# Advanced Algorithms I

Summer 2018 • Lecture 08/09

# The Rest of the Quarter

- Lectures 1-13 covered the bulk of the material, congrats!
  - This material will be emphasized on the final (~80% of points).
- Lectures 14-15 will cover additional topics.
  - Intractable problems, approximation algorithms, max-flow

# Outline for Today

- Tractability
  - TSP
- Approximation algorithms
  - Vertex Cover
  - Set Cover
  - 0/1 Knapsack

# Background

# Defining Efficiency

What is an efficient algorithm?

- An algorithm is efficient iff it runs in polynomial time on a serial computer.
- Runtimes of "efficient" algorithms: $O(n)$, $O(n\log(n))$, $O(n^8\log^4(n))$, $O(n^{1,000,000})$.
- Runtimes of "inefficient" algorithms: $O(2^n)$, $O(n!)$, $O(1.0000001^n)$.

# Some Caveats

**Parallelism** Some problems can be solved in polynomial time on machines with a polynomial number of processors.

Are all efficient algorithms parallelizable?

**Randomization** Some algorithms can be solved in expected polynomial time, or have poly-time Monte Carlo algorithms that work with high probability.

Are randomized efficient algorithms efficient solutions? P $\subseteq$? RP$\subseteq$? NP.

**Quantum computation** Some algorithms can be solved in polynomial time on a quantum computer.

Are quantum efficient algorithms efficient solutions?

**These are all open problems!**

# Tractability

A problem is called **tractable** iff there is an efficient (i.e. polynomial time) algorithm that solves it.

A problem is called **intractable** iff there is no efficient algorithm that solves it.

# NP

A **decision problem** is a problem with a yes/no answer.

The class **NP** consists of all decision problems where “yes” answers can be verified efficiently.

- Is the kth order statistic of A equal to x?
- Is there a cut in G of size at least k?

All tractable decision problems are in **NP**, plus a lot of problems whose difficulty is unknown.

# NP-Completeness

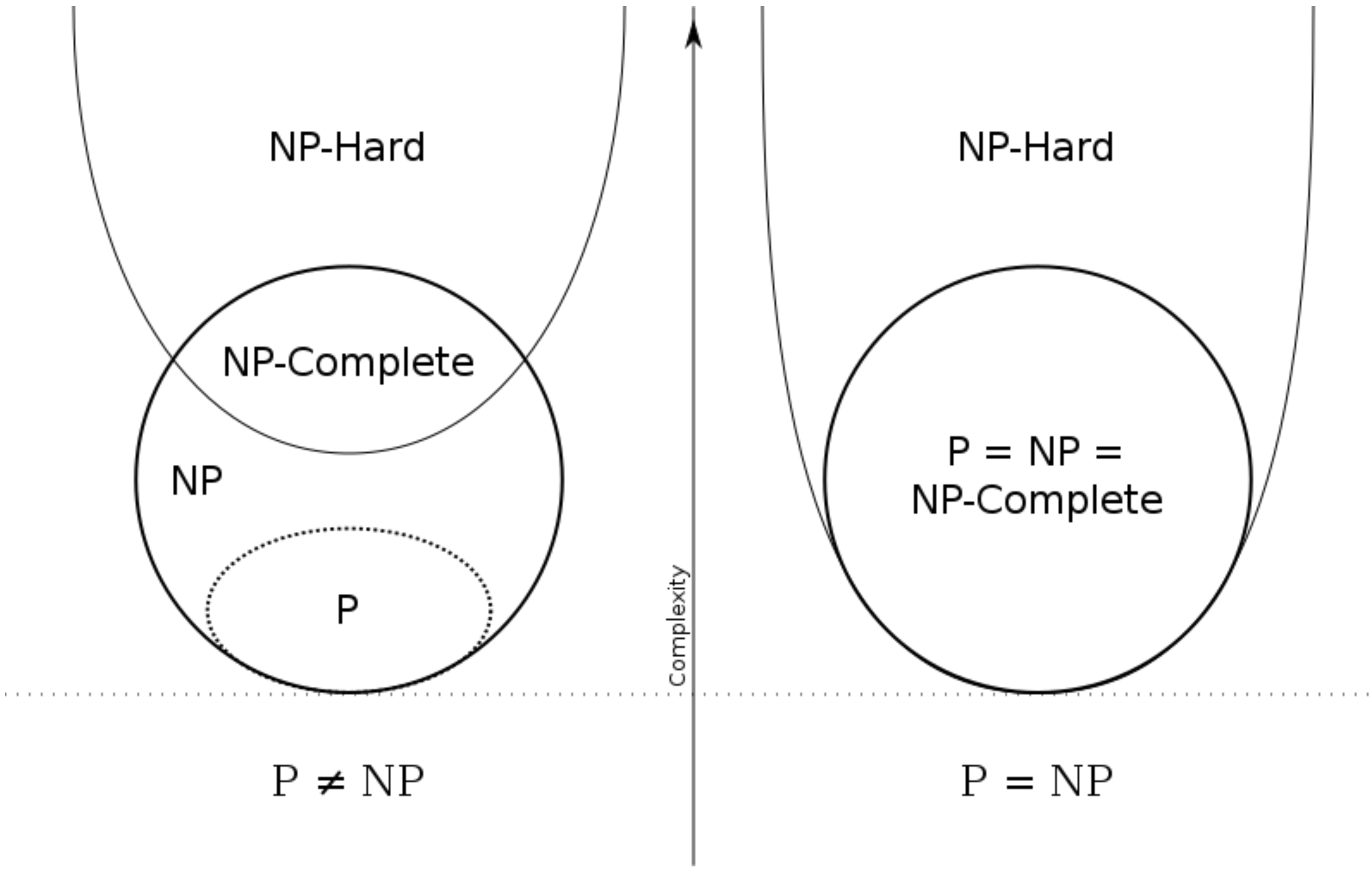
The **NP-complete** problems are (intuitively) the hardest problems in NP.

Either every **NP**-complete problem is tractable or no **NP**-complete problem is tractable.

- This is an open problem: the P = NP question!

There are no known polynomial-time algorithms for any **NP**-complete problem.

# Complexity Classes



# NP-Hardness

A problem (which may or may not be a decision problem) is called **NP-hard** if (intuitively) it is at least as hard as every problem in **NP**.

As before: no polynomial-time algorithms are known for any **NP**-hard problem.

# NP-Hardness

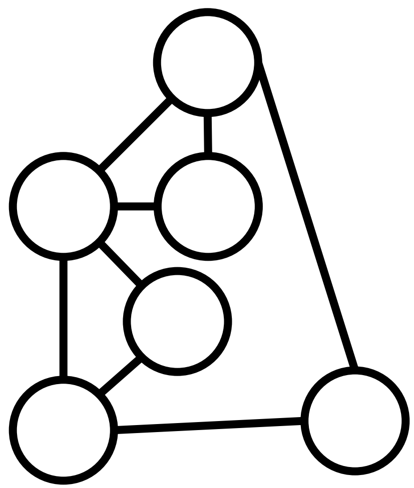
Assuming that $P \neq NP$, all NP-hard problems are intractable.

- This does not mean that brute-force algorithms are the only option.
- This does not mean that it is hard to get approximate answers.

# Traveling Salesperson Problem

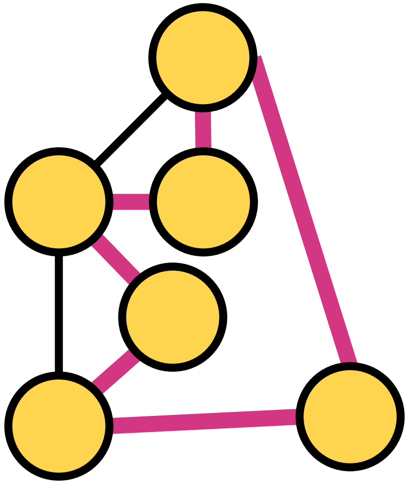
# TSP

A **Hamiltonian cycle** in an undirected graph G is a simple cycle that visits every vertex in G.



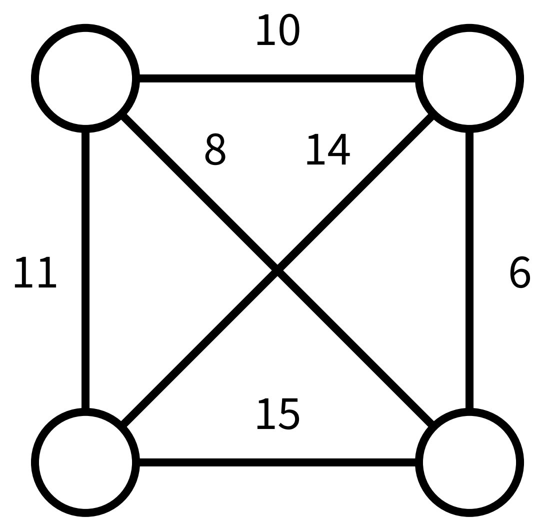
# TSP

A **Hamiltonian cycle** in an undirected graph G is a simple cycle that visits every vertex in G.



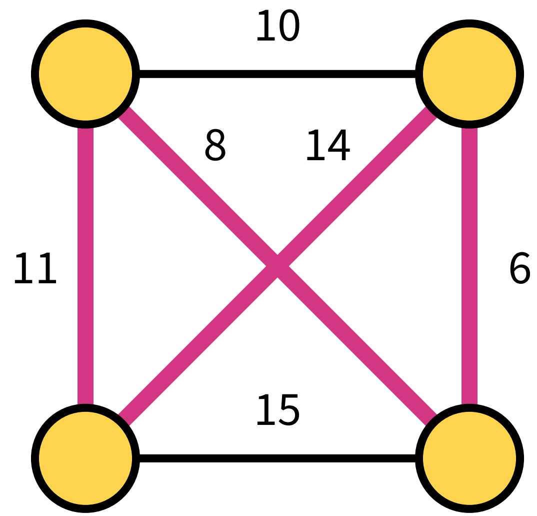
# TSP

Given a complete, undirected weighted graph G, the traveling salesman problem is to find a Hamiltonian cycle in G of least total cost.



# TSP

Given a complete, undirected weighted graph G, the traveling salesman problem is to find a Hamiltonian cycle in G of least total cost.



# TSP

**Formally** Given a complete, undirected G and a set of positive integer edge weights, the TSP is to find a Hamiltonian cycle in G with least total weight.

Note that since G is complete, there must be at least one Hamiltonian cycle. The challenge is finding the cycle with least cost.

This problem is known to be **NP**-hard.

# TSP

- Try all possible Hamiltonian cycles in the graph?
  - How many Hamiltonian cycles are there? (n-1)! / 2
  - Since each cycle takes O(n)-time, the total time is `O(n!)`.

# TSP

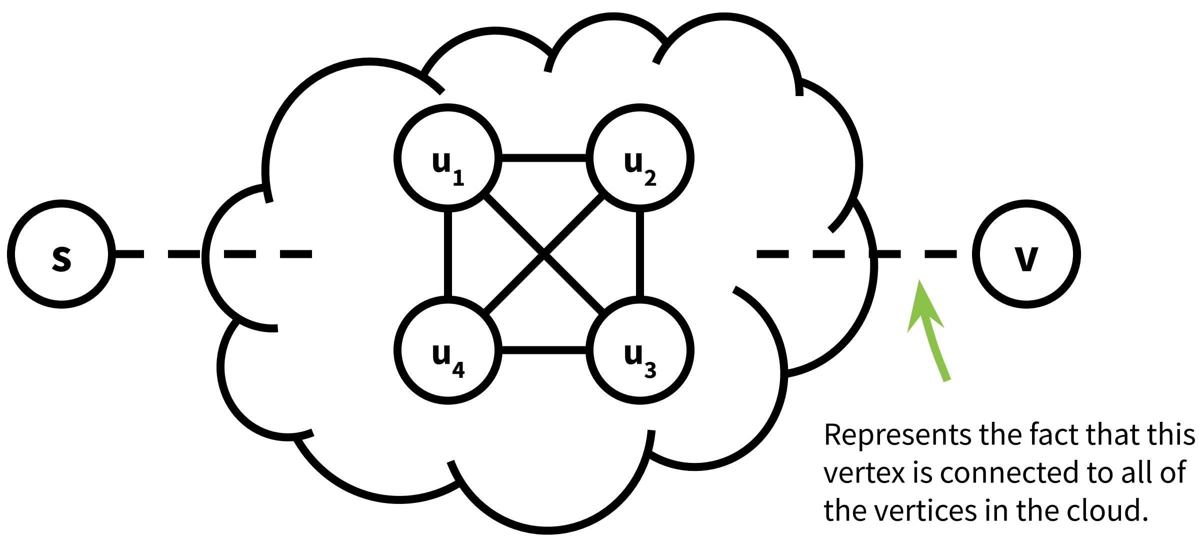
Let OPT(v, S) be the minimum cost of an s - v path that visits exactly the vertices in S. We assume v ∈ S. Let w(u, v) be the weight of the edge (u, v).

**Claim** OPT(v, S) satisfies the recurrence:

$$\mathrm{OPT}(v,S) = \begin{cases} 0 & \text{if } v = s \text{ and } S = \{s\} \\ \infty & \text{if } s \notin S \\ \min_{u \in S-\{v\}} \{\mathrm{OPT}(u, S-\{v\}) + w(u,v)\} & \text{otherwise} \end{cases}$$

# TSP

$$\text{OPT}(v,S) = \begin{cases} 0 & \text{if } v = s \text{ and } S = \{s\} \\ \infty & \text{if } s \notin S \\ \min_{u \in S-\{v\}} \{\text{OPT}(u, S-\{v\}) + w(u,v)\} & \text{otherwise} \end{cases}$$



# TSP

To solve OPT(v, S), a problem of size |S|, we need to solve subproblems of size |S| - 1.

**Idea** Evaluate the recurrence on sets of size 1, 2, 3 …, n.

There are $2^n$ possible subsets of a set S, of which $2^{n-1}$ contain s.

# TSP

```
def tsp(G):
  n = |G.V|
  DP = []  # n × 2^(n-1) table
  s = random vertex from G.V
  DP[s][{s}] = 0
  for k in range(1, n):
    for all sets S ⊆ V where |S| = k and s ∈ S:
      for all v ∈ S - {s}:
        DP[v][S] = min_{u∈S-{v}}{DP[u][S-{v}] + w(u,v)}
  return min_{v≠s}{DP[v][V] + w(v,s)}
```

**Runtime:** $O(2^n n^2)$

# TSP

```
def tsp(G):
  n = |G.V|
  DP = []  # n × 2^(n-1) table
  s = random vertex from G.V
  DP[s][{s}] = 0
  for k in range(1, n):
    for all sets S ⊆ V where |S| = k and s ∈ S:
      for all v ∈ S - {s}:
        DP[v][S] = min_{u∈S-{v}}{DP[u][S-{v}] + w(u,v)}
  return min_{v≠s}{DP[v][V] + w(v,s)}
```

**Runtime:** $O(2^n n^2)$

We'll talk about this in a bit.

# TSP

Each subset of V containing s can be mapped to a unique integer in 0, 1, 2, …, $2^{n-1}$ - 1.

- Think of the number as a bitvector where the present elements are 1s and the absent elements are 0s.
- Takes `O(n)`-time to compute the above number and index into the table, the cost per subproblem.

# TSP

- Each subset of V containing s can be mapped to a unique integer in 0, 1, 2, …, $2^{n-1} - 1$.
  - Think of the number as a bitvector where the present elements are 1s and the absent elements are 0s.
  - Takes $O(n)$-time to compute the above number and index into the table, the cost per subproblem.
- $O(2^n n^2)$ total time.
  - $O(2^n n)$ total subproblems (cells in the table).
  - Solving each subproblem requires us to look at $O(n)$ different subproblems, and $O(1)$-time for each one.
  - Map all subsets of V to bitvectors in $O(n)$-time.

# TSP

What's the difference between $n!$ and $2^n n^2$?

- Compare $20!$ and $2^{20}20^2$:
  - $20! \approx 2.4 \times 10^{18}$
  - $2^{20}20^2 \approx 4.2 \times 10^{8}$

# TSP

What's the difference between $n!$ and $2^n n^2$?

- Compare $20!$ and $2^{20}20^2$:
  - $20! \approx 2.4 \times 10^{18}$
  - $2^{20}20^2 \approx 4.2 \times 10^{8}$
- Compare $30!$ and $2^{30}30^2$:
  - $30! \approx 2.6 \times 10^{32}$
  - $2^{30}30^2 \approx 9.7 \times 10^{11}$

# TSP

What's the difference between $n!$ and $2^n n^2$?

- Compare $20!$ and $2^{20}20^2$:
  - $20! \approx 2.4 \times 10^{18}$
  - $2^{20}20^2 \approx 4.2 \times 10^{8}$
- Compare $30!$ and $2^{30}30^2$:
  - $30! \approx 2.6 \times 10^{32}$
  - $2^{30}30^2 \approx 9.7 \times 10^{11}$
- Compare $40!$ and $2^{40}40^2$:
  - $40! \approx 8.2 \times 10^{47}$
  - $2^{40}40^2 = 1.8 \times 10^{15}$

# TSP

Why this matters?

Improving upon brute-force (e.g. n!) increases the size of problems that can be solved with exact answers.

Though there might not exist a poly-time solution, an exponential solution often offers a considerable improvement.

# 0/1 Knapsack, revisited

# Knapsack

## 0/1 Knapsack

| | | | | | |
|---|---|---|---|---|---|
| weight | 6 | 2 | 4 | 3 | 11 |
| value | 20 | 8 | 14 | 13 | 35 |

Suppose I only have one copy of each item.

What's the most valuable way to fill the knapsack?


capacity: 10

Total weight: 9

Total value: 35

**Task** Find the items to put in a 0/1 knapsack.

# 0/1 Knapsack

What I didn't say is this problem is known to be **NP**-hard.

- $O(n2^n)$ Brute-force solution: try all possible subsets of the items and find the feasible set with the largest total value.
- $O(nW \log(n))$ Greedy solution: sort items by their "unit value" $v_k / w_k$.
- $O(nW)$ Dynamic programming solution

# 0/1 Knapsack

## Did we just prove P = NP?

A poly-time algorithm is one that runs in time polynomial in the total number of bits required to write out the input to the problem.

