Tribhuvan University
Institute of Engineering
Purwanchal Campus
पूर्वाञ्चल क्याम्पस

9. Approximation Algorithms

Pukar Karki
Assistant Professor

Approximation Algorithms

- Many problems of practical significance are **NP-complete**.
- Even if a problem is NP-complete, there may be hope. We have at least three ways to get around NP-completeness.

1. First, if the actual inputs are small, an algorithm with exponential running time may be perfectly satisfactory.
2. Second, we may be able to isolate important special cases that we can solve in polynomial time.
3. Third, we might come up with approaches to find near-optimal solutions in polynomial time (either in the worst case or the expected case).

- In practice, **near-optimality** is often good enough. We call an algorithm that returns **near-optimal solutions** an **approximation algorithm.**

Performance Ratios For Approximation Algorithms

- Suppose that we are working on an optimization problem in which each potential solution has a positive cost, and we wish to find a near-optimal solution.
- Depending on the problem, we may define an optimal solution as one with maximum possible cost or one with minimum possible cost; that is, the problem may be either a maximization or a minimization problem.

Performance Ratios For Approximation Algorithms

- We say that an algorithm for a problem has an approximation ratio of **ρ(n)** if, for any input of size n, the cost C of the solution produced by the algorithm is within a factor of **ρ(n)** of the cost C* of an optimal solution:

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \le \rho(n)$$

- If an algorithm achieves an approximation ratio of **ρ(n)**, we call it a **ρ(n)** approximation algorithm.

Performance Ratios For Approximation Algorithms

- For a maximization problem, $0 < C \leq C^*$, and the ratio C^*/C gives the factor by which the cost of an optimal solution is larger than the cost of the approximate solution.

- Similarly, for a minimization problem, $0 < C^* \leq C$, and the ratio C/C^* gives the factor by which the cost of the approximate solution is larger than the cost of an optimal solution.

The Vertex-Cover Problem

- A **vertex cover** of an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ such that if (u, v) is an edge of G, then either $u \in V'$ or $v \in V'$ (or both).
- The size of a **vertex cover** is the number of vertices in it.
- The **vertex-cover problem** is to fi**nd a vertex cover of minimum size in a given undirected graph**.
- We call such a vertex cover an optimal vertex cover.

