $S_k$ cannot be much larger than the weight of $s_k$.

Then $w \leq \sum_{S_j \in C^*} \left( \sum_{t \in S_j} c_t \right) \leq \sum_{S_j \in C^*} \alpha w_j = \alpha w^*$, giving an algorithm with approximation factor $\alpha$.

For every set $S_k$ in the input, goal is to prove an upper bound on $\frac{\sum_{t \in S_k} c_t}{w_k}$.
Upper Bounding Cost-by-Weight Ratio

- Consider any set $S_k$ (even one not selected by the algorithm).
- How large can $\frac{\sum_{t\in S_k} c_t}{w_k}$ get?
Upper Bounding Cost-by-Weight Ratio

- Consider any set $S_k$ (even one not selected by the algorithm).
- How large can $\frac{\sum_{t \in S_k} c_t}{w_k}$ get?
- The harmonic function
  \[ H(n) = \sum_{i=1}^{n} \frac{1}{i} = \Theta(\ln n). \]
Upper Bounding Cost-by-Weight Ratio

- Consider any set $S_k$ (even one not selected by the algorithm).
- How large can $\frac{\sum_{t \in S_k} c_t}{w_k}$ get?
- The harmonic function
  $$H(n) = \sum_{i=1}^{n} \frac{1}{i} = \Theta(\ln n).$$
- Claim: For every set $S_k$, the sum $\sum_{t \in S_k} c_t \leq H(|S_k|)w_k$. 

T. M. Murali  May 3, 5, 2021  Coping with NP-Completeness
Renumbering Elements in $S_k$

- Renumber elements in $U$ so that elements in $S_k$ are the first $d = |S_k|$ elements of $U$, i.e., $S_k = \{t_1, t_2, \ldots, t_d\}$.

- Order elements of $S_k$ in the order they get covered by the algorithm (i.e., when they get assigned a cost by Greedy-Set-Cover).
Renumbering Elements in $S_k$

- Renumber elements in $U$ so that elements in $S_k$ are the first $d = |S_k|$ elements of $U$, i.e., $S_k = \{t_1, t_2, \ldots, t_d\}$.

- Order elements of $S_k$ in the order they get covered by the algorithm (i.e., when they get assigned a cost by Greedy-Set-Cover).
Proving \( \sum_{t \in S_k} c_t \leq H(|S_k|)w_k \)

1. What happens in the iteration when the algorithm covers element \( t_j \in S_k, j \leq d \)?
Proving $\sum_{t \in S_k} c_t \leq H(\|S_K\|) w_k$

- What happens in the iteration when the algorithm covers element $t_j \in S_k, j \leq d$?
- At the start of this iteration, $R$ must contain $t_j, t_{j+1}, \ldots t_d$, i.e., $|S_k \cap R| \geq d - j + 1$. ($R$ may contain other elements of $S_k$ as well.)
Proving $\sum_{t \in S_k} c_t \leq H(|S_K|) w_k$

- What happens in the iteration when the algorithm covers element $t_j \in S_k, j \leq d$?
- At the start of this iteration, $R$ must contain $t_j, t_{j+1}, \ldots t_d$, i.e., $|S_k \cap R| \geq d - j + 1$. ($R$ may contain other elements of $S_k$ as well.)
- Therefore, $\frac{w_k}{|S_k \cap R|} \leq \frac{w_k}{d - j + 1}$.
Proving $\sum_{t \in S_k} c_t \leq H(|S_K|)w_k$

• What happens in the iteration when the algorithm covers element $t_j \in S_k, j \leq d$?

• At the start of this iteration, $R$ must contain $t_j, t_{j+1}, \ldots t_d$, i.e., $|S_k \cap R| \geq d - j + 1$. ($R$ may contain other elements of $S_k$ as well.)

• Therefore, $\frac{w_k}{|S_k \cap R|} \leq \frac{w_k}{d - j + 1}$.

• What cost did the algorithm assign to $t_j$?