Therefore, `O(nW)` is exponential in the number of bits required to write out the input (e.g. adding one more bit to the end of the representation of W doubles Its size and doubles the runtime).

The DP runtime of `O(nW)` is better than our brute-force runtime of $O(n2^n)$, provided that $W = o(2^n)$.

That's a little-o, not a big-O.

For any fixed W, this algorithm runs in linear time!

# Parameterized Complexity

Parameterized complexity is a branch of complexity theory that studies the hardness of problems with respect to different "parameters" of the input.

- In the case of 0/1 Knapsack, `O(nW)` has two parameters: the number of items (n) and capacity (W).
- Often, **NP**-hard problems aren't entirely infeasible as long as some parameter of the problem is fixed.

# Fixed Parameter Tractability

Suppose that the input to a problem P can be characterized by two parameters, n and k.

P is called fixed-parameter tractable iff there is some algorithm that solves P in time `O(f(k)p(n))`.

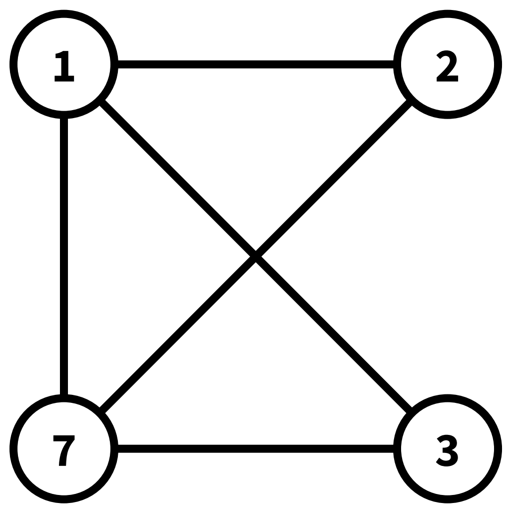
- f(k) is an arbitrary function and p(n) is a polynomial in n..
- Intuitively, for any fixed k, the algorithm runs in a polynomial in n since that polynomial p(n) does not depend on choice of k.

# Vertex Cover

# Vertex Cover

**Task** Given an undirected graph G = (V, E), and a cost function on the vertices, find a minimum cost vertex cover, i.e. a set of V’ ⊆ V such that every edge has at least one endpoint incident at V’.

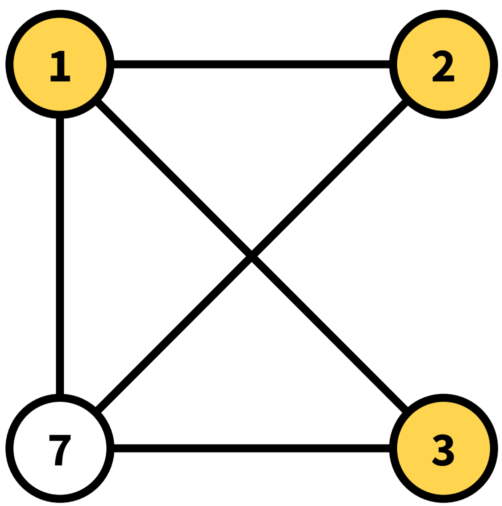
Is this easier or harder than edge cover? 🤔



# Vertex Cover

**Task** Given an undirected graph G = (V, E), and a cost function on the vertices, find a minimum cost vertex cover, i.e. a set of V' ⊆ V such that every edge has at least one endpoint incident at V'.

Is this easier or harder than edge cover? 🤔

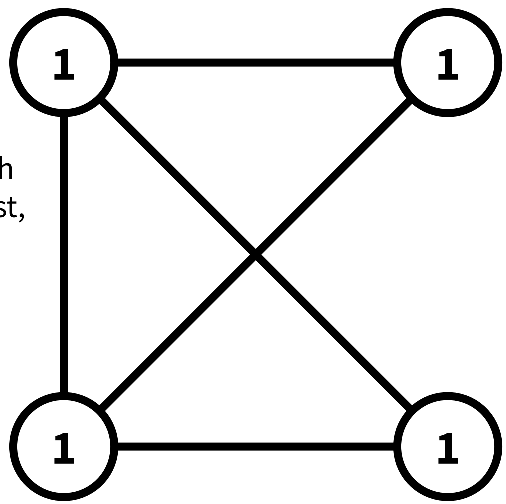


# Vertex Cover

**Task** Given an undirected graph G = (V, E), and a cost function on the vertices, find a minimum cost vertex cover, i.e. a set of V' ⊆ V such that every edge has at least one endpoint incident at V'.

Is this easier or harder than edge cover? 🤔

The special case, in which all vertices are of unit cost, is called the "cardinality vertex cover problem."

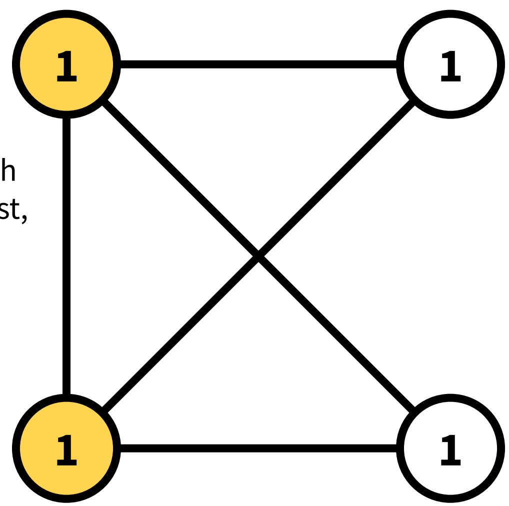


# Vertex Cover

**Task** Given an undirected graph G = (V, E), and a cost function on the vertices, find a minimum cost vertex cover, i.e. a set of V’ ⊆ V such that every edge has at least one endpoint incident at V’.

Is this easier or harder than edge cover? 🤔

The special case, in which all vertices are of unit cost, is called the “cardinality vertex cover problem.”



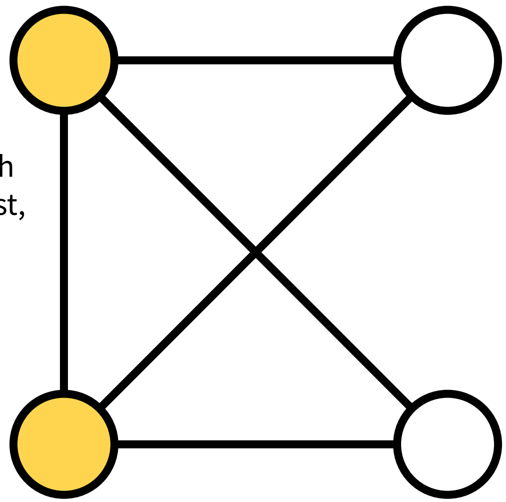
# Vertex Cover

**Task** Given an undirected graph G = (V, E), and a cost function on the vertices, find a minimum cost vertex cover, i.e. a set of V’ ⊆ V such that every edge has at least one endpoint incident at V’.

Is this easier or harder than edge cover? 🤔

The special case, in which all vertices are of unit cost, is called the “cardinality vertex cover problem.”

This will be represented with blank vertices.



# Vertex Cover

**Task** Given an undirected graph $G = (V, E)$, and a cost function on the vertices, find a minimum cost vertex cover, i.e. a set of $V' \subseteq V$ such that every edge has at least one endpoint incident at $V'$.

Is this easier or harder than edge cover? 🤔

The cardinality vertex cover problem is **NP**-hard. How might we solve it??

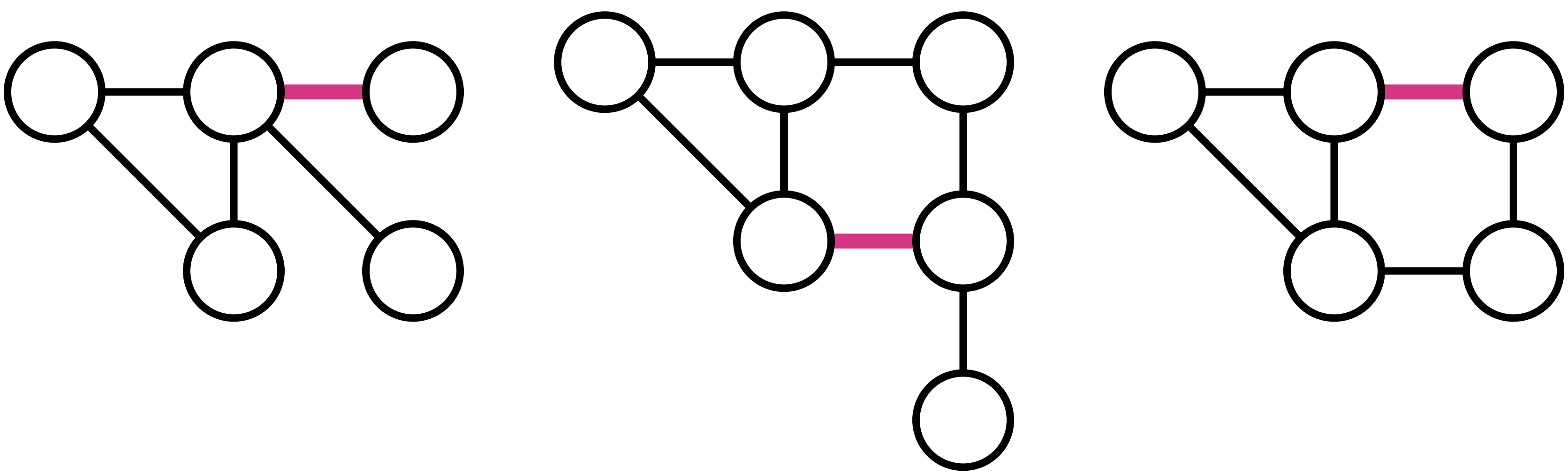
# Vertex Cover

Let's introduce some useful terminology …

A **matching** is a set of edges such that no two edges share a common vertex.

A **maximal matching** is a matching such that if any if any edge is added to the matching, it's no longer a matching.

A **maximum matching** is a maximal matching that contains the largest possible number of edges.



# Vertex Cover

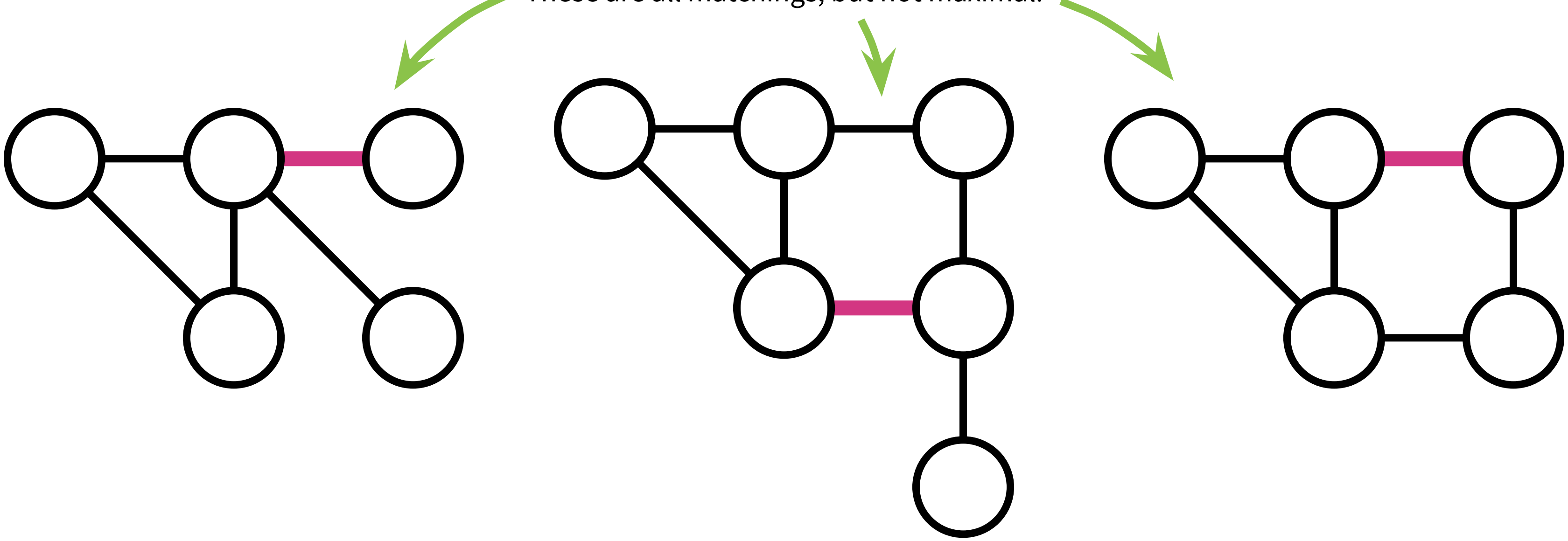
Let’s introduce some useful terminology …

A **matching** is a set of edges such that no two edges share a common vertex.

A **maximal matching** is a matching such that if any if any edge is added to the matching, it’s no longer a matching.

A **maximum matching** is a maximal matching that contains the largest possible number of edges.

These are all matchings, but not maximal.



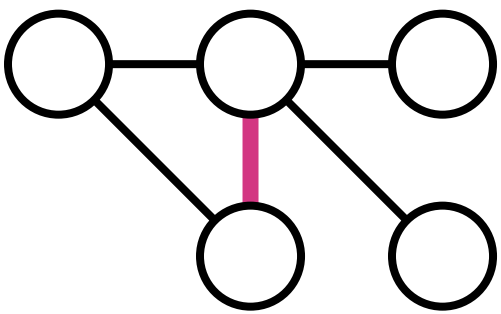
# Vertex Cover

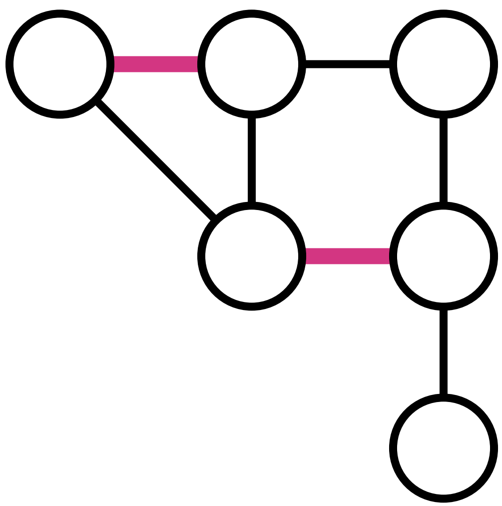
Let's introduce some useful terminology …

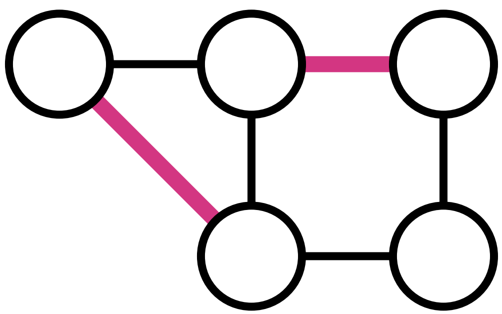
A **matching** is a set of edges such that no two edges share a common vertex.

A **maximal matching** is a matching such that if any if any edge is added to the matching, it's no longer a matching.

A **maximum matching** is a maximal matching that contains the largest possible number of edges.







# Vertex Cover

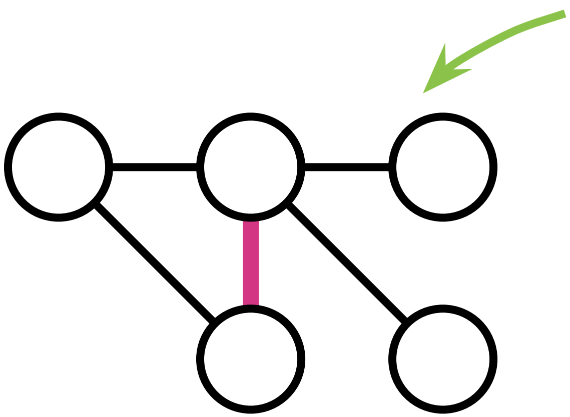
Let's introduce some useful terminology …

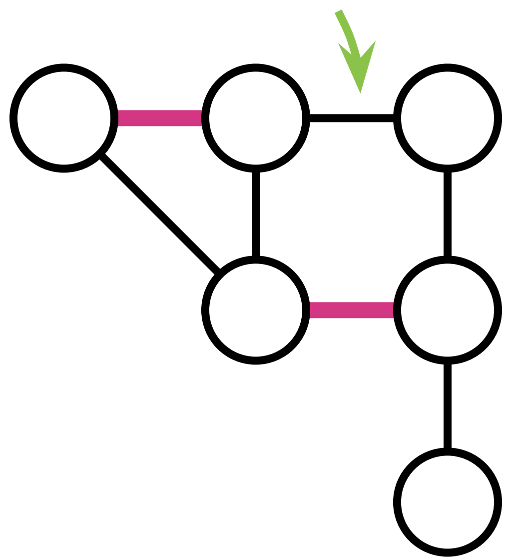
A **matching** is a set of edges such that no two edges share a common vertex.

A **maximal matching** is a matching such that if any if any edge is added to the matching, it's no longer a matching.

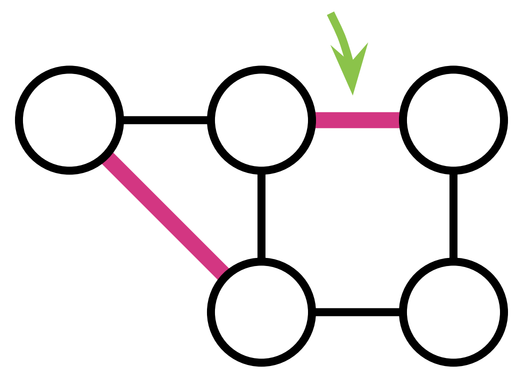
A **maximum matching** is a maximal matching that contains the largest possible number of edges.