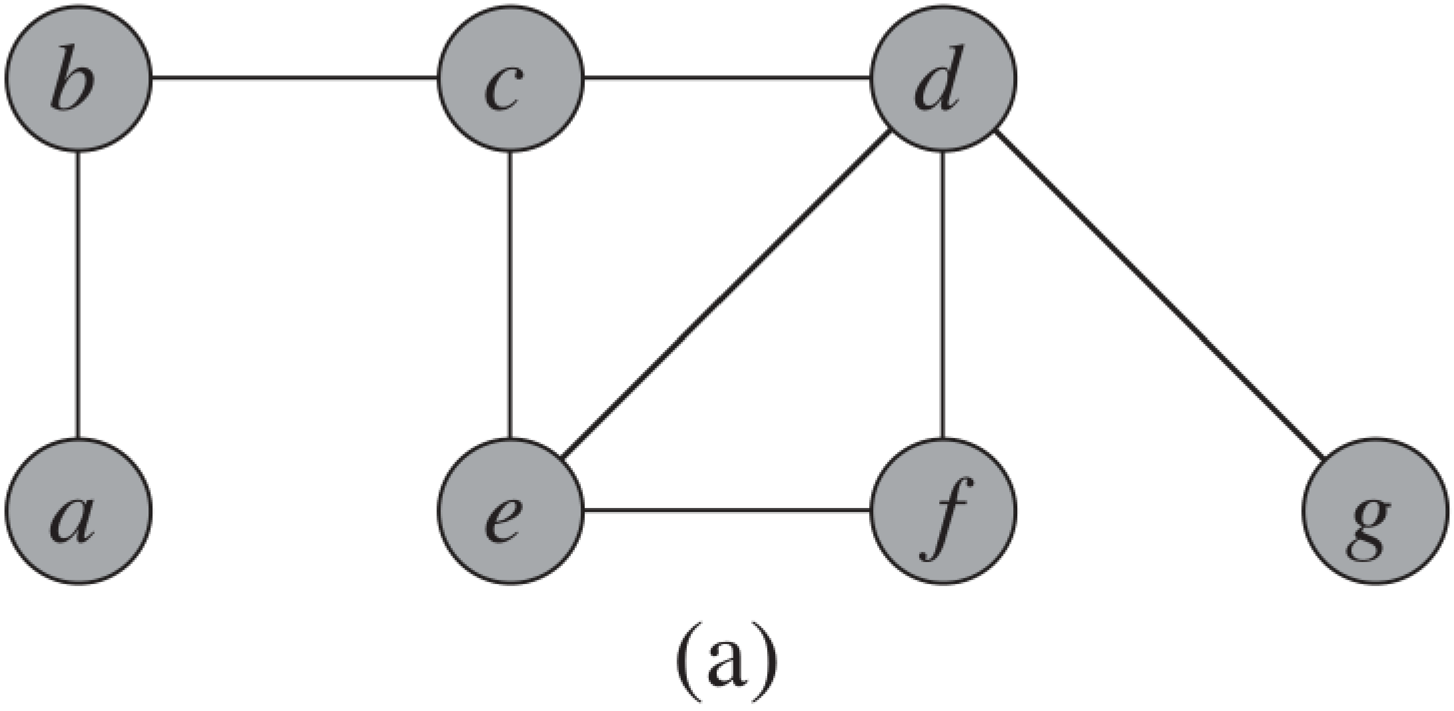
The Vertex-Cover Problem

- Even though we don’t know how to find an optimal vertex cover in a graph G in polynomial time, we can efficiently find a vertex cover that is near-optimal.
- The following approximation algorithm takes as input an undirected graph G and returns a vertex cover whose size is guaranteed to be no more than twice the size of an optimal vertex cover.

The Vertex-Cover Problem

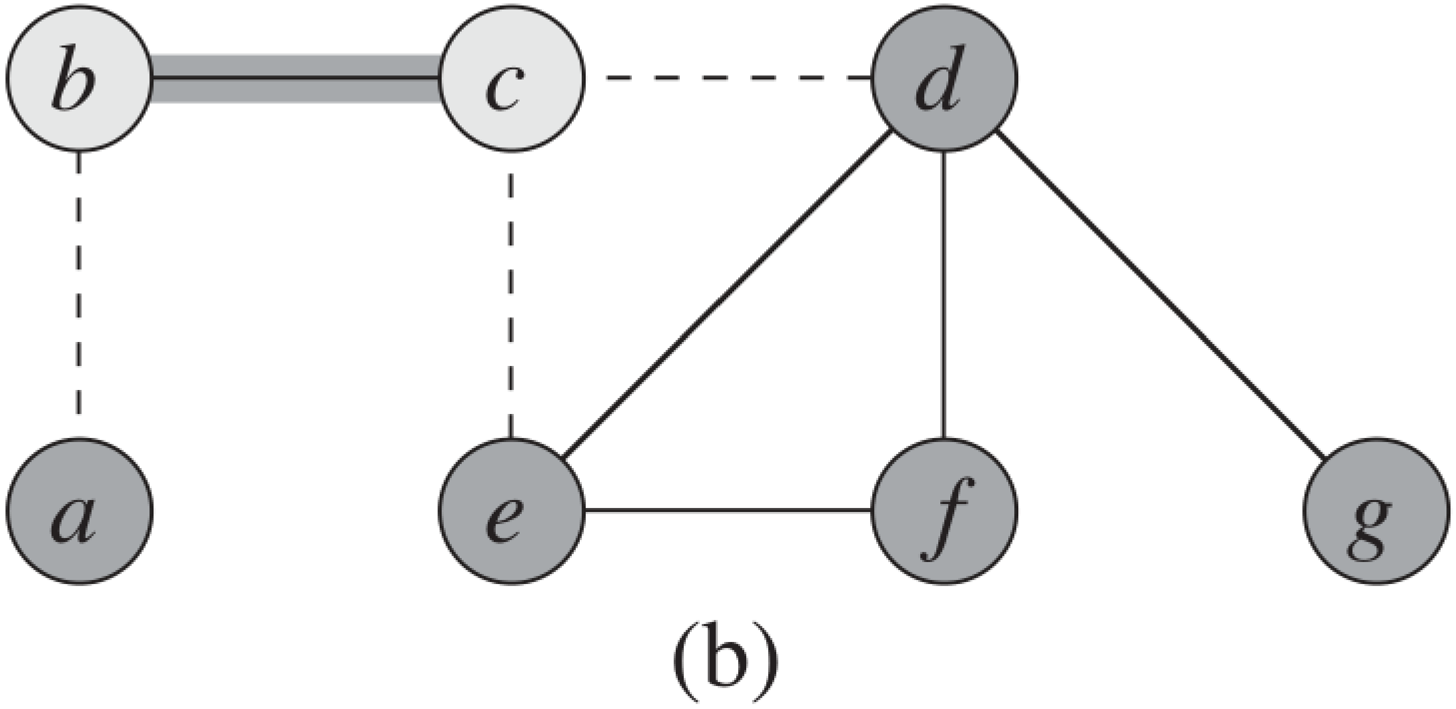
```
APPROX-VERTEX-COVER$(G)$
$C = \emptyset$
$E' = G.E$
while $E' \neq \emptyset$
    let $(u, v)$ be an arbitrary edge of $E'$
    $C = C \cup \{u, v\}$
    remove from $E'$ every edge incident on either $u$ or $v$
return $C$
```

The Vertex-Cover Problem



(a) The input graph G, which has 7 vertices and 8 edges.