• Suppose the algorithm selected set $S_i$ in this iteration.

$$c_{t_j} = \frac{w_i}{|S_i \cap R|} \leq \frac{w_k}{|S_k \cap R|} \leq \frac{w_k}{d - j + 1}.$$
Proving $\sum_{t \in S_k} c_t \leq H(|S_K|)w_k$

- What happens in the iteration when the algorithm covers element $t_j \in S_k, j \leq d$?

- At the start of this iteration, $R$ must contain $t_j, t_{j+1}, \ldots t_d$, i.e., $|S_k \cap R| \geq d - j + 1$. ($R$ may contain other elements of $S_k$ as well.)

- Therefore, $\frac{w_k}{|S_k \cap R|} \leq \frac{w_k}{d - j + 1}$.

- What cost did the algorithm assign to $t_j$?

- Suppose the algorithm selected set $S_i$ in this iteration.

$$c_{t_j} = \frac{w_i}{|S_i \cap R|} \leq \frac{w_k}{|S_k \cap R|} \leq \frac{w_k}{d - j + 1}.$$ 

- We are done!

$$\sum_{t \in S_k} c_t = \sum_{j=1}^{d} c_{s_j} \leq \sum_{j=1}^{d} \frac{w_k}{d - j + 1} = H(d)w_k.$$
Proving Upper Bound on Cost of Greedy-Set-Cover

- Let $d^*$ be the size of the largest set in the collection.
- Recall that $C^*$ is the optimal set cover and $w^* = \sum_{S_j \in C^*} w_j$. 

Let $d^*$ be the size of the largest set in the collection.
Recall that $C^*$ is the optimal set cover and $w^* = \sum_{S_j \in C^*} w_j$. 

We have proven that Greedy-Set-Cover computes a set cover whose weight is at most $H(d^*)$ times the optimal weight.
Proving Upper Bound on Cost of Greedy-Set-Cover

- Let \( d^* \) be the size of the largest set in the collection.
- Recall that \( C^* \) is the optimal set cover and \( w^* = \sum_{S_j \in C^*} w_j \).
- For each set \( S_j \) in \( C^* \), we proved \( w_j \geq \frac{\sum_{s \in S_j} c_s}{H(|S_i|)} \geq \frac{\sum_{s \in S_j} c_s}{H(d^*)} \).
- Combining with \( \sum_{S_i \in C} w_i = \sum_{t \in U} c_t \), we have

\[
    w^* = \sum_{S_j \in C^*} w_j
\]
Proving Upper Bound on Cost of Greedy-Set-Cover

- Let \( d^* \) be the size of the largest set in the collection.
- Recall that \( C^* \) is the optimal set cover and \( w^* = \sum_{j \in C^*} w_j \).
- For each set \( S_j \) in \( C^* \), we proved \( w_j \geq \frac{\sum_{s \in S_j} c_s}{H(|S_j|)} \geq \frac{\sum_{s \in S_j} c_s}{H(d^*)} \).
- Combining with \( \sum_{i \in C} w_i = \sum_{t \in U} c_t \), we have

\[
\begin{align*}
    w^* &= \sum_{j \in C^*} w_j \\
    &\geq \sum_{j \in C^*} \left( \frac{1}{H(d^*)} \sum_{t \in S_j} c_t \right) \\
    &\geq \frac{1}{H(d^*)} \sum_{t \in U} c_t
\end{align*}
\]
Let $d^*$ be the size of the largest set in the collection.

Recall that $C^*$ is the optimal set cover and $w^* = \sum_{S_j \in C^*} w_j$.

For each set $S_j$ in $C^*$, we proved $w_j \geq \frac{\sum_{s \in S_j} c_s}{H(|S_i|)} \geq \frac{\sum_{s \in S_j} c_s}{H(d^*)}$.