These are maximal matchings,
but not maximum matchings.





This is a maximum matching.



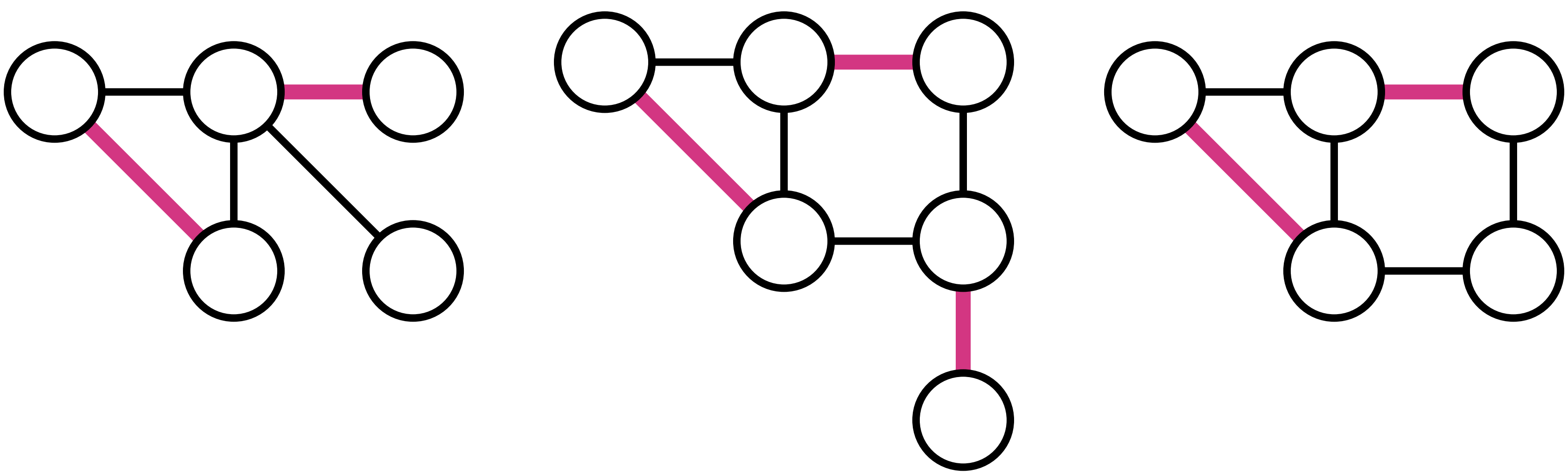
# Vertex Cover

Let's introduce some useful terminology …

A **matching** is a set of edges such that no two edges share a common vertex.

A **maximal matching** is a matching such that if any if any edge is added to the matching, it's no longer a matching.

A **maximum matching** is a maximal matching that contains the largest possible number of edges.



# Vertex Cover

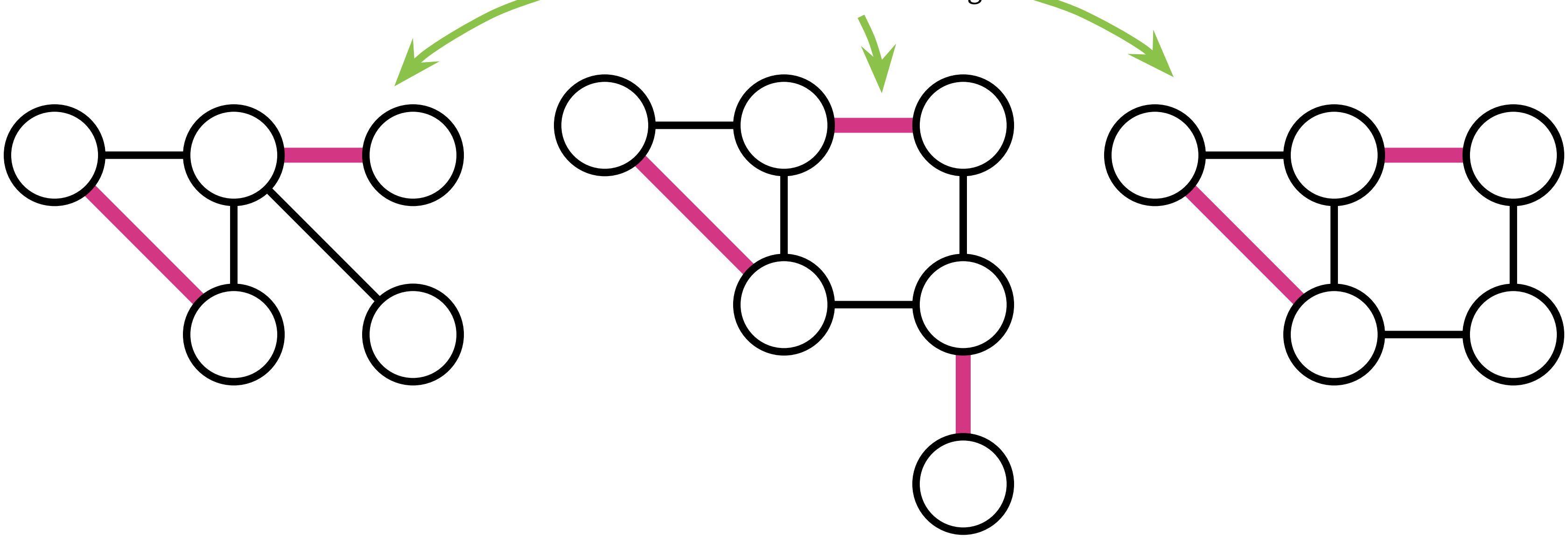
Let's introduce some useful terminology …

A **matching** is a set of edges such that no two edges share a common vertex.

A **maximal matching** is a matching such that if any if any edge is added to the matching, it's no longer a matching.

A **maximum matching** is a maximal matching that contains the largest possible number of edges.

These are all maximum matchings.



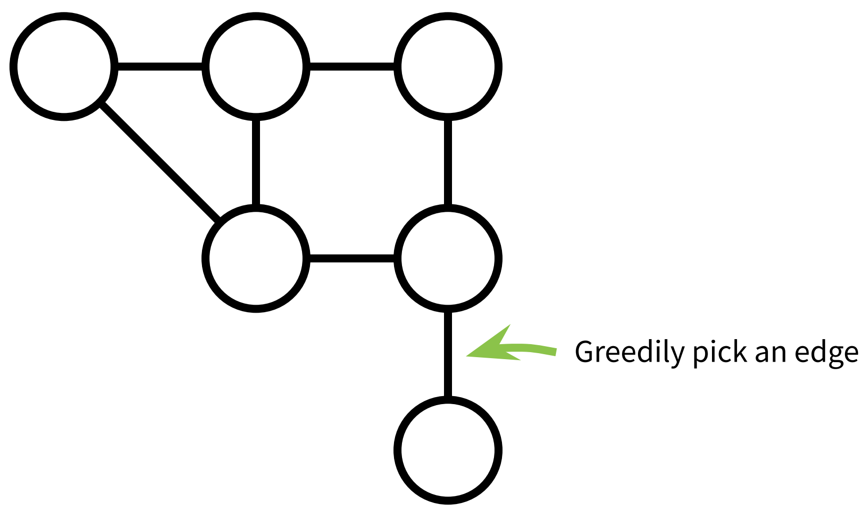
# Vertex Cover

Let's attempt to design a greedy solution first.

Recall, greedy solutions are often the most natural algorithms to design, but sometimes it's difficult to prove their correctness.

**Intuition** Greedily pick an edge, add its endpoints to the vertex cover, and remove edges adjacent to the endpoints.

i.e. Find a maximal matching in G and output the set of matched vertices.



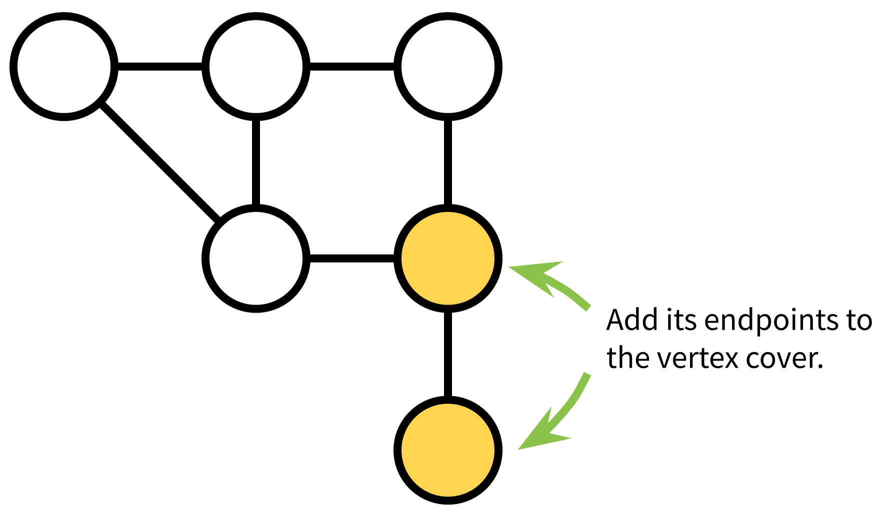
# Vertex Cover

Let's attempt to design a greedy solution first.

Recall, greedy solutions are often the most natural algorithms to design, but sometimes it's difficult to prove their correctness.

**Intuition** Greedily pick an edge, add its endpoints to the vertex cover, and remove edges adjacent to the endpoints.

i.e. Find a maximal matching in G and output the set of matched vertices.



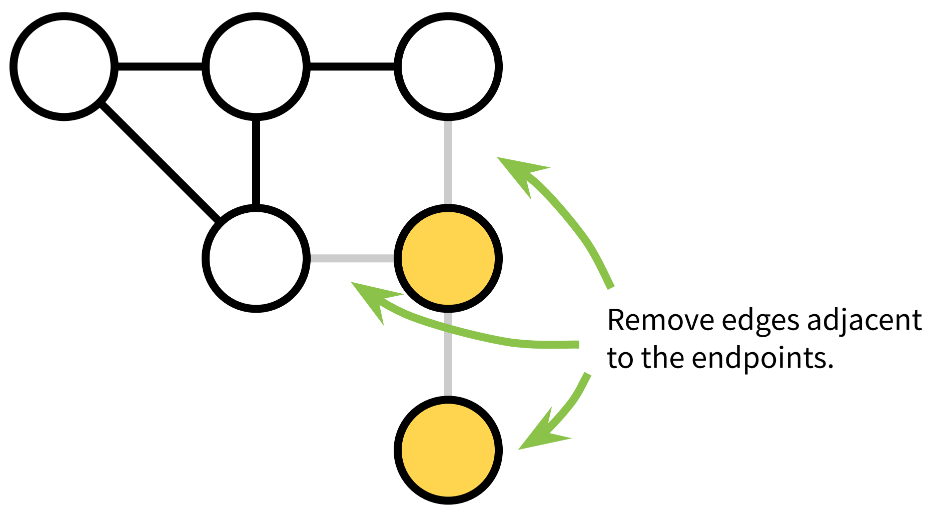
# Vertex Cover

Let's attempt to design a greedy solution first.

Recall, greedy solutions are often the most natural algorithms to design, but sometimes it's difficult to prove their correctness.

**Intuition** Greedily pick an edge, add its endpoints to the vertex cover, and remove edges adjacent to the endpoints.

i.e. Find a maximal matching in G and output the set of matched vertices.



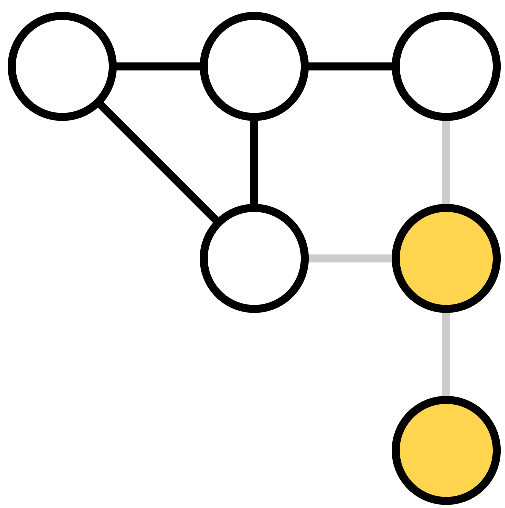
# Vertex Cover

Let's attempt to design a greedy solution first.

Recall, greedy solutions are often the most natural algorithms to design, but sometimes it's difficult to prove their correctness.

**Intuition** Greedily pick an edge, add its endpoints to the vertex cover, and remove edges adjacent to the endpoints.

i.e. Find a maximal matching in G and output the set of matched vertices.



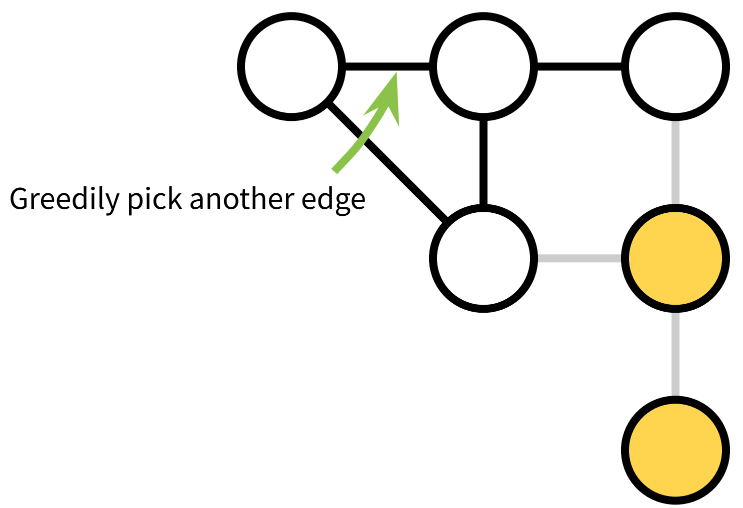
# Vertex Cover

Let's attempt to design a greedy solution first.

Recall, greedy solutions are often the most natural algorithms to design, but sometimes it's difficult to prove their correctness.

**Intuition** Greedily pick an edge, add its endpoints to the vertex cover, and remove edges adjacent to the endpoints.

i.e. Find a maximal matching in G and output the set of matched vertices.



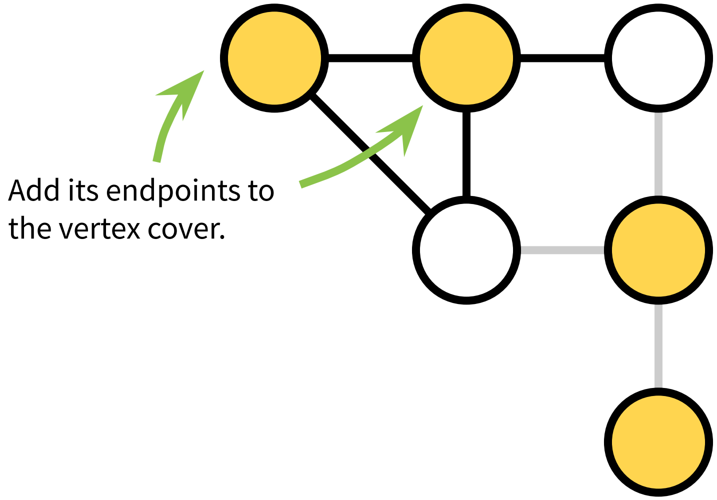
# Vertex Cover

Let’s attempt to design a greedy solution first.

Recall, greedy solutions are often the most natural algorithms to design, but sometimes it’s difficult to prove their correctness.

**Intuition** Greedily pick an edge, add its endpoints to the vertex cover, and remove edges adjacent to the endpoints.

i.e. Find a maximal matching in G and output the set of matched vertices.



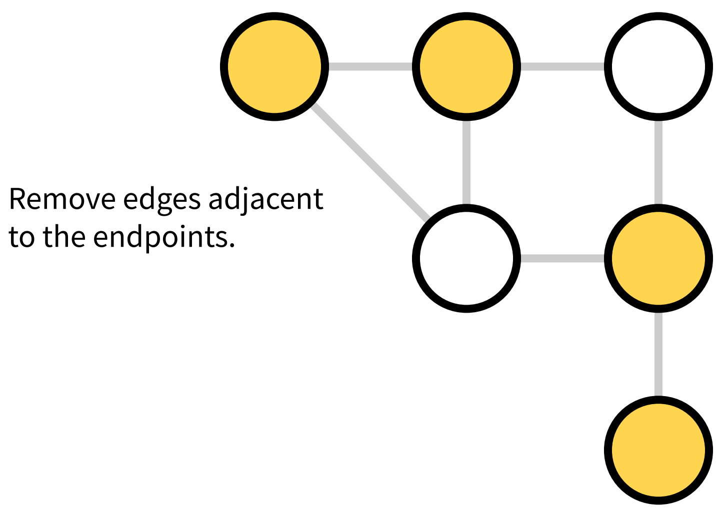
# Vertex Cover

Let's attempt to design a greedy solution first.

Recall, greedy solutions are often the most natural algorithms to design, but sometimes it's difficult to prove their correctness.

**Intuition** Greedily pick an edge, add its endpoints to the vertex cover, and remove edges adjacent to the endpoints.

i.e. Find a maximal matching in G and output the set of matched vertices.



# Vertex Cover

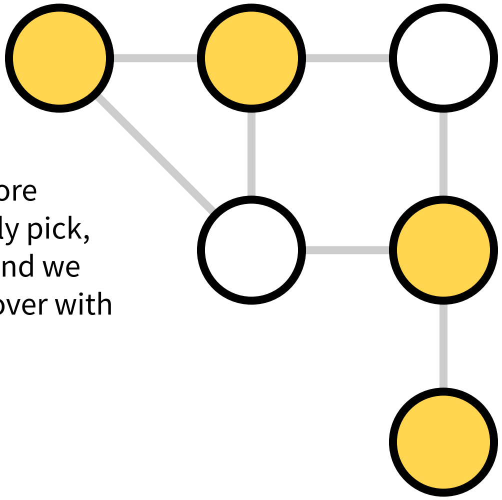
Let's attempt to design a greedy solution first.