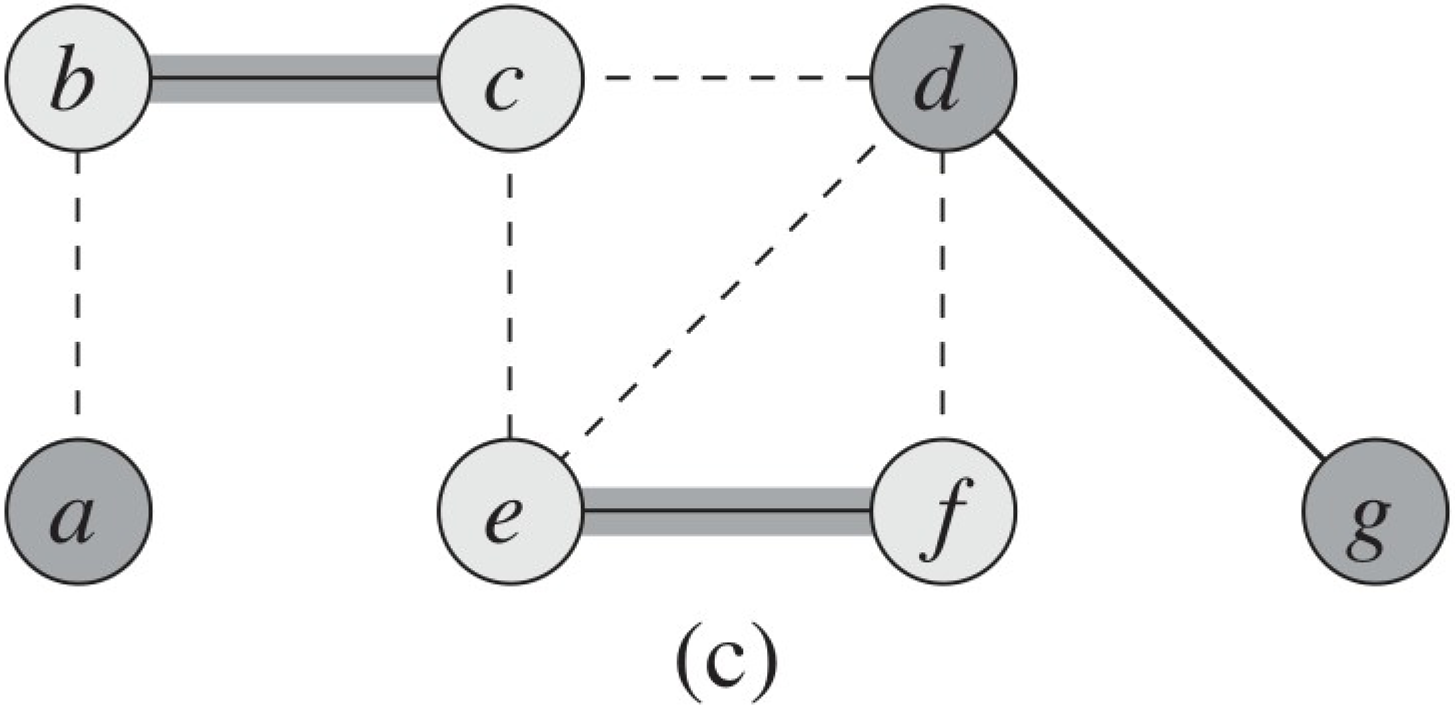
The Vertex-Cover Problem



(b) The edge (b,c), shown heavy, is the first edge chosen by our algorithm.

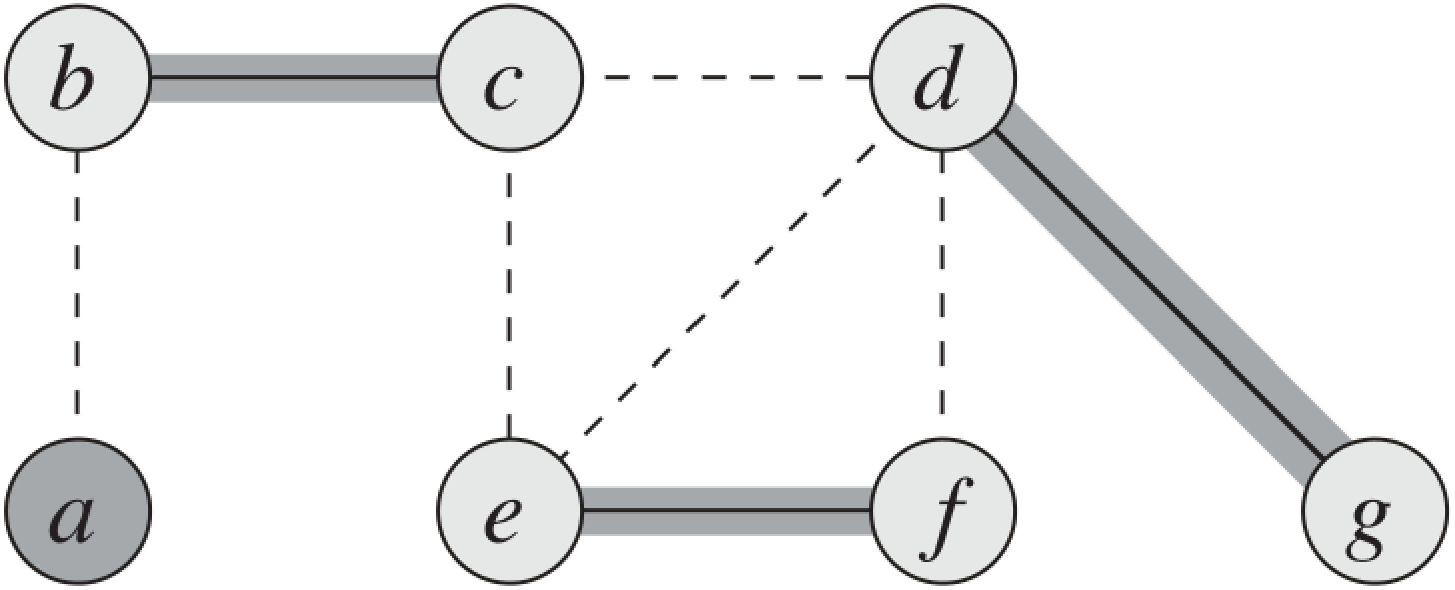
Vertices b and c, shown lightly shaded, are added to the set C containing the vertex cover being created. Edges (a, b), (c, e), and (c, d), shown dashed, are removed since they are now covered by some vertex in C.