Combining with $\sum_{S_i \in C} w_i = \sum_{t \in U} c_t$, we have

$$w^* = \sum_{S_j \in C^*} w_j \geq \sum_{S_j \in C^*} \left( \frac{1}{H(d^*)} \sum_{t \in S_j} c_t \right) \geq \frac{1}{H(d^*)} \sum_{t \in U} c_t = \frac{1}{H(d^*)} \sum_{S_i \in C} w_i = w.$$
Proving Upper Bound on Cost of Greedy-Set-Cover

- Let $d^*$ be the size of the largest set in the collection.
- Recall that $C^*$ is the optimal set cover and $w^* = \sum_{S_j \in C^*} w_j$.
- For each set $S_j$ in $C^*$, we proved $w_j \geq \frac{\sum_{s \in S_j} c_s}{H(|S_j|)} \geq \frac{\sum_{s \in S_j} c_s}{H(d^*)}$.
- Combining with $\sum_{S_i \in C} w_i = \sum_{t \in U} c_t$, we have

$$w^* = \sum_{S_j \in C^*} w_j \geq \sum_{S_j \in C^*} \left( \frac{1}{H(d^*)} \sum_{t \in S_j} c_t \right) \geq \frac{1}{H(d^*)} \sum_{t \in U} c_t = \frac{1}{H(d^*)} \sum_{S_i \in C} w_i = w.$$ 

- We have proven that Greedy-Set-Cover computes a set cover whose weight is at most $H(d^*)$ times the optimal weight.
How Badly Can Greedy-Set-Cover Perform?

- Generalise this example to show that algorithm produces a set cover of weight $\Omega(\log n)$ even though optimal weight is $2 + \varepsilon$.
- More complex constructions show greedy algorithm incurs a weight close to $H(n)$ times the optimal weight.
How Badly Can Greedy-Set-Cover Perform?

- Generalise this example to show that algorithm produces a set cover of weight $\Omega(\log n)$ even though optimal weight is $2 + \varepsilon$.
- More complex constructions show greedy algorithm incurs a weight close to $H(n)$ times the optimal weight.
- No polynomial time algorithm can achieve an approximation bound better than $(1 - \Omega(1)) \ln n$ times optimal unless $P = NP$ (Dinur and Steurer, 2014)
Traveling Salesman Problem

- General case: Cannot be approximated within any polynomial time computable function unless $P = NP$ (Sahni, Gonzalez, 1976).
Traveling Salesman Problem

- General case: Cannot be approximated within any polynomial time computable function unless $\mathcal{P} = \mathcal{NP}$ (Sahni, Gonzalez, 1976).
- Metric TSP (distances are symmetric, positive, satisfy triangle inequality): $3/2$-factor approximation algorithm (Christofides, 1976), innapproximable to better than $123/122$ ratio unless $\mathcal{P} = \mathcal{NP}$ (Karpinski, Lampis, Schmied, 2013).
Traveling Salesman Problem

- General case: Cannot be approximated within any polynomial time computable function unless $P = NP$ (Sahni, Gonzalez, 1976).
- Metric TSP (distances are symmetric, positive, satisfy triangle inequality): 3/2-factor approximation algorithm (Christofides, 1976), inapproximable to better than 123/122 ratio unless $P = NP$ (Karpinski, Lampis, Schmied, 2013).
- 1-2 TSP: 8/7 approximation factor (Berman, Karpinski, 2006).
Traveling Salesman Problem

- General case: Cannot be approximated within any polynomial time computable function unless $P = NP$ (Sahni, Gonzalez, 1976).
- Metric TSP (distances are symmetric, positive, satisfy triangle inequality): 3/2-factor approximation algorithm (Christofides, 1976), inapproximable to better than 123/122 ratio unless $P = NP$ (Karpinski, Lampis, Schmied, 2013).
- 1-2 TSP: 8/7 approximation factor (Berman, Karpinski, 2006).
- Euclidean TSP (distances defined by points in $d$ dimensions): PTAS in $O(n(\log n)^{1/\varepsilon})$ time (Arora, 1997; Mitchell, 1999) (second algorithm is slower).
Problems in $\mathcal{P}$