Recall, greedy solutions are often the most natural algorithms to design, but sometimes it's difficult to prove their correctness.

**Intuition** Greedily pick an edge, add its endpoints to the vertex cover, and remove edges adjacent to the endpoints.

i.e. Find a maximal matching in G and output the set of matched vertices.

There are no more edges to greedily pick, so we're done and we have a vertex cover with cardinality 4!



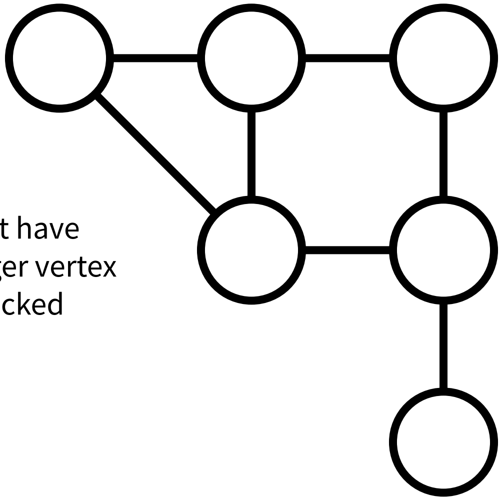
# Vertex Cover

Let's attempt to design a greedy solution first.

Recall, greedy solutions are often the most natural algorithms to design, but sometimes it's difficult to prove their correctness.

**Intuition** Greedily pick an edge, add its endpoints to the vertex cover, and remove edges adjacent to the endpoints.

i.e. Find a maximal matching in G and output the set of matched vertices.



Obviously, this algorithm might have produced a larger vertex cover had we picked different edges.

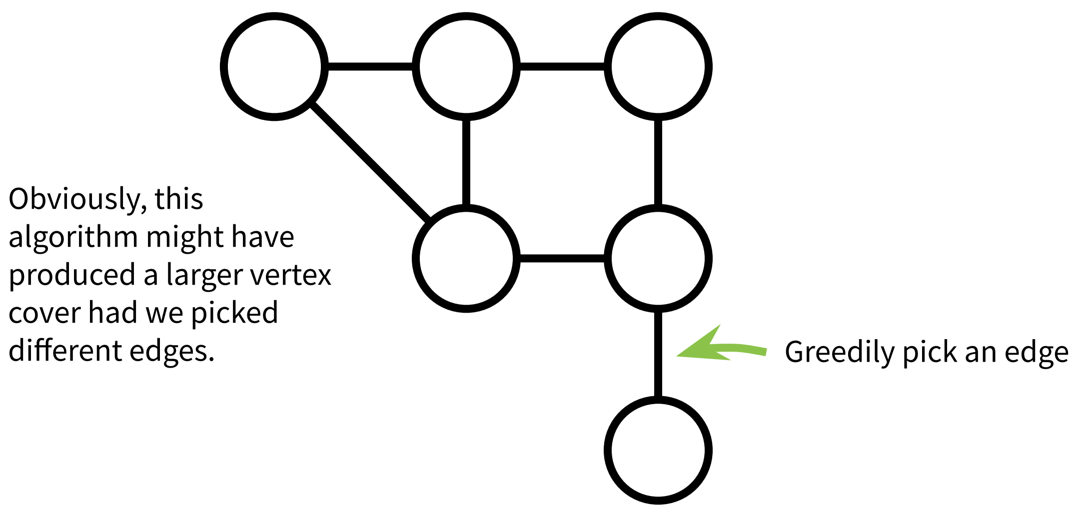
# Vertex Cover

Let's attempt to design a greedy solution first.

Recall, greedy solutions are often the most natural algorithms to design, but sometimes it's difficult to prove their correctness.

**Intuition** Greedily pick an edge, add its endpoints to the vertex cover, and remove edges adjacent to the endpoints.

i.e. Find a maximal matching in G and output the set of matched vertices.



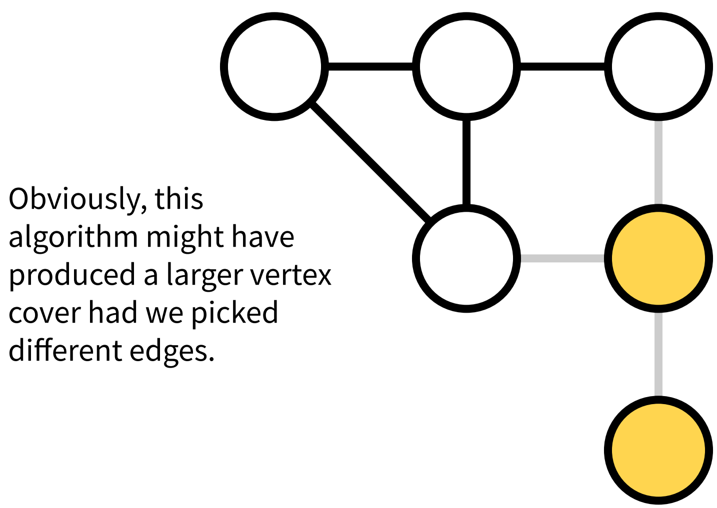
# Vertex Cover

Let’s attempt to design a greedy solution first.

Recall, greedy solutions are often the most natural algorithms to design, but sometimes it’s difficult to prove their correctness.

**Intuition** Greedily pick an edge, add its endpoints to the vertex cover, and remove edges adjacent to the endpoints.

i.e. Find a maximal matching in G and output the set of matched vertices.



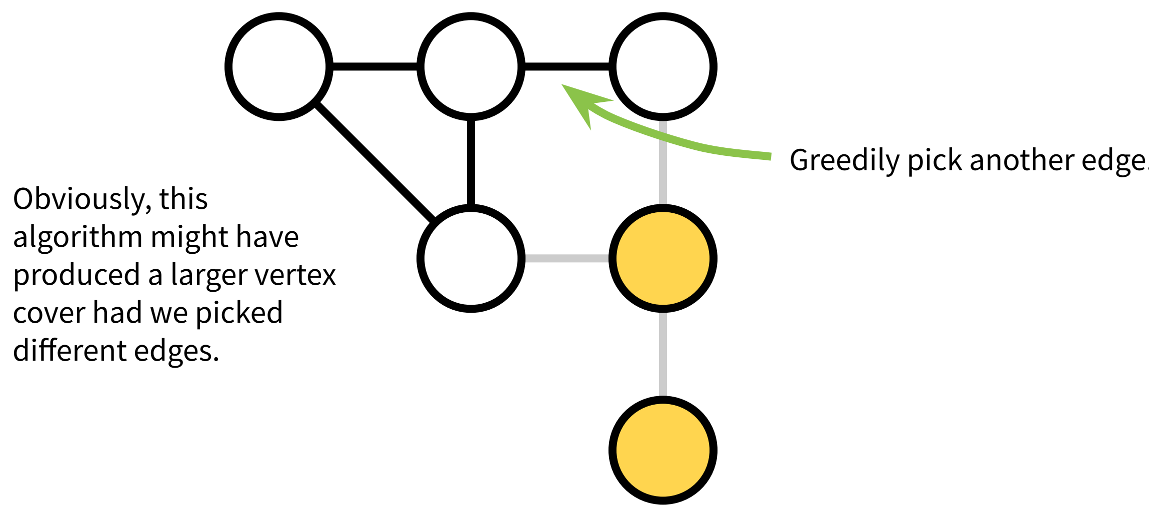
# Vertex Cover

Let's attempt to design a greedy solution first.

Recall, greedy solutions are often the most natural algorithms to design, but sometimes it's difficult to prove their correctness.

**Intuition** Greedily pick an edge, add its endpoints to the vertex cover, and remove edges adjacent to the endpoints.

i.e. Find a maximal matching in G and output the set of matched vertices.



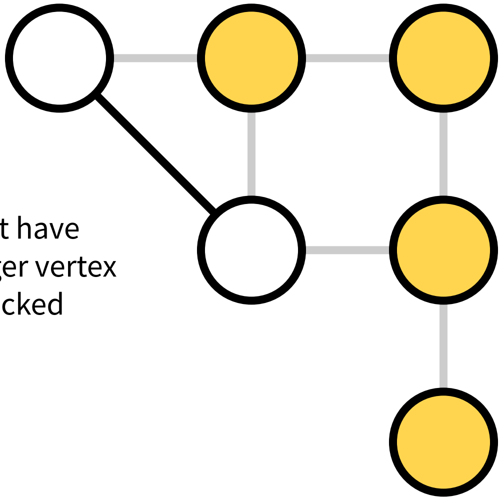
# Vertex Cover

Let's attempt to design a greedy solution first.

Recall, greedy solutions are often the most natural algorithms to design, but sometimes it's difficult to prove their correctness.

**Intuition** Greedily pick an edge, add its endpoints to the vertex cover, and remove edges adjacent to the endpoints.

i.e. Find a maximal matching in G and output the set of matched vertices.



Obviously, this algorithm might have produced a larger vertex cover had we picked different edges.

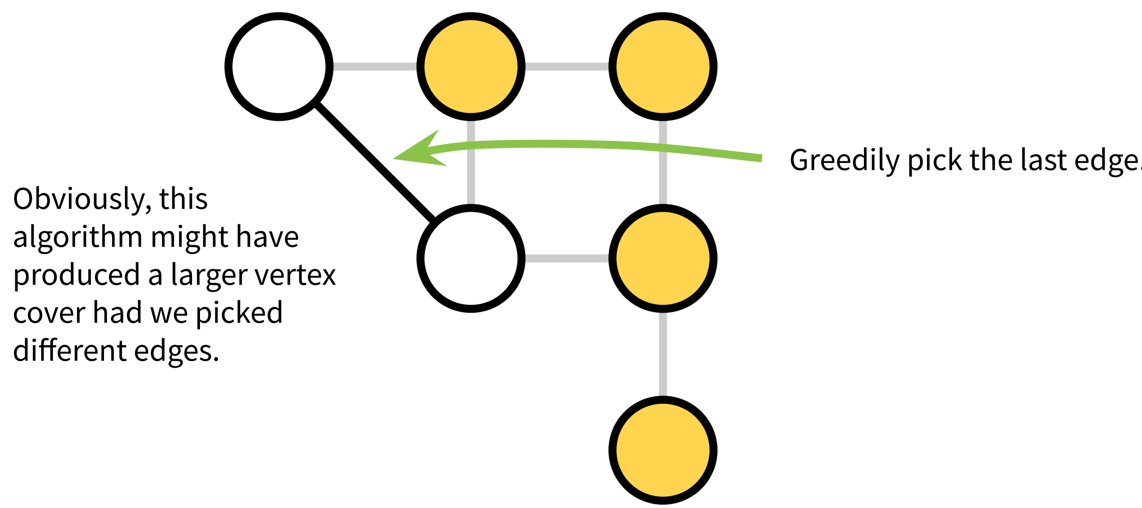
# Vertex Cover

Let's attempt to design a greedy solution first.

Recall, greedy solutions are often the most natural algorithms to design, but sometimes it's difficult to prove their correctness.

**Intuition** Greedily pick an edge, add its endpoints to the vertex cover, and remove edges adjacent to the endpoints.

i.e. Find a maximal matching in G and output the set of matched vertices.



# Vertex Cover

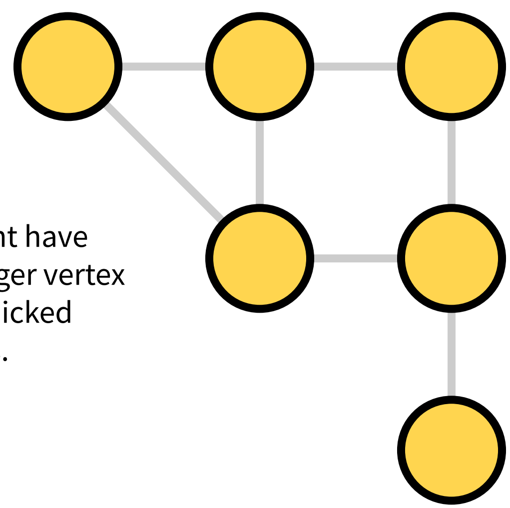
Let's attempt to design a greedy solution first.

Recall, greedy solutions are often the most natural algorithms to design, but sometimes it's difficult to prove their correctness.

**Intuition** Greedily pick an edge, add its endpoints to the vertex cover, and remove edges adjacent to the endpoints.

i.e. Find a maximal matching in G and output the set of matched vertices.

Obviously, this algorithm might have produced a larger vertex cover had we picked different edges.



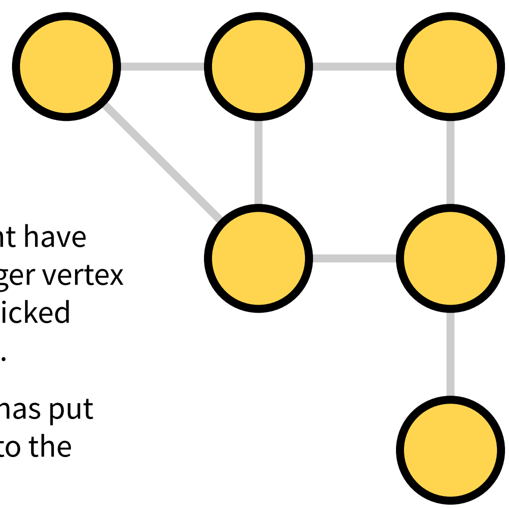
# Vertex Cover

Let's attempt to design a greedy solution first.

Recall, greedy solutions are often the most natural algorithms to design, but sometimes it's difficult to prove their correctness.

**Intuition** Greedily pick an edge, add its endpoints to the vertex cover, and remove edges adjacent to the endpoints.

i.e. Find a maximal matching in G and output the set of matched vertices.



Obviously, this algorithm might have produced a larger vertex cover had we picked different edges.

The algorithm has put every vertex into the vertex cover!

# Vertex Cover

```
def find_vertex_cover(G):
  matching, vc = find_maximal_matching(G)
  return vc

def find_maximal_matching(G):
  matching = {}  # a set of edges
  matched_vertices = {}  # a set of vertices
  ce = {}  # stands for "covered edges"
  while ce ≠ G.E:
    e = pick an edge from G.E - ce at random
    add e to matching
    add e.v1 and e.v2 to matched_vertices
    add edges adjacent to v1, v2 to ce
  return matching, matched_vertices
```

**Runtime:** $O(|V|+|E|)$

# Vertex Cover

(1) How do we establish an approximation guarantee for our algorithm?

**Theorem:** `find_vertex_cover` is a factor 2-approximation algorithm for the cardinality vertex cover problem.

**Proof:**

# Vertex Cover

## (1) How do we establish an approximation guarantee for our algorithm?

**Theorem:** `find_vertex_cover` is a factor 2-approximation algorithm for the cardinality vertex cover problem.

**Proof:**

Let OPT be the optimal cardinality vertex cover in G. Notice that the size of a maximal matching M in G provides a lower bound for OPT since any vertex cover has to pick at least one endpoint of each matched edge. Thus, $|M| \leq \text{OPT}$.

# Vertex Cover

(1) How do we establish an approximation guarantee for our algorithm?

**Theorem:** `find_vertex_cover` is a factor 2-approximation algorithm for the cardinality vertex cover problem.

**Proof:**

Let OPT be the optimal cardinality vertex cover in G. Notice that the size of a maximal matching M in G provides a lower bound for OPT since any vertex cover has to pick at least one endpoint of each matched edge. Thus, $|M| \le OPT$.

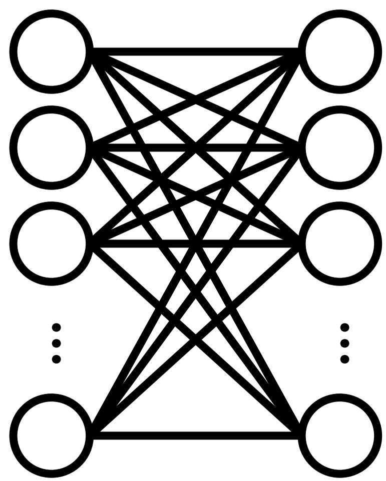
The algorithm picks a cover with cardinality $2 \cdot |M|$, which is at most $2 \cdot OPT$. ■

# Vertex Cover

(2) Can the approximation guarantee of `find_vertex_cover` be improved by a better analysis?

No, our analysis is tight. Consider an infinite complete bipartite graph, called a **tight example** since the approximation guarantee is tight

i.e. `find_vertex_cover` produces a solution twice the optimal.



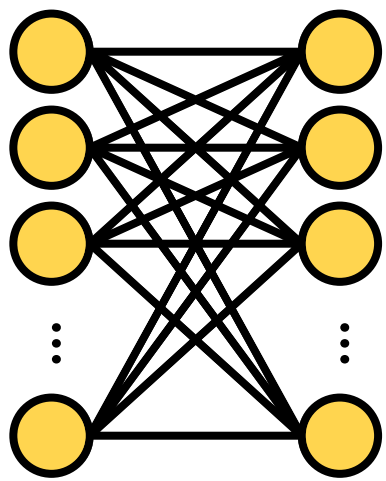
# Vertex Cover

(2) Can the approximation guarantee of `find_vertex_cover` be improved by a better analysis?

No, our analysis is tight. Consider an infinite complete bipartite graph, called a **tight example** since the approximation guarantee is tight

i.e. `find_vertex_cover` produces a solution twice the optimal.

In general, we will look for these tight examples to prove the tightness of our approximation.



`find_vertex_cover` will pick all 2n vertices.

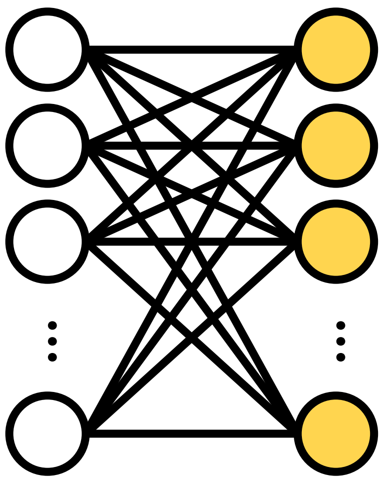
# Vertex Cover

(2) Can the approximation guarantee of `find_vertex_cover` be improved by a better analysis?