The Vertex-Cover Problem



(c) Edge (e, f) is chosen; vertices e and f are added to C.

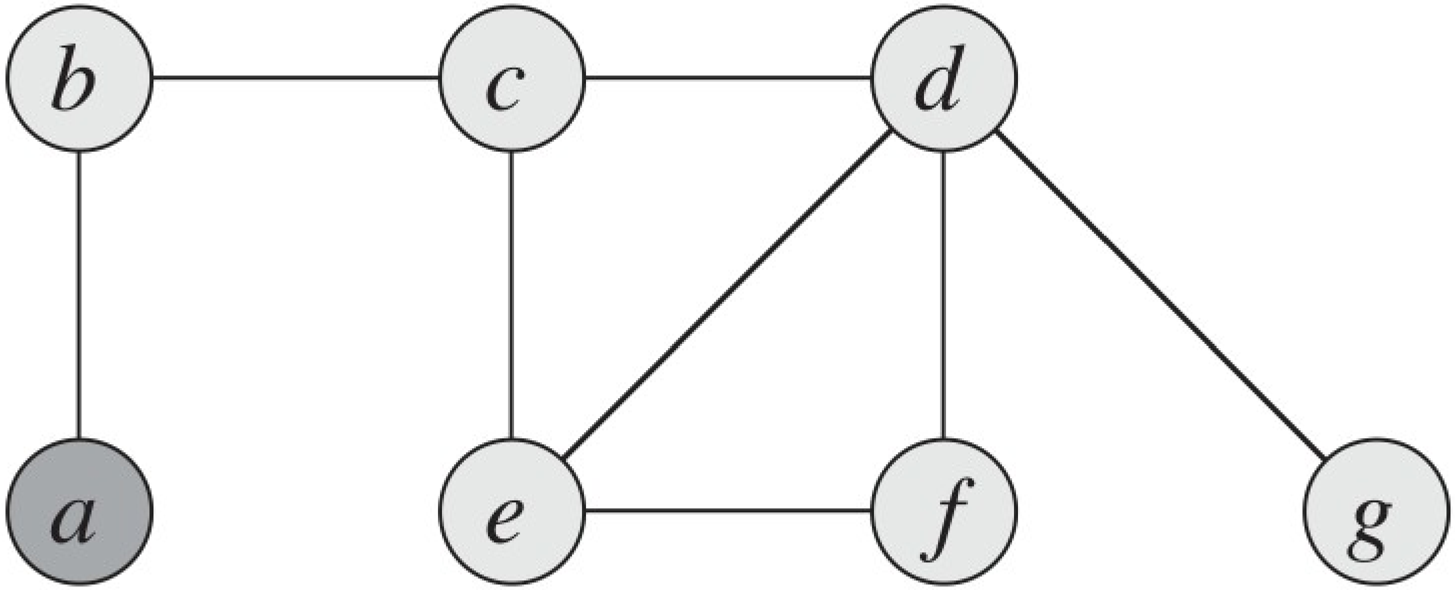
The Vertex-Cover Problem



(d)

d) Edge (d, g) is chosen; vertices d and g are added to C.