- 3-SUM: Given a set of $n$ numbers, are there three elements in it whose sum is 0?
Problems in \( \mathcal{P} \)

- 3-SUM: Given a set of \( n \) numbers, are there three elements in it whose sum is 0? Can be solved in \( O(n^2) \) time.
- Many simple problems are quadratic-time reducible to 3-SUM, e.g., Given \( n \) lines in the plane, are any three concurrent?
- Conjecture: Any algorithm for this problem requires \( n^{2-o(1)} \) time.
Problems in \( \mathcal{P} \)

- **3-SUM**: Given a set of \( n \) numbers, are there three elements in it whose sum is 0? Can be solved in \( O(n^2) \) time.

- Many simple problems are quadratic-time reducible to 3-SUM, e.g., Given \( n \) lines in the plane, are any three concurrent?

- **Conjecture**: Any algorithm for this problem requires \( n^{2-o(1)} \) time.

- **All pairs shortest paths**: Any algorithm for this problem requires \( n^{3-o(1)} \) time.
Problems in \( \mathcal{P} \)

- **3-SUM**: Given a set of \( n \) numbers, are there three elements in it whose sum is 0? Can be solved in \( O(n^2) \) time.

- Many simple problems are quadratic-time reducible to 3-SUM, e.g., Given \( n \) lines in the plane, are any three concurrent?

- **Conjecture**: Any algorithm for this problem requires \( n^{2-o(1)} \) time.

- **All pairs shortest paths**: Any algorithm for this problem requires \( n^{3-o(1)} \) time.

- **Strongly exponential time hypothesis (SETH)**: For every \( \varepsilon > 0 \), there exists an integer \( k \) such that \( k \)-SAT on \( n \) variables cannot be solved in \( O(2^{(1-\varepsilon)n} \text{poly}(n)) \) time.
Problems in $\mathcal{P}$

- 3-SUM: Given a set of $n$ numbers, are there three elements in it whose sum is 0? Can be solved in $O(n^2)$ time.
- Many simple problems are quadratic-time reducible to 3-SUM, e.g., Given $n$ lines in the plane, are any three concurrent?
- Conjecture: Any algorithm for this problem requires $n^{2-o(1)}$ time.
- All pairs shortest paths: Any algorithm for this problem requires $n^{3-o(1)}$ time.
- Strongly exponential time hypothesis (SETH): For every $\varepsilon > 0$, there exists an integer $k$ such that $k$-SAT on $n$ variables cannot be solved in $O(2^{(1-\varepsilon)n}\text{poly}(n))$ time.
- Edit distance (sequence alignment) between two strings of length $n$: 

T. M. Murali
May 3, 5, 2021
Coping with NP-Completeness
Problems in \( \mathcal{P} \)

- **3-SUM**: Given a set of \( n \) numbers, are there three elements in it whose sum is 0? Can be solved in \( O(n^2) \) time.
- Many simple problems are quadratic-time reducible to 3-SUM, e.g., Given \( n \) lines in the plane, are any three concurrent?
- **Conjecture**: Any algorithm for this problem requires \( n^{2-o(1)} \) time.
- **All pairs shortest paths**: Any algorithm for this problem requires \( n^{3-o(1)} \) time.
- **Strongly exponential time hypothesis (SETH)**: For every \( \varepsilon > 0 \), there exists an integer \( k \) such that \( k \)-SAT on \( n \) variables cannot be solved in \( O(2^{(1-\varepsilon)n}\text{poly}(n)) \) time.
- **Edit distance (sequence alignment)** between two strings of length \( n \): If it can be computed in \( O(n^{2-\delta}) \) time for some constant \( \delta > 0 \), then SAT with \( n \) variables and \( m \) clauses can be solved in \( m^{O(1)}2^{(1-\varepsilon)n} \) time, for some \( \varepsilon > 0 \) (Backurs, Indyk, 2015).