No, our analysis is tight. Consider an infinite complete bipartite graph, called a **tight example** since the approximation guarantee is tight

i.e. `find_vertex_cover` produces a solution twice the optimal.

In general, we will look for these tight examples to prove the tightness of our approximation.



`find_vertex_cover` will pick all 2n vertices.

But picking one side of the bipartition gives a cover of size n.

# Vertex Cover

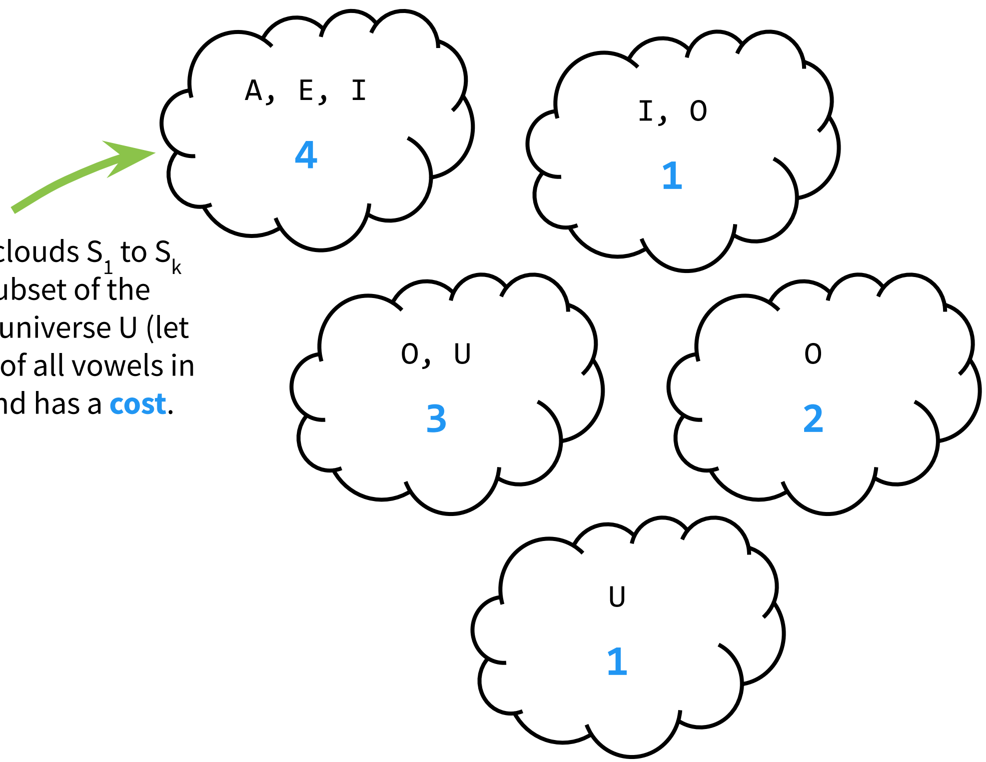
(3) Is there some other lower bounding method that can lead to an improved approximation guarantee for vertex cover?

**This is an open problem!**

# Set Cover

# Set Cover

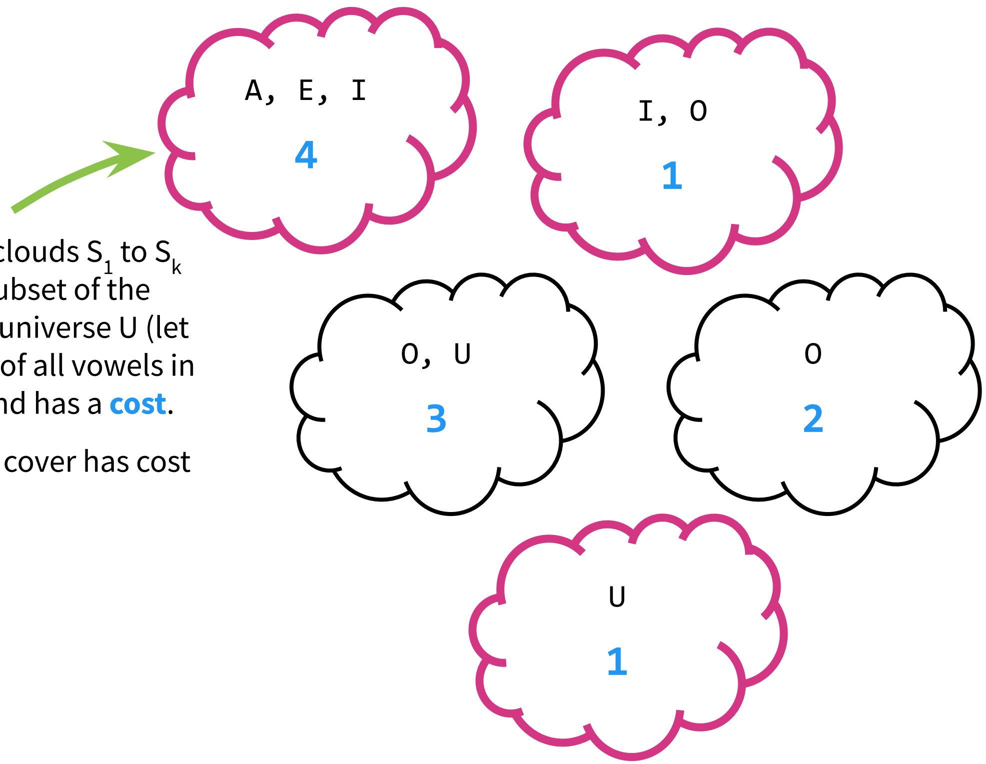
**Task** Given a universe U of n elements, a collection of subsets $S = \{S_1, \ldots, S_k\}$ of U each of which has a cost, find a min cost subcollection of S that covers all elements of U.




Each of the clouds $S_1$ to $S_k$ contains a subset of the elements in universe U (let U be the list of all vowels in this case), and has a **cost**.

# Set Cover

**Task** Given a universe U of n elements, a collection of subsets $S = \{S_1, \ldots, S_k\}$ of U each of which has a cost, find a min cost subcollection of S that covers all elements of U.




Each of the clouds $S_1$ to $S_k$ contains a subset of the elements in universe U (let U be the list of all vowels in this case), and has a **cost**.

The optimal cover has cost $4 + 1 + 1 = 6$.

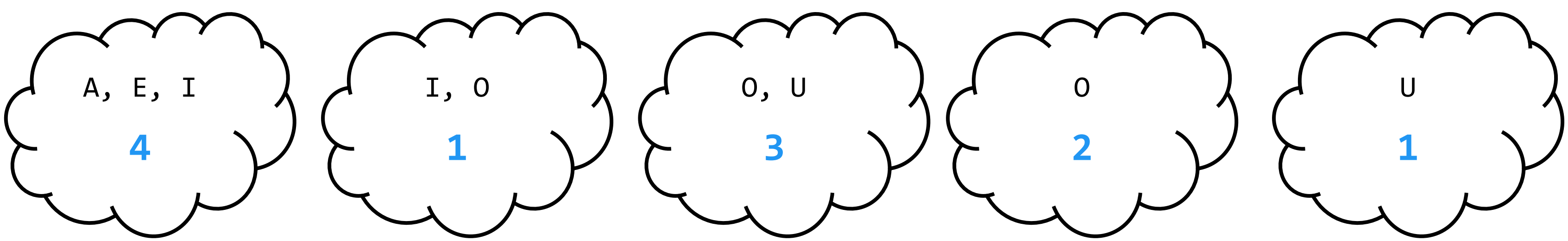
# Set Cover

Let's attempt to design a greedy solution first.

Recall, greedy solutions are often the most natural algorithms to design, but sometimes it's difficult to prove their correctness.

**Intuition** Iteratively pick the most "cost-effective" set and remove the covered elements until all elements are covered.

Let the **cost-effectiveness score** of a set $S_i$ be the average cost at which It covers new elements i.e. $\text{cost}(S_i) / |S_i - C|$ where C is the set of elements covered already.



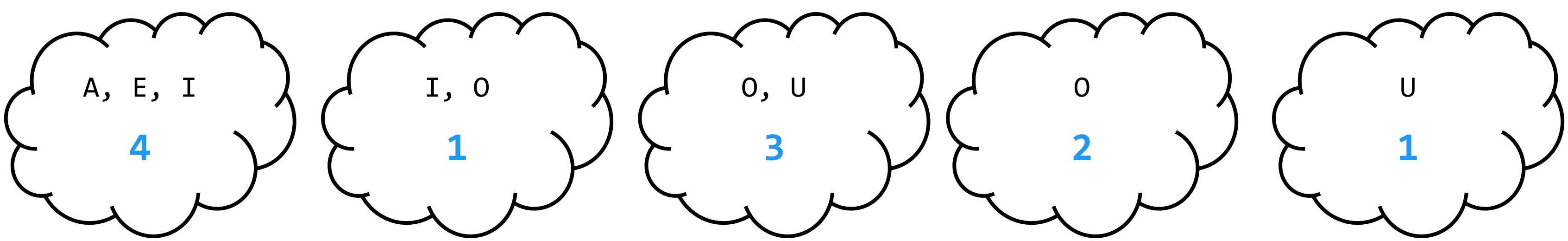
# Set Cover

Let's attempt to design a greedy solution first.

Recall, greedy solutions are often the most natural algorithms to design, but sometimes it's difficult to prove their correctness.

**Intuition** Iteratively pick the most "cost-effective" set and remove the covered elements until all elements are covered.

Let the **cost-effectiveness score** of a set $S_i$ be the average cost at which It covers new elements i.e. cost($S_i$) / $|S_i - C|$ where C is the set of elements covered already.




**4** / |{A, E, I}| = **4/3** **1** / |{I, O}| = **1/2** **3** / |{O, U}| = **3/2** **2** / |{O}| = **2** **1** / |{U}| = **1**

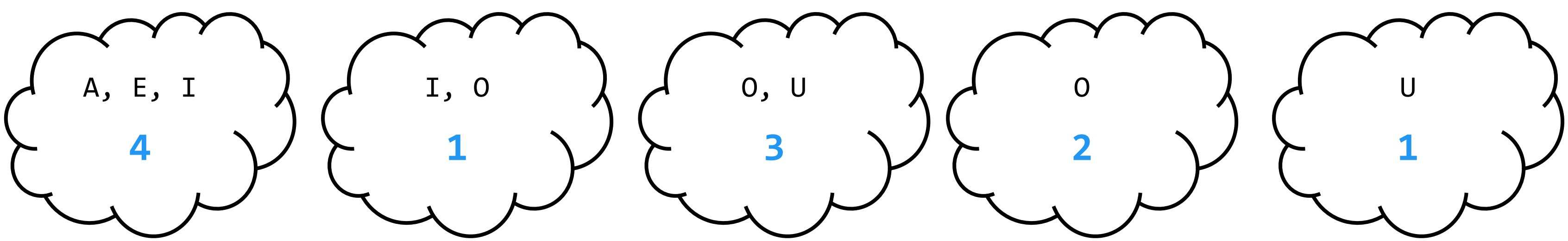
# Set Cover

## Let's attempt to design a greedy solution first.

Recall, greedy solutions are often the most natural algorithms to design, but sometimes it's difficult to prove their correctness.

## **Intuition** Iteratively pick the most "cost-effective" set and remove the covered elements until all elements are covered.

Let the **cost-effectiveness score** of a set $S_i$ be the average cost at which It covers new elements i.e. $\text{cost}(S_i) / |S_i - C|$ where C is the set of elements covered already.




$4 / |\{A, E, I\}| = 4/3$ $\quad$ $1 / |\{I, O\}| = 1/2$ $\quad$ $3 / |\{O, U\}| = 3/2$ $\quad$ $2 / |\{O\}| = 2$ $\quad$ $1 / |\{U\}| = 1$

The "most cost-effective" set is the one with lowest cost-effectiveness score.

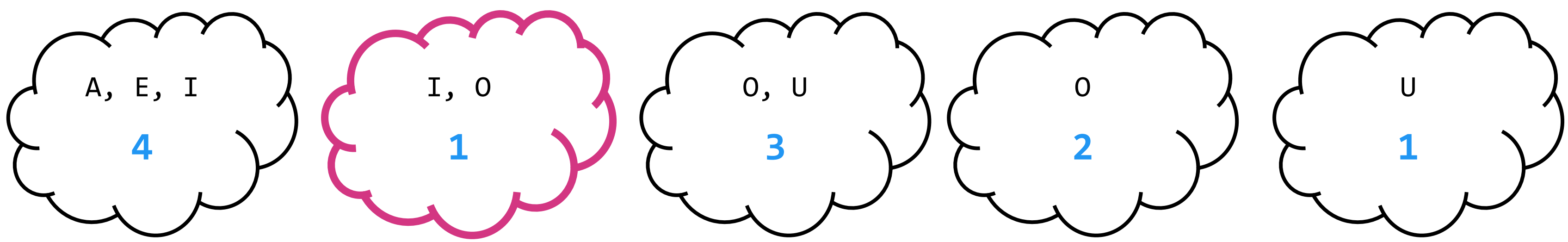
# Set Cover

## Let's attempt to design a greedy solution first.

Recall, greedy solutions are often the most natural algorithms to design, but sometimes it's difficult to prove their correctness.

**Intuition** Iteratively pick the most "cost-effective" set and remove the covered elements until all elements are covered.

Let the **cost-effectiveness score** of a set $S_i$ be the average cost at which It covers new elements i.e. $\text{cost}(S_i) / |S_i - C|$ where C is the set of elements covered already.




$4 / |\{A, E, I\}| = 4/3$ $\quad$ $1 / |\{I, O\}| = 1/2$ $\quad$ $3 / |\{O, U\}| = 3/2$ $\quad$ $2 / |\{O\}| = 2$ $\quad$ $1 / |\{U\}| = 1$

The "most cost-effective" set is the one with lowest cost-effectiveness score.

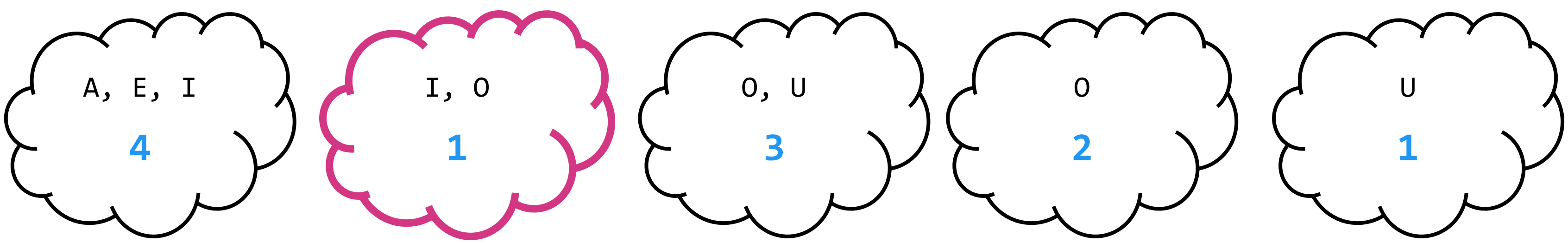
# Set Cover

Let's attempt to design a greedy solution first.

Recall, greedy solutions are often the most natural algorithms to design, but sometimes it's difficult to prove their correctness.

**Intuition** Iteratively pick the most "cost-effective" set and remove the covered elements until all elements are covered.

Let the **cost-effectiveness score** of a set $S_i$ be the average cost at which It covers new elements i.e. $\text{cost}(S_i) / |S_i - C|$ where C is the set of elements covered already.




4 / |{A, E, I}| = 4/3

3 / |{O, U}| = 3/2

2 / |{O}| = 2

1 / |{U}| = 1

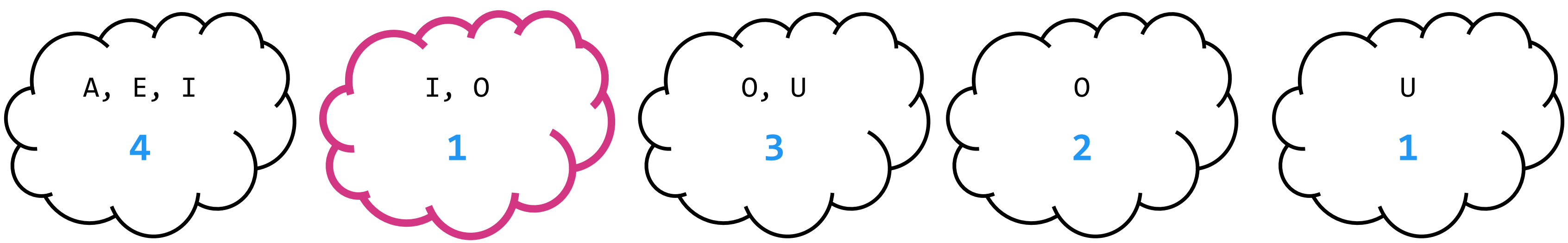
# Set Cover

Let's attempt to design a greedy solution first.

Recall, greedy solutions are often the most natural algorithms to design, but sometimes it's difficult to prove their correctness.

**Intuition** Iteratively pick the most "cost-effective" set and remove the covered elements until all elements are covered.

Let the **cost-effectiveness score** of a set $S_i$ be the average cost at which It covers new elements i.e. $\text{cost}(S_i) / |S_i - C|$ where C is the set of elements covered already.



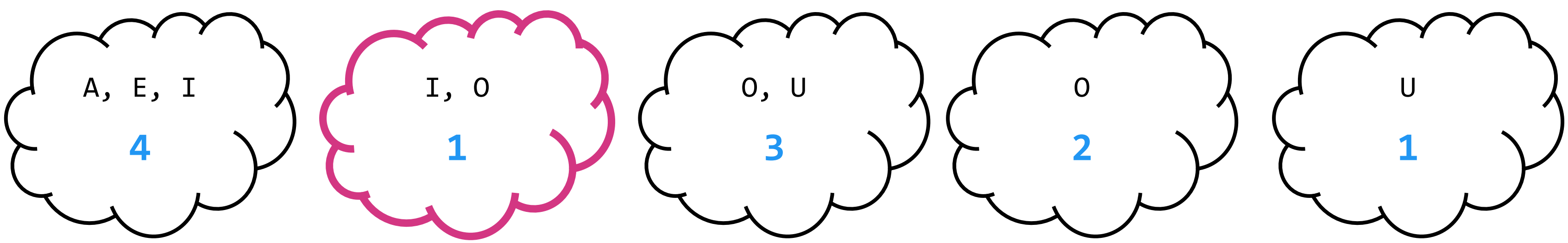
# Set Cover

Let's attempt to design a greedy solution first.