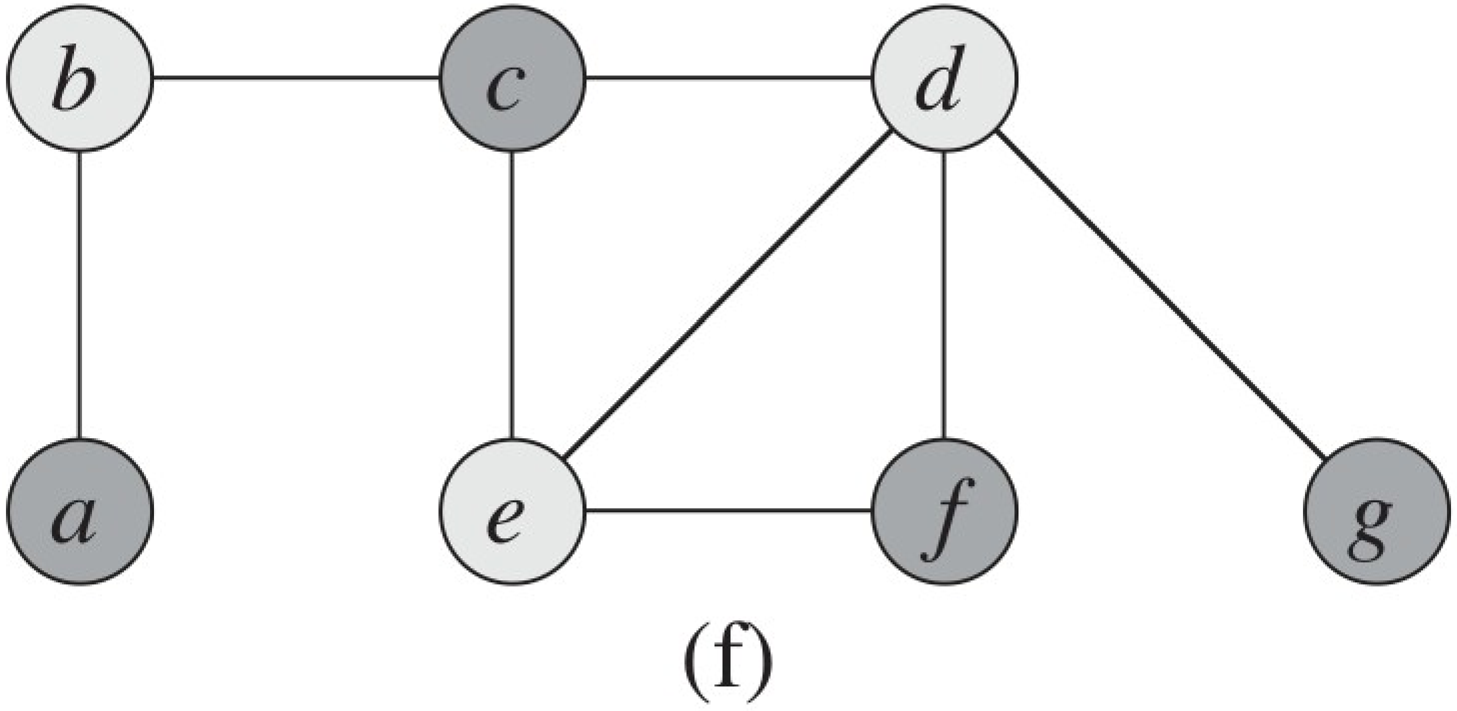
The Vertex-Cover Problem



(e)

e) The set C, which is the vertex cover produced contains the six vertices {b, c, d, e, f, g}.

The Vertex-Cover Problem



(f) The optimal vertex cover for this problem contains only three vertices: b, d, and e.

The Vertex-Cover Problem

```
APPROX-VERTEX-COVER$(G)$
1  $C = \emptyset$
2  $E' = G.E$
3  while $E' \neq \emptyset$
4      let $(u, v)$ be an arbitrary edge of $E'$
5      $C = C \cup \{u, v\}$
6      remove from $E'$ every edge incident on either $u$ or $v$
7  return $C$
```

- The variable C contains the vertex cover being constructed. Line 1 initializes C to the empty set.
- Line 2 sets E' to be a copy of the edge set G.E of the graph.
- The loop of lines 3–6 repeatedly picks an edge (u, v) from E', adds its endpoints u and v to C, and deletes all edges in E' that are covered by either u or v.
- Finally, line 7 returns the vertex cover C
- The running time of this algorithm is
  O(V + E),
  using adjacency lists to represent E'.

APPROX-VERTEX-COVER is a polynomial-time 2-approximation algorithm.

Proof:

- ✓ The set C of vertices that is returned by APPROX-VERTEX-COVER is a vertex cover, since the algorithm loops until every edge in G:E has been covered by some vertex in C.
- ✓ To see that APPROX-VERTEX-COVER returns a vertex cover that is at most twice the size of an optimal cover, let A denote the set of edges that line 4 of APPROX-VERTEX-COVER picked.
- ✓ In order to cover the edges in A, any vertex cover—in particular, an optimal cover C* - must include at least one endpoint of each edge in A.
- ✓ No two edges in A share an endpoint, since once an edge is picked in line 4, all other edges that are incident on its endpoints are deleted from E' in line 6.
- ✓ Thus, no two edges in A are covered by the same vertex from C*, and we have the lower bound

$$|C^*| \geq |A| \text{ ----- i)}$$

on the size of an optimal vertex cover.

APPROX-VERTEX-COVER is a polynomial-time 2-approximation algorithm.

Proof: (Cont...)