Recall, greedy solutions are often the most natural algorithms to design, but sometimes it's difficult to prove their correctness.

**Intuition** Iteratively pick the most "cost-effective" set and remove the covered elements until all elements are covered.

Let the **cost-effectiveness score** of a set $S_i$ be the average cost at which It covers new elements i.e. $\text{cost}(S_i) / |S_i - C|$ where C is the set of elements covered already.




4 / |{A, E, I} - {I, O}| = 4/2 = 2

3 / |{O, U} - {I, O}| = 3

2 / |{O} - {I, O}| = 2/0 = ∞

1 / |{U} - {I, O}| = 1

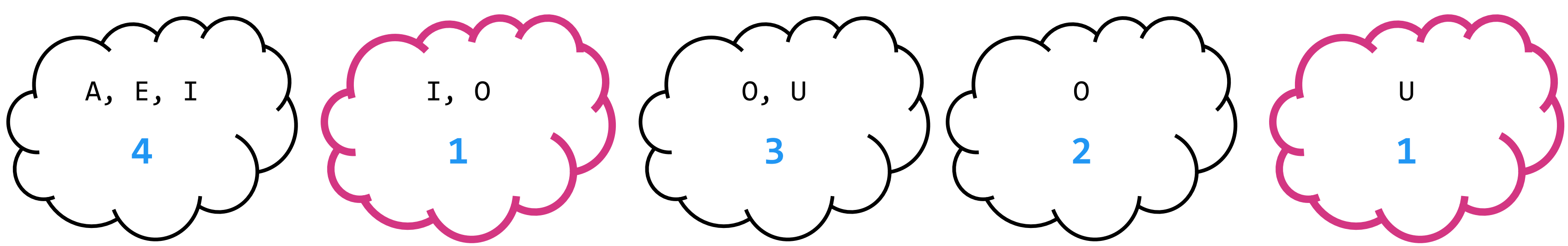
# Set Cover

Let's attempt to design a greedy solution first.

Recall, greedy solutions are often the most natural algorithms to design, but sometimes it's difficult to prove their correctness.

**Intuition** Iteratively pick the most "cost-effective" set and remove the covered elements until all elements are covered.

Let the **cost-effectiveness score** of a set $S_i$ be the average cost at which It covers new elements i.e. $\text{cost}(S_i) / |S_i - C|$ where C is the set of elements covered already.




$4 / |\{A, E, I\} - \{I, O\}| = 4/2 = 2$

$3 / |\{O, U\} - \{I, O\}| = 3$

$2 / |\{O\} - \{I, O\}| = 2/0 = \infty$

$1 / |\{U\} - \{I, O\}| = 1$

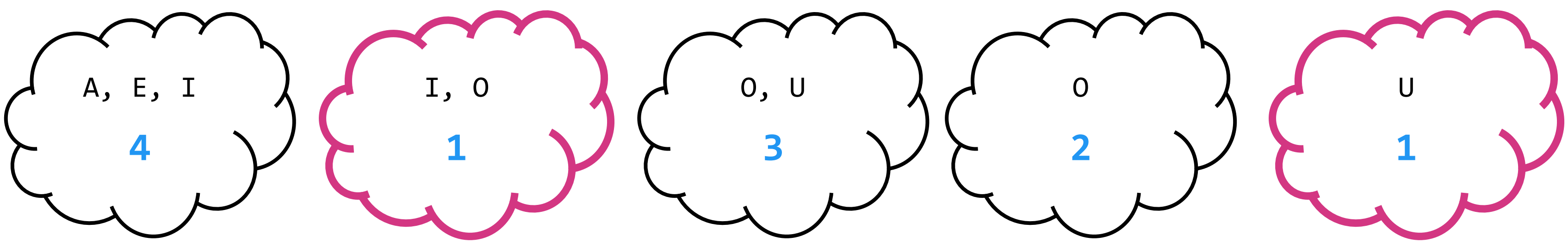
# Set Cover

Let's attempt to design a greedy solution first.

Recall, greedy solutions are often the most natural algorithms to design, but sometimes it's difficult to prove their correctness.

**Intuition** Iteratively pick the most "cost-effective" set and remove the covered elements until all elements are covered.

Let the **cost-effectiveness score** of a set $S_i$ be the average cost at which It covers new elements i.e. cost($S_i$) / $|S_i - C|$ where C is the set of elements covered already.




$4 / |\{A, E, I\} - \{I, O\}| = 4/2 = 2$

$3 / |\{O, U\} - \{I, O\}| = 3$

$2 / |\{O\} - \{I, O\}| = 2/0 = \infty$

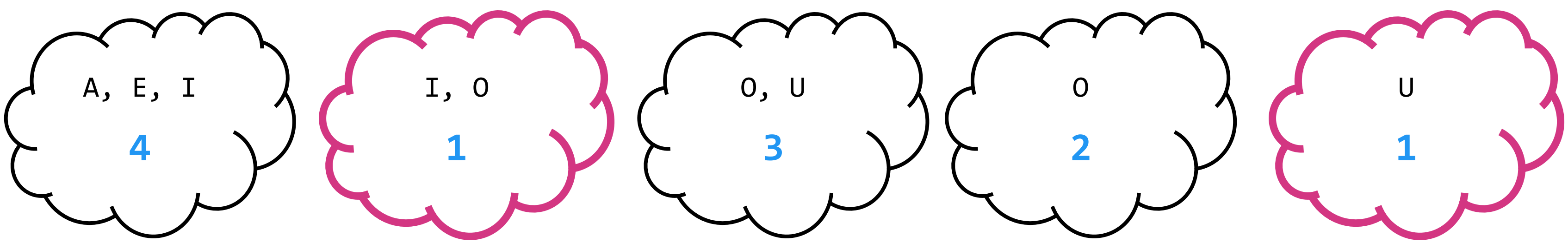
# Set Cover

Let's attempt to design a greedy solution first.

Recall, greedy solutions are often the most natural algorithms to design, but sometimes it's difficult to prove their correctness.

**Intuition** Iteratively pick the most "cost-effective" set and remove the covered elements until all elements are covered.

Let the **cost-effectiveness score** of a set $S_i$ be the average cost at which It covers new elements i.e. $\text{cost}(S_i) / |S_i - C|$ where C is the set of elements covered already.



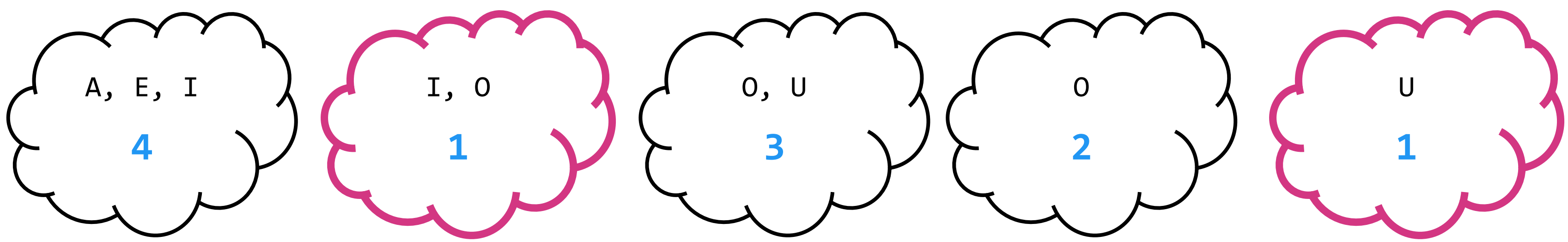
# Set Cover

## Let's attempt to design a greedy solution first.

Recall, greedy solutions are often the most natural algorithms to design, but sometimes it's difficult to prove their correctness.

**Intuition** Iteratively pick the most "cost-effective" set and remove the covered elements until all elements are covered.

Let the **cost-effectiveness score** of a set $S_i$ be the average cost at which It covers new elements i.e. $\text{cost}(S_i) / |S_i - C|$ where C is the set of elements covered already.




$4 / |\{A, E, I\} - \{I, O, U\}| = 4/2 = 2$

$3 / |\{O, U\} - \{I, O, U\}| = 3/0 = \infty$

$2 / |\{O\} - \{I, O, U\}| = 2/0 = \infty$

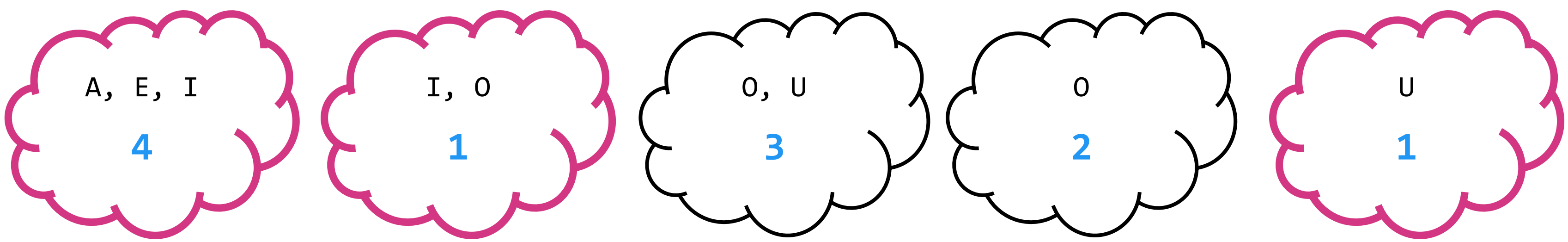
# Set Cover

## Let's attempt to design a greedy solution first.

Recall, greedy solutions are often the most natural algorithms to design, but sometimes it's difficult to prove their correctness.

**Intuition** Iteratively pick the most "cost-effective" set and remove the covered elements until all elements are covered.

Let the **cost-effectiveness score** of a set $S_i$ be the average cost at which It covers new elements i.e. $\text{cost}(S_i) / |S_i - C|$ where C is the set of elements covered already.




$4 / |\{A, E, I\} - \{I, O, U\}| = 4/2 = 2$

$3 / |\{O, U\} - \{I, O, U\}| = 3/0 = \infty$

$2 / |\{O\} - \{I, O, U\}| = 2/0 = \infty$

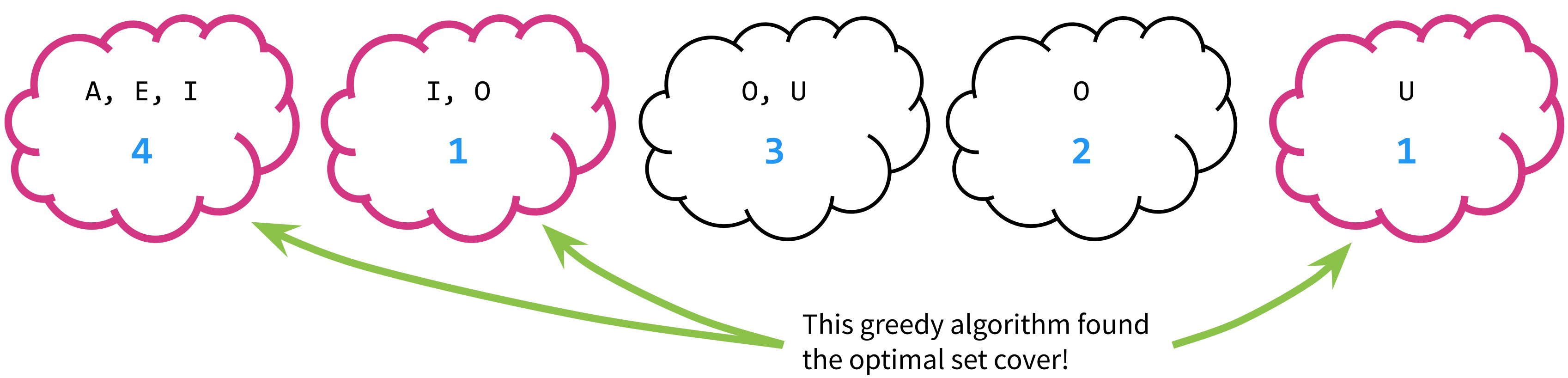
# Set Cover

## Let's attempt to design a greedy solution first.

Recall, greedy solutions are often the most natural algorithms to design, but sometimes it's difficult to prove their correctness.

**Intuition** Iteratively pick the most "cost-effective" set and remove the covered elements until all elements are covered.

Let the **cost-effectiveness score** of a set $S_i$ be the average cost at which It covers new elements i.e. $\text{cost}(S_i) / |S_i - C|$ where C is the set of elements covered already.



# Set Cover

```
def find_set_cover(S):
  C = {}  # stands for "covered"
  sc = {}
  prices = dict()
  while C ≠ U:
    # α is the cost effectiveness of the most
    # cost-effective set, S_i
    S_i, α = get_most_cost_effective(S, C)
    sc.add(S_i)
    prices[e] = α for e in S_i - C
    C = C ∪ S_i
  return sc
```

i.e. The elements in $S_i$ that haven't been covered by a previous set.

**Runtime:** $O(|S|^2)$

We can improve this runtime by using fancy data structures.

# Set Cover

## (1) How do we establish an approximation guarantee for our algorithm?

Number the elements in U in the order in which they were covered by the algorithm, resolving ties arbitrarily. Let $e_1, \ldots, e_n$ be this numbering. Note $n = |U|$.

**Lemma:** For each $k \in \{1, \ldots, n\}$, $\text{price}[e_k] \leq \text{OPT} / (n - k + 1)$.

**Proof:**

# Set Cover

## (1) How do we establish an approximation guarantee for our algorithm?

Number the elements in U in the order in which they were covered by the algorithm, resolving ties arbitrarily. Let $e_1, \ldots, e_n$ be this numbering. Note $n = |U|$.

**Lemma:** For each $k \in \{1, \ldots, n\}$, $\text{price}[e_k] \leq \text{OPT} / (n - k + 1)$.

**Proof:**

In any iteration, the leftover sets of the optimal solution can cover the remaining elements at a cost of at most OPT.

# Set Cover

## (1) How do we establish an approximation guarantee for our algorithm?

Number the elements in U in the order in which they were covered by the algorithm, resolving ties arbitrarily. Let $e_1, \ldots, e_n$ be this numbering. Note $n = |U|$.

**Lemma:** For each $k \in \{1, \ldots, n\}$, $\text{price}[e_k] \leq \text{OPT} / (n - k + 1)$.

**Proof:**

In any iteration, the leftover sets of the optimal solution can cover the remaining elements at a cost of at most OPT. Therefore, among these sets, there must be one having cost-effectiveness of at most $\text{OPT}/|C^C|$.

# Set Cover

(1) How do we establish an approximation guarantee for our algorithm?

Number the elements in U in the order in which they were covered by the algorithm, resolving ties arbitrarily. Let $e_1, \ldots, e_n$ be this numbering. Note $n = |U|$.

**Lemma:** For each $k \in \{1, \ldots, n\}$, $\text{price}[e_k] \leq \text{OPT} / (n - k + 1)$.

**Proof:**

In any iteration, the leftover sets of the optimal solution can cover the remaining elements at a cost of at most OPT. Therefore, among these sets, there must be one having cost-effectiveness of at most $\text{OPT}/|C^C|$.

In the iteration in which element $e_k$ was covered, $C^C$ contained at least $n - k + 1$ elements.

# Set Cover

(1) How do we establish an approximation guarantee for our algorithm?

Number the elements in U in the order in which they were covered by the algorithm, resolving ties arbitrarily. Let $e_1, \ldots, e_n$ be this numbering. Note $n = |U|$.

**Lemma:** For each $k \in \{1, \ldots, n\}$, $\text{price}[e_k] \leq \text{OPT} / (n - k + 1)$.

**Proof:**

In any iteration, the leftover sets of the optimal solution can cover the remaining elements at a cost of at most OPT. Therefore, among these sets, there must be one having cost-effectiveness of at most $\text{OPT}/|C^C|$.

In the iteration in which element $e_k$ was covered, $C^C$ contained at least $n - k + 1$ elements. Since $e_k$ was covered by the most cost-effective set in this iteration, it follows that $\text{price}[e_k] \leq \text{OPT}/|C^C| \leq \text{OPT} / (n - k + 1)$. ■

# Set Cover

## (1) How do we establish an approximation guarantee for our algorithm?

Number the elements in U in the order in which they were covered by the algorithm, resolving ties arbitrarily. Let $e_1, \ldots, e_n$ be this numbering. Note $n = |U|$.

**Lemma:** For each $k \in \{1, \ldots, n\}$, $\text{price}[e_k] \leq \text{OPT} / (n - k + 1)$.

**Theorem:** `find_set_cover` is a factor ε-approximation algorithm for the set cover problem, where $\varepsilon = H_n$ i.e. the nth Harmonic number.

**Proof:**

# Set Cover

## (1) How do we establish an approximation guarantee for our algorithm?

Number the elements in U in the order in which they were covered by the algorithm, resolving ties arbitrarily. Let $e_1, \ldots, e_n$ be this numbering. Note $n = |U|$.

**Lemma:** For each $k \in \{1, \ldots, n\}$, $\text{price}[e_k] \leq \text{OPT} / (n - k + 1)$.

**Theorem:** `find_set_cover` is a factor ε-approximation algorithm for the set cover problem, where $\varepsilon = H_n$ i.e. the nth Harmonic number.

**Proof:**

Since the cost of each set picked is distributed among the new elements covered, the total cost of the set cover picked is equal to $\sum_k \text{price}[e_k]$.

# Set Cover

## (1) How do we establish an approximation guarantee for our algorithm?

Number the elements in U in the order in which they were covered by the algorithm, resolving ties arbitrarily. Let $e_1, \ldots, e_n$ be this numbering. Note $n = |U|$.

**Lemma:** For each $k \in \{1, \ldots, n\}$, $\text{price}[e_k] \leq \text{OPT} / (n - k + 1)$.

**Theorem:** `find_set_cover` is a factor ε-approximation algorithm for the set cover problem, where $\varepsilon = H_n$ i.e. the nth Harmonic number.

**Proof:**

Since the cost of each set picked is distributed among the new elements covered, the total cost of the set cover picked is equal to $\sum_k \text{price}[e_k]$.

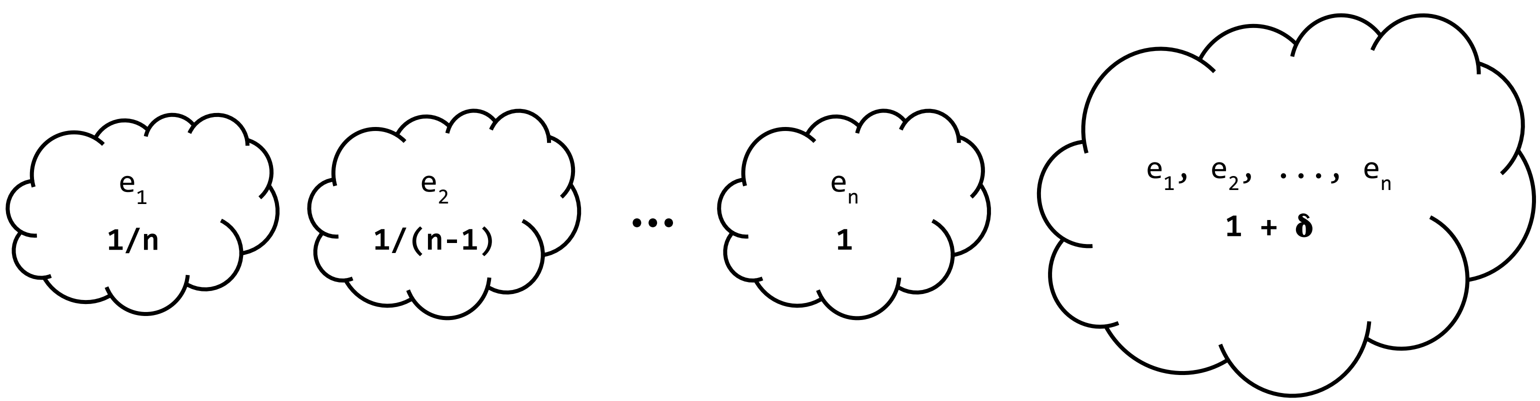
By the lemma, $\sum_k \text{price}[e_k] \leq (1 + 1/2 + \ldots + 1/n) \cdot \text{OPT} \leq \log(n) \cdot \text{OPT}$. ■

# Set Cover

(2) Can the approximation guarantee of `find_set_cover` be improved by a better analysis?

No. Let's find a **tight example** to convince ourselves that our approximation guarantee is tight i.e. `find_set_cover` produces a solution log(n) times the optimal.

Consider a set of singleton sets and one exhaustive set, each with specific costs.

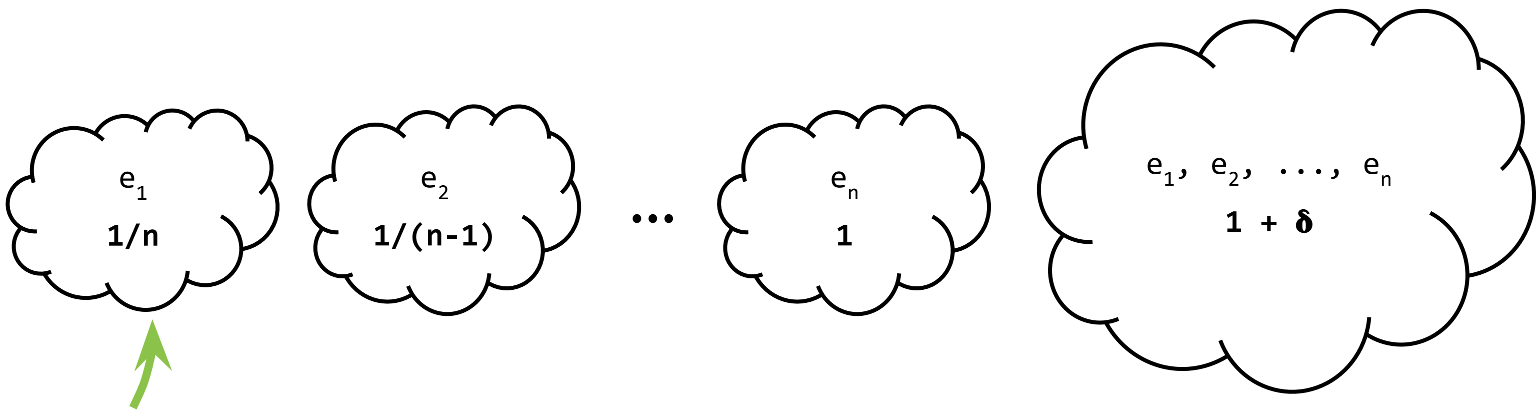


# Set Cover

## (2) Can the approximation guarantee of `find_set_cover` be improved by a better analysis?

No. Let's find a **tight example** to convince ourselves that our approximation guarantee is tight i.e. `find_set_cover` produces a solution log(n) times the optimal.

Consider a set of singleton sets and one exhaustive set, each with specific costs.



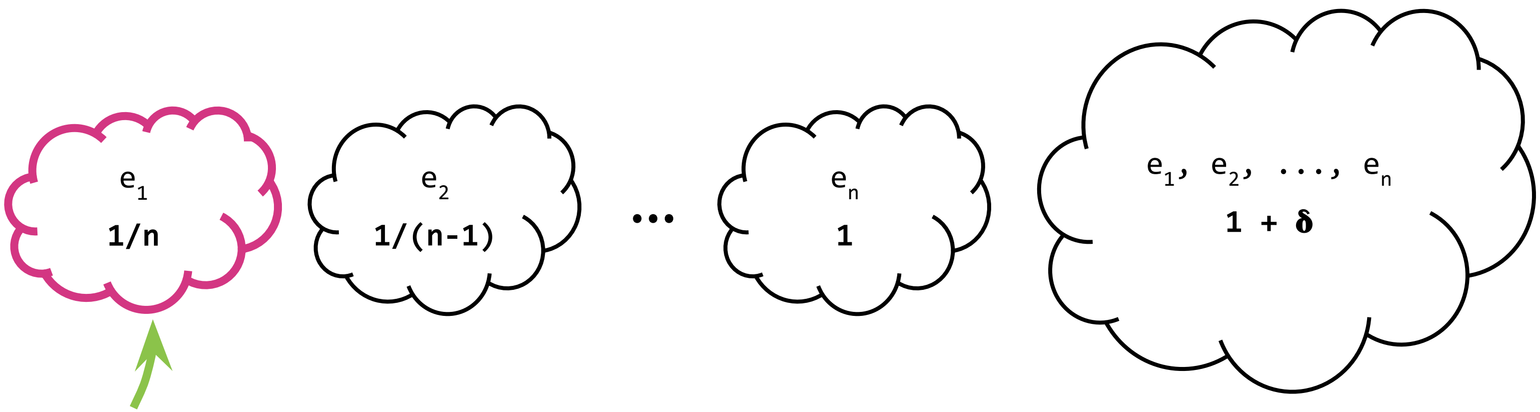

cost($e_1$) = 1/n, so the cost-effectiveness of this set is 1/n/1 = 1/n.

# Set Cover

(2) Can the approximation guarantee of `find_set_cover` be improved by a better analysis?

No. Let's find a **tight example** to convince ourselves that our approximation guarantee is tight i.e. `find_set_cover` produces a solution log(n) times the optimal.

Consider a set of singleton sets and one exhaustive set, each with specific costs.



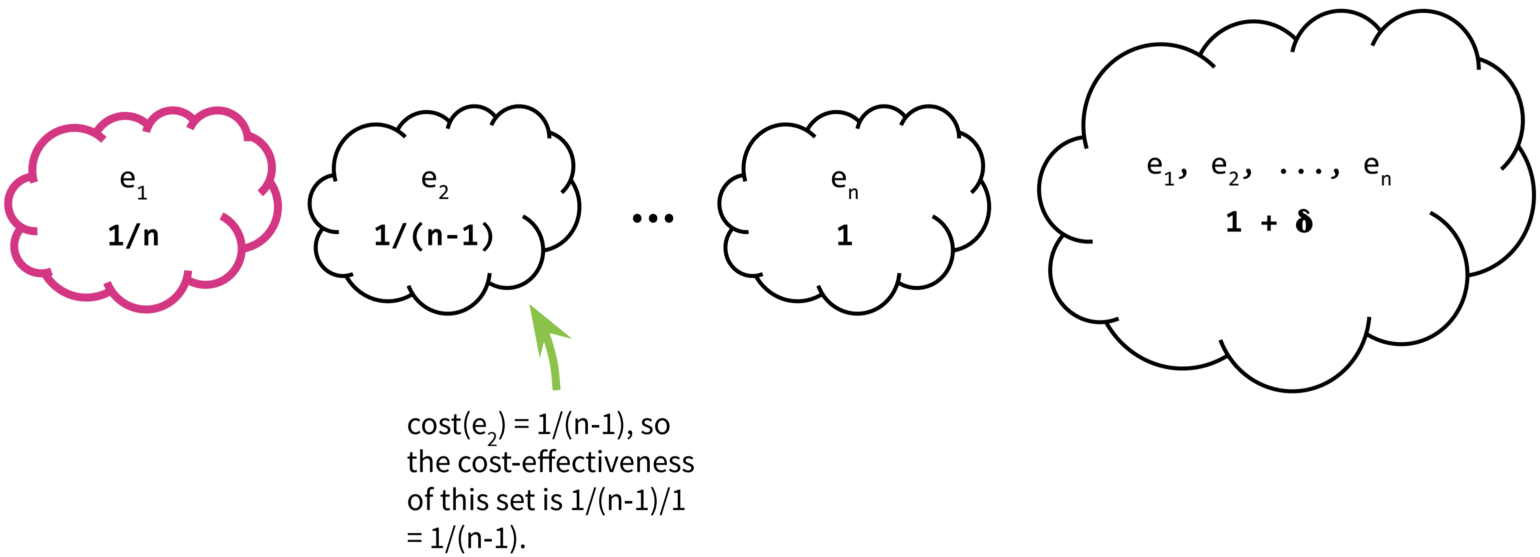

cost($e_1$) = 1/n, so the cost-effectiveness of this set is 1/n/1 = 1/n.

# Set Cover

(2) Can the approximation guarantee of `find_set_cover` be improved by a better analysis?

No. Let's find a **tight example** to convince ourselves that our approximation guarantee is tight i.e. `find_set_cover` produces a solution log(n) times the optimal.

Consider a set of singleton sets and one exhaustive set, each with specific costs.

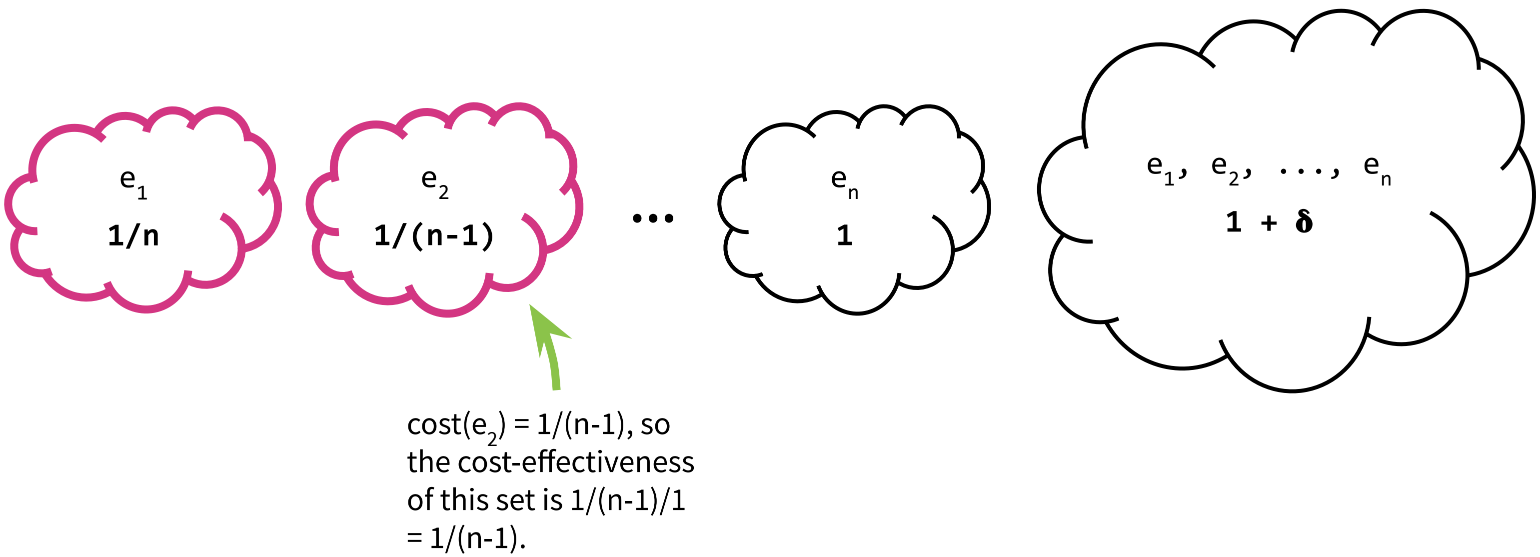


# Set Cover

(2) Can the approximation guarantee of `find_set_cover` be improved by a better analysis?

No. Let's find a **tight example** to convince ourselves that our approximation guarantee is tight i.e. `find_set_cover` produces a solution log(n) times the optimal.

Consider a set of singleton sets and one exhaustive set, each with specific costs.

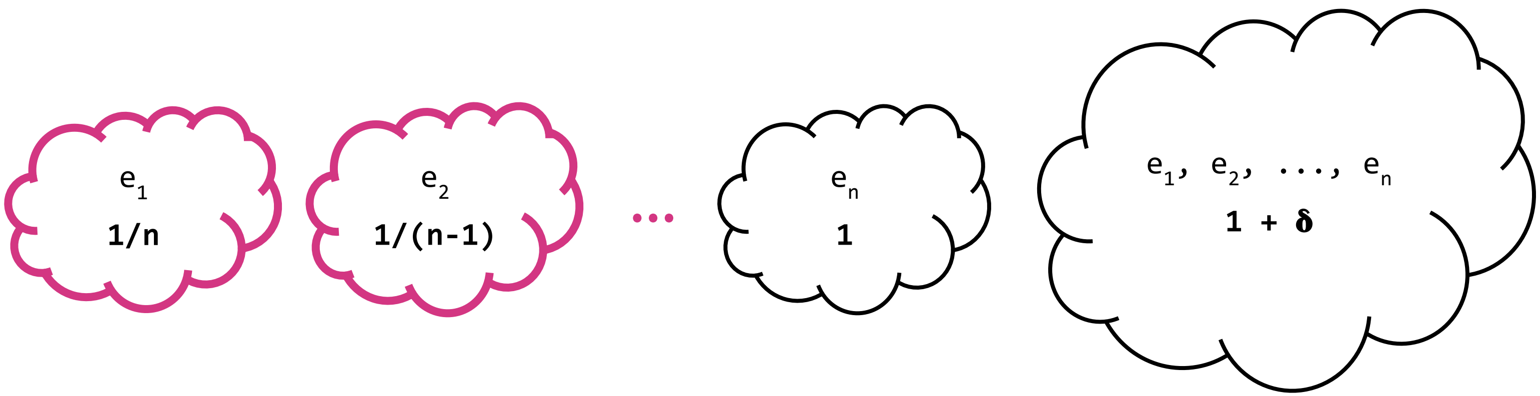


# Set Cover

(2) Can the approximation guarantee of `find_set_cover` be improved by a better analysis?

No. Let's find a **tight example** to convince ourselves that our approximation guarantee is tight i.e. `find_set_cover` produces a solution log(n) times the optimal.

Consider a set of singleton sets and one exhaustive set, each with specific costs.

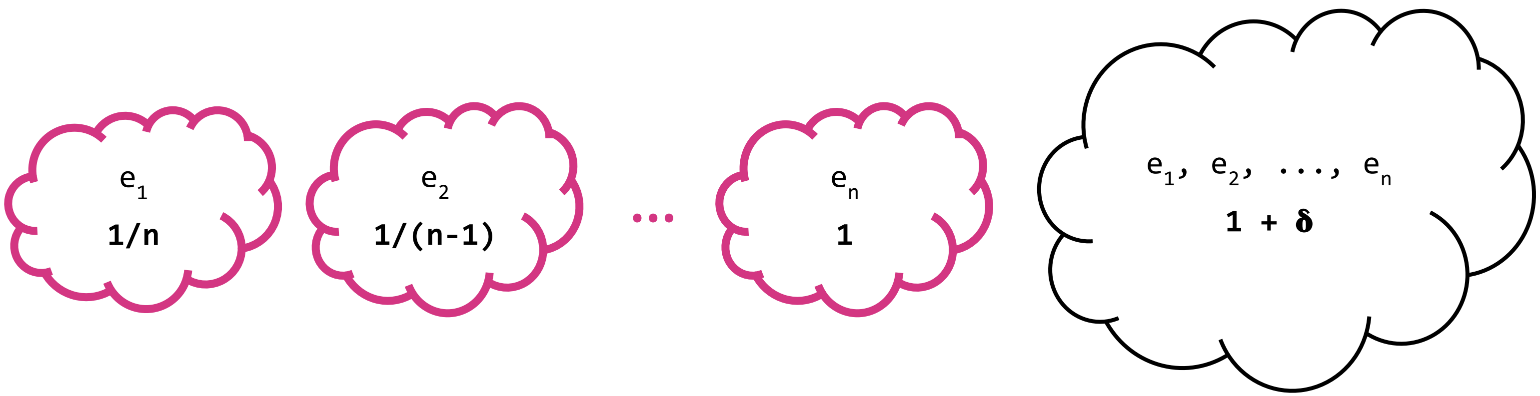


# Set Cover

(2) Can the approximation guarantee of `find_set_cover` be improved by a better analysis?