- ✓ Each execution of line 4 picks an edge for which neither of its endpoints is already in C, yielding an upper bound (an exact upper bound, in fact) on the size of the vertex cover returned:

$$|C| = 2|A| \text{ ----- ii)}$$

- ✓ Combining equations,

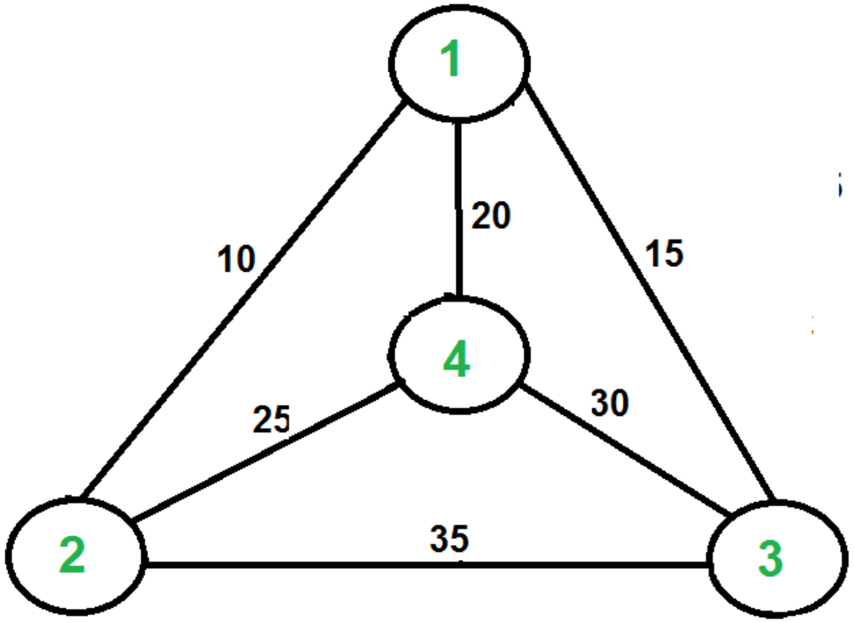
$$|C| = 2|A|$$

$$\text{or, } |C| \leq 2|C^*|$$

- ✓

The Traveling-Salesman Problem

- In the **traveling-salesman problem**, we are given a complete undirected graph G = (V, E) that has a non-negative integer cost c(u, v) associated with each edge (u, v) ∈ E, and we must find a **hamiltonian cycle** (a tour) of G with minimum cost.



A TSP tour in the graph is 1-2-4-3-1. The cost of the tour is 10+25+30+15 which is 80.

The Traveling-Salesman Problem

- Let c(A) denote the total cost of the edges in the subset $A \subseteq E$:

$$c(A) = \sum_{(u,v)\in A} c(u,v)$$

- Triangle inequality states that if, for all vertices u, v, w $\in V$,

$$c(u, w) \leq c(u, v) + c(v, w)$$

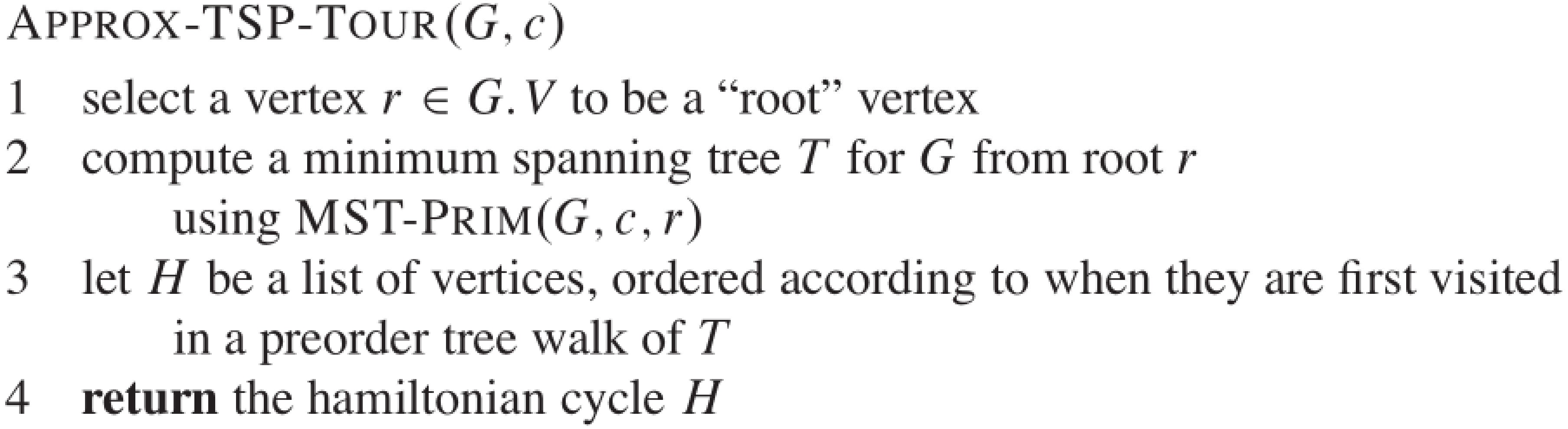
The Traveling-Salesman Problem

- The triangle inequality seems as though it should naturally hold, and it is automatically satisfied in several applications.
- For example, if the vertices of the graph are points in the plane and the cost of traveling between two vertices is the ordinary euclidean distance between them, then the triangle inequality is satisfied.
- Furthermore, many cost functions other than euclidean distance satisfy the triangle inequality.