No. Let's find a **tight example** to convince ourselves that our approximation guarantee is tight i.e. `find_set_cover` produces a solution log(n) times the optimal.

Consider a set of singleton sets and one exhaustive set, each with specific costs.

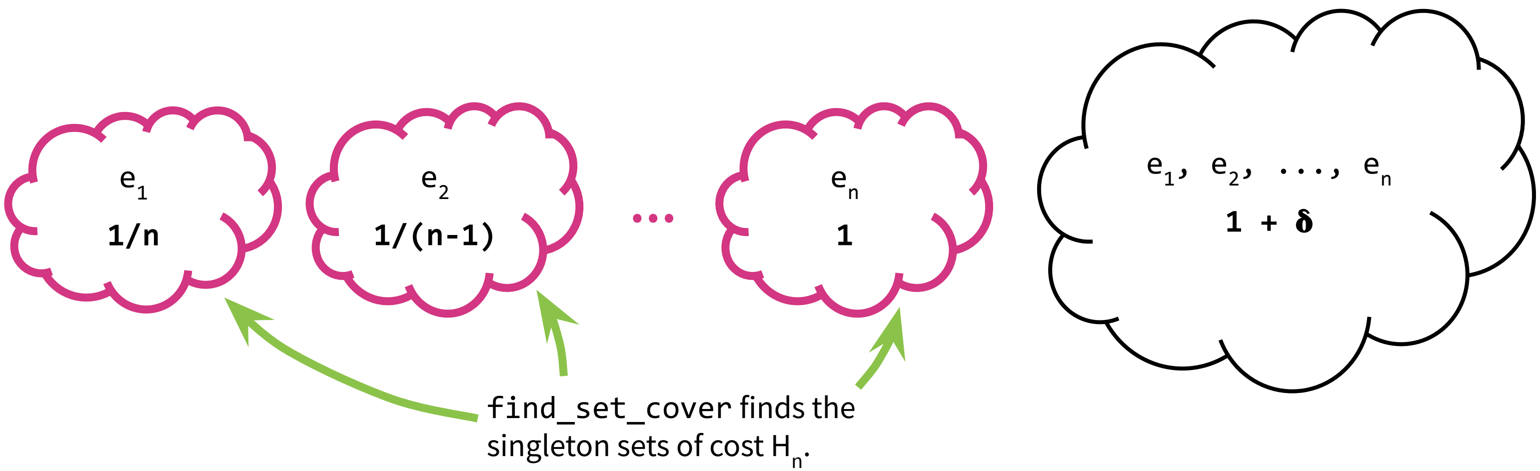


# Set Cover

(2) Can the approximation guarantee of `find_set_cover` be improved by a better analysis?

No. Let's find a **tight example** to convince ourselves that our approximation guarantee is tight i.e. `find_set_cover` produces a solution log(n) times the optimal.

Consider a set of singleton sets and one exhaustive set, each with specific costs.

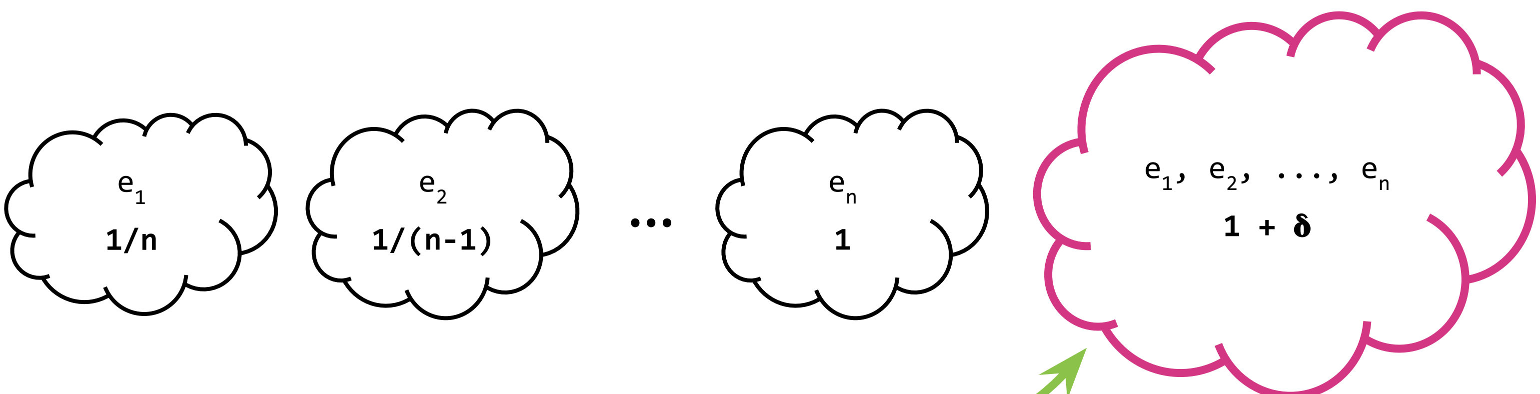


# Set Cover

## (2) Can the approximation guarantee of `find_set_cover` be improved by a better analysis?

No. Let's find a **tight example** to convince ourselves that our approximation guarantee is tight i.e. `find_set_cover` produces a solution log(n) times the optimal.

Consider a set of singleton sets and one exhaustive set, each with specific costs.




`find_set_cover` finds the singleton sets of cost $H_n$.

But the optimal cover has cost $1 + \delta$.

# Set Cover

Study of the **set cover** problem led to the development of fundamental techniques for the entire field of approximation algorithms.

This approximation algorithm is widely (and wildly) useful!

- The human DNA can be represented as a very long string over a four-letter Alphabet. Since it's very long, several overlapping short segments of this string get deciphered, but the locations of these segments on the original remains unknown.
- We can use set cover to find the shortest string which contains these segments as substrings to approximate the original DNA string.

# 0/1 Knapsack III

# Approximation Schemes

## Approximation schemes

Let P be an **NP**-hard optimization problem.

# Approximation Schemes

Approximation schemes

Let P be an **NP**-hard optimization problem.

Let $f_P$ describe the value being optimized

# Approximation Schemes

## Approximation schemes

Let P be an **NP**-hard optimization problem.

Let $f_P$ describe the value being optimized

What is $f_P$ in 0/1 Knapsack? 🤔

# Approximation Schemes

## Approximation schemes

Let P be an **NP**-hard optimization problem.

Let $f_P$ describe the value being optimized

What is $f_P$ in 0/1 Knapsack? 🤔 The total value derived from the items taken.

# Approximation Schemes

## Approximation schemes

- Let P be an **NP**-hard optimization problem.
- Let $f_P$ describe the value being optimized
  - What is $f_P$ in 0/1 Knapsack? 🤔 The total value derived from the items taken.
- Let A(i, ε) be an algorithm, where i is an instance of problem P and ε > 0 is an error parameter.

# Approximation Schemes

## Approximation schemes

Let P be an **NP**-hard optimization problem.

Let $f_P$ describe the value being optimized

What is $f_P$ in 0/1 Knapsack? 🤔 The total value derived from the items taken.

Let A(i, ε) be an algorithm, where i is an instance of problem P and ε > 0 is an error parameter.

A is an **approximation scheme** if it outputs a solution s such that:

$f_P(s) \leq (1 + \varepsilon) \cdot OPT$ if P is a minimization problem

$f_P(s) \geq (1 - \varepsilon) \cdot OPT$ if P is a maximization problem

# Approximation Schemes

## Approximation schemes

Let P be an **NP**-hard optimization problem.

Let $f_P$ describe the value being optimized

What is $f_P$ in 0/1 Knapsack? 🤔 The total value derived from the items taken.

Let A(i, ε) be an algorithm, where i is an instance of problem P and ε > 0 is an error parameter.

A is an **approximation scheme** if it outputs a solution s such that:

$f_P(s) \leq (1 + \varepsilon) \cdot OPT$ if P is a minimization problem

$f_P(s) \geq (1 - \varepsilon) \cdot OPT$ if P is a maximization problem

A is a **fully polynomial time approximation scheme** (FPTAS) if for each fixed ε > 0, its running time is bounded by a polynomial in the size of instance i and 1/ε.

# Approximation Schemes

## Approximation schemes

Let P be an **NP**-hard optimization problem.

Let $f_P$ describe the value being optimized

What is $f_P$ in 0/1 Knapsack? 🤔 The total value derived from the items taken.

Let A(i, ε) be an algorithm, where i is an instance of problem P and $ε > 0$ is an error parameter.

A is an **approximation scheme** if it outputs a solution s such that:

$f_P(s) \leq (1 + ε) \cdot OPT$ if P is a minimization problem

$f_P(s) \geq (1 - ε) \cdot OPT$ if P is a maximization problem

A is a **fully polynomial time approximation scheme** (FPTAS) if for each fixed $ε > 0$, its running time is bounded by a polynomial in the size of instance i and $1/ε$.

FPTAS is the best you can get for an **NP**-hard optimization problem, assuming **P** ≠ **NP**. In other words, it's the holy grail solution for **NP**-hard problems!

# Fixed Parameter Tractability

Suppose that the input to a problem P can be characterized by two parameters, n and k.

P is called fixed-parameter tractable iff there is some algorithm that solves P in time `O(f(k)p(n))`.

- f(k) is an arbitrary function and p(n) is a polynomial in n.
- Intuitively, for any fixed k, the algorithm runs in a polynomial in n since that polynomial p(n) does not depend on choice of k.

# Fixed Parameter Tractability

- Suppose that the input to a problem P can be characterized by two parameters, n and k.
- P is called fixed-parameter tractable iff there is some algorithm that solves P in time `O(f(k)p(n))`.
  - f(k) is an arbitrary function and p(n) is a polynomial in n.
  - Intuitively, for any fixed k, the algorithm runs in a polynomial in n since that polynomial p(n) does not depend on choice of k.
- Which parameter gets “fixed” depends on your perspective.
  - If you’re the robber designing an algorithm to help you steal items, you’ll fix the capacity of the knapsack and reason about variable sets of items.
  - If you’re the victim designing an algorithm to predict which items the robbers with variable size knapsacks will steal, you’ll fix the value of the items.

# Pseudo-Polynomial Time

Recall that to be considered efficient, an algorithm must have runtime polynomial in the size of its input.

- An instance of 0/1 Knapsack requires log(n) bits to represent a number n in binary (e.g. adding one more bit to the end of the representation of W doubles its size and doubles the runtime).

# Pseudo-Polynomial Time

Recall that to be considered efficient, an algorithm must have runtime polynomial in the size of its input.

- An instance of 0/1 Knapsack requires log(n) bits to represent a number n in binary (e.g. adding one more bit to the end of the representation of W doubles its size and doubles the runtime).

To be considered weakly efficient, an algorithm must have runtime pseudo-polynomial in the of its input.

- What is polynomial vs. pseudo-polynomial runtime? Polynomial runtime assumes inputs represented in binary. Pseudo-polynomial runtime assumes inputs represented in unary.
- An instance of 0/1 Knapsack requires n bits to represent a number n in unary (e.g. adding one more bit to the end of the representation of W doesn't really affect its size or its runtime).

6 = 1 1 1 1 1 1

# 0/1 Knapsack

**First DP alg** Previously, we described a fixed-parameter polynomial time algorithm for 0/1 Knapsack.

Our original solution from Lecture 11 answers “What is the maximum value that fits in X capacity given just the first k items?” and fills this table.

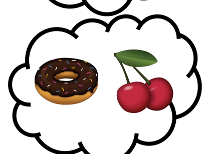
First solve the problem for small knapsacks

Then larger knapsacks

Then larger knapsacks

First solve the problem for few items

Then more items



Then more items

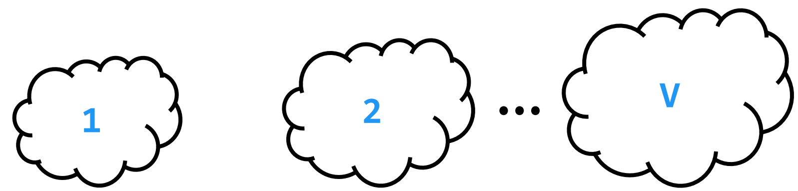
**Fill in maximum values here.**

# 0/1 Knapsack

**New DP alg** Here, we'll describe a pseudo-polynomial time algorithm for 0/1 Knapsack.

Our new solution answers "What is the min capacity needed to make X value with the first k items?" and fills this table.

Let V be the maximum possible value obtainable: $V = v_1 + v_2 + \ldots + v_n$.




First solve the problem for small values

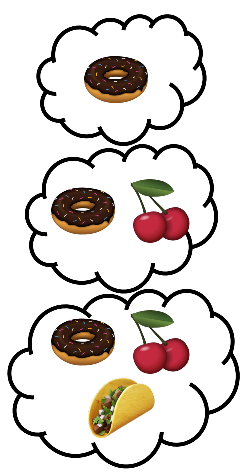
Then larger values

Then larger values

First solve the problem for few items

Then more items

Then more items



**Fill in minimum capacities here.**

# 0/1 Knapsack

## First DP alg

Let **OPT[c,i]** be the optimal (max) value for capacity c with i items.

$$\texttt{OPT[c,i]} = \begin{cases} 0 & \text{if } \mathbf{c} \text{ or } \mathbf{i} \text{ are } 0 \\ \max\{\texttt{OPT[c,i-1]},\ \texttt{OPT}[c-w_i, i-1] + v_i\} & \text{otherwise} \end{cases}$$

O(nW) Dynamic programming solution

## New DP alg

Let **OPT[i,v]** be the optimal (min) capacity for value v with i items.

$$\texttt{OPT[i,v]} = \begin{cases} 0 & \text{if } \mathbf{i} \text{ and } \mathbf{v} \text{ are } 0 \\ \infty & \text{if } \mathbf{i} \text{ is } 0, \mathbf{v} > 0 \\ \texttt{OPT[i-1, v]} & \text{if } v_i > v \\ \min\{\texttt{OPT[i-1,v]},\ \texttt{OPT}[i-1, v-v_i] + w_i\} & \text{otherwise} \end{cases}$$

O(nV) Dynamic programming solution

# 0/1 Knapsack

| | Brute-force | First DP solution | New DP solution |
|---|---|---|---|
| **Runtime** | $O(n2^n)$ **worst-case** | $O(nW)$ **worst-case** | $O(nV)$ **worst-case** |
| **Usage** | Don't do it. | You're the robber! Capacity is fixed and the number of items might grow large. | You're the victim! Total value is fixed and the number of items might grow large. |
| **Analysis** | Exponential, anyway you look at it. | Fixed-parameter polynomial. | Pseudo-polynomial. |

# 0/1 Knapsack

Let's extend on our idea!

**Intuition** If the values of objects were small numbers bounded by a polynomial in n, then **New DP alg** would be a regular polynomial-time algorithm, so let's coerce the values of the items to be small.

# 0/1 Knapsack

```
def zero_one_knapsack(capacity, weights, values):
  k = εv_max/n
  v_i' = ⌊v_i/k⌋ for v_i in values
  S', value = use the value-based DP algorithm to find the
    most valuable items using values v_i' and weights.
  return S', value * k
```

**Runtime:** $O(nV)$

# 0/1 Knapsack

## (1) How do we establish an approximation guarantee for our algorithm?

Let A be the set output by `zero_one_knapsack`.

**Lemma:** value(A) $\geq (1 - \varepsilon) \cdot$ OPT

**Proof:**

Let O be the optimal set. For any object a, because of rounding down, $k \cdot v_a'$ can be smaller than $v_a$ but by not more than k. Thus,

$$\text{value}(O) - k \cdot \text{value}'(O) \leq nk$$

The DP step must return a set at least as good as O under the new values. Therefore,

$$\text{value}(S') \geq k \cdot \text{value}'(O) \geq \text{value}(O) - nk = \text{OPT} - \varepsilon v_{max} \geq (1 - \varepsilon) \cdot \text{OPT}$$

where the last inequality follows from the fact that OPT $\geq v_{max}$. ■

# 0/1 Knapsack

(1) How do we establish an approximation guarantee for our algorithm?

Let A be the set output by `zero_one_knapsack`.

**Lemma:** value(A) $\geq (1 - \varepsilon) \cdot$ OPT

**Theorem:** `zero_one_knapsack` is a FPTAS for Knapsack.

**Proof:**

By the lemma, the solution found is within $(1 - \varepsilon)$ factor of OPT. Since the running time of the algorithm is $O(n^2 \lfloor v_{max}/k \rfloor) = O(n^2 \lfloor n/\varepsilon \rfloor)$, which is polynomial in n and $1/\varepsilon$, the theorem follows. ◾

**We have the holy grail of approximation algorithms!**

# 0/1 Knapsack

## (1) How do we establish an approximation guarantee for our algorithm?

Let A be the set output by `zero_one_knapsack`.

**Lemma:** value(A) $\geq (1 - \varepsilon) \cdot \mathrm{OPT}$

**Theorem:** `zero_one_knapsack` is a FPTAS for Knapsack.

**Proof:**

By the lemma, the solution found is within $(1 - \varepsilon)$ factor of OPT. Since the running time of the algorithm is $O(n) = O(n^2 \lfloor v_{max}/k \rfloor) = O(n^2 \lfloor n/\varepsilon \rfloor)$, which is polynomial in $n$ and $1/\varepsilon$, the theorem follows. ■

**We have the holy grail of approximation algorithms!**

# Conclusion

| | Vertex Cover | Set Cover | 0/1 Knapsack |
|---|---|---|---|
| **Runtime** | $O(\|V\|+\|E\|)$ worst-case | $O(\|S\|^2)$ worst-case | $O(nV)$ worst-case |
| **Approximation** | 2 | $\log(\|U\|)$ | $1 - \varepsilon$ |
| **Analysis** | - | - | FPTAS. |