Traveling-Salesman Problem with the Triangle Inequality

- We first compute a structure—a minimum spanning tree—whose weight gives a lower bound on the length of an optimal traveling-salesman tour.
- We shall then use the minimum spanning tree to create a tour whose cost is no more than twice that of the minimum spanning tree’s weight, as long as the cost function satisfies the triangle inequality.
- The following algorithm implements this approach, calling the minimum-spanning-tree algorithm MST-PRIM as a subroutine.
- The parameter G is a complete undirected graph, and the cost function c satisfies the triangle inequality.

Traveling-Salesman Problem with the Triangle Inequality

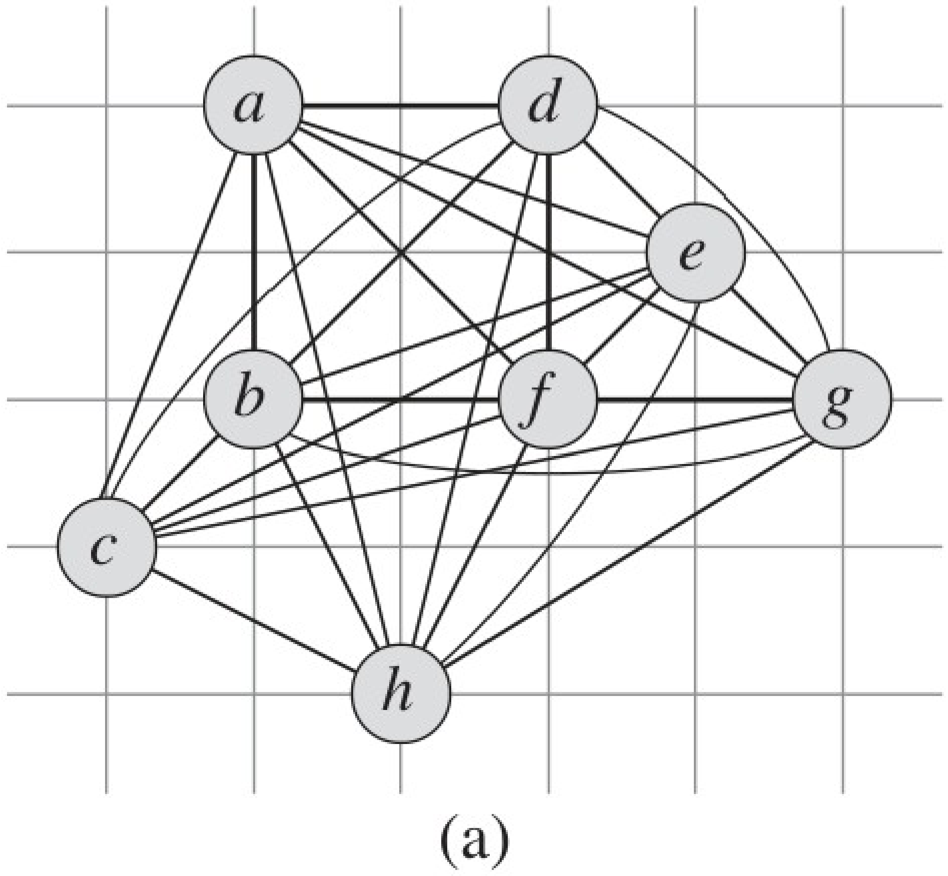


```
APPROX-TSP-TOUR($G, c$)
1  select a vertex $r \in G.V$ to be a "root" vertex
2  compute a minimum spanning tree $T$ for $G$ from root $r$
       using MST-PRIM($G, c, r$)
3  let $H$ be a list of vertices, ordered according to when they are first visited
       in a preorder tree walk of $T$
4  **return** the hamiltonian cycle $H$
```

Traveling-Salesman Problem with the Triangle Inequality

```
MST-PRIM($G, w, r$)
for each $u \in G.V$
    $u.key = \infty$
    $u.\pi = \text{NIL}$
$r.key = 0$
$Q = G.V$
while $Q \neq \emptyset$
    $u = \text{EXTRACT-MIN}(Q)$
    for each $v \in G.Adj[u]$
        if $v \in Q$ and $w(u, v) < v.key$
            $v.\pi = u$
            $v.key = w(u, v)$
```

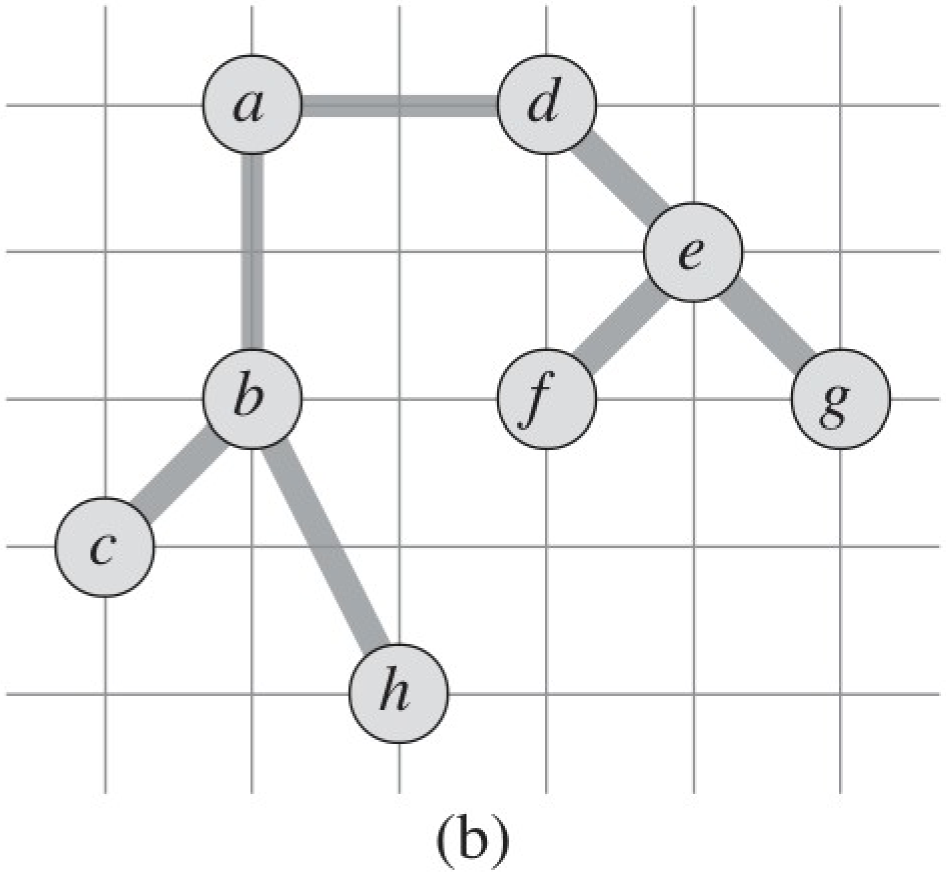
Traveling-Salesman Problem with the Triangle Inequality



(a) A complete undirected graph.

- **Vertices lie on intersections of integer grid lines.**
- **For example, f is one unit to the right and two units up from h.**
- **The cost function between two points is the ordinary euclidean distance.**

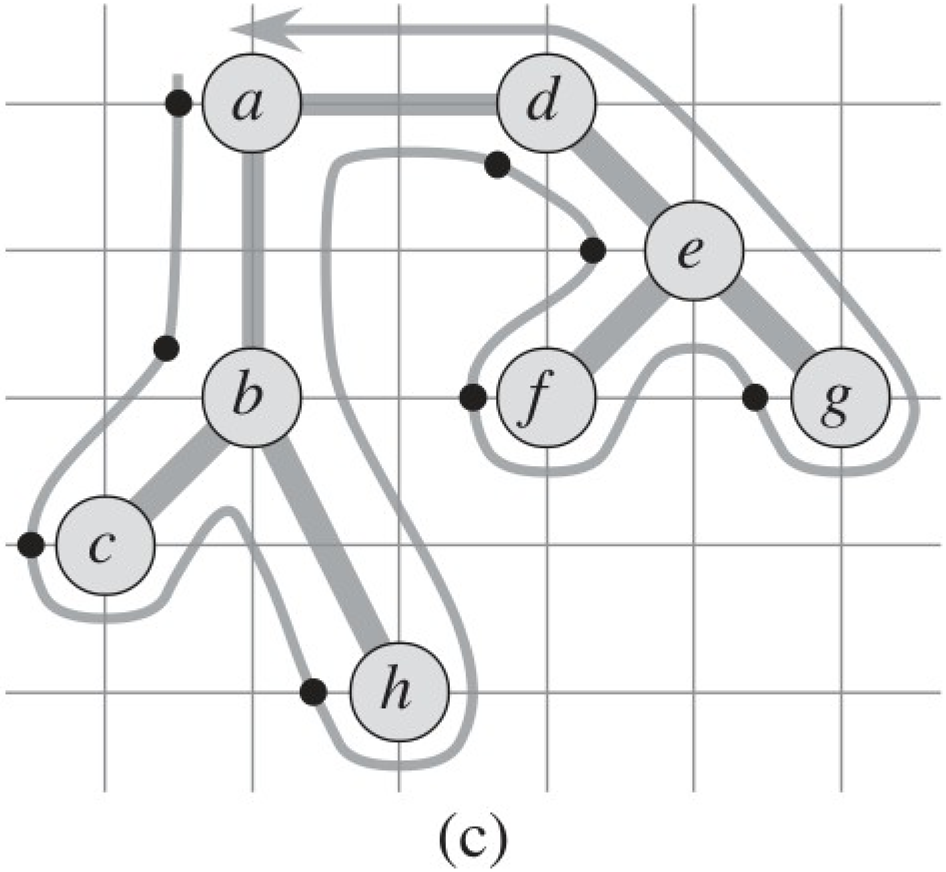
Traveling-Salesman Problem with the Triangle Inequality



(b) A minimum spanning tree T of the complete graph, as computed by MST-PRIM.

- **Vertex a is the root vertex.**
- **Only edges in the minimum spanning tree are shown.**
- **The vertices happen to be labeled in such a way that they are added to the main tree by MST-PRIM in alphabetical order.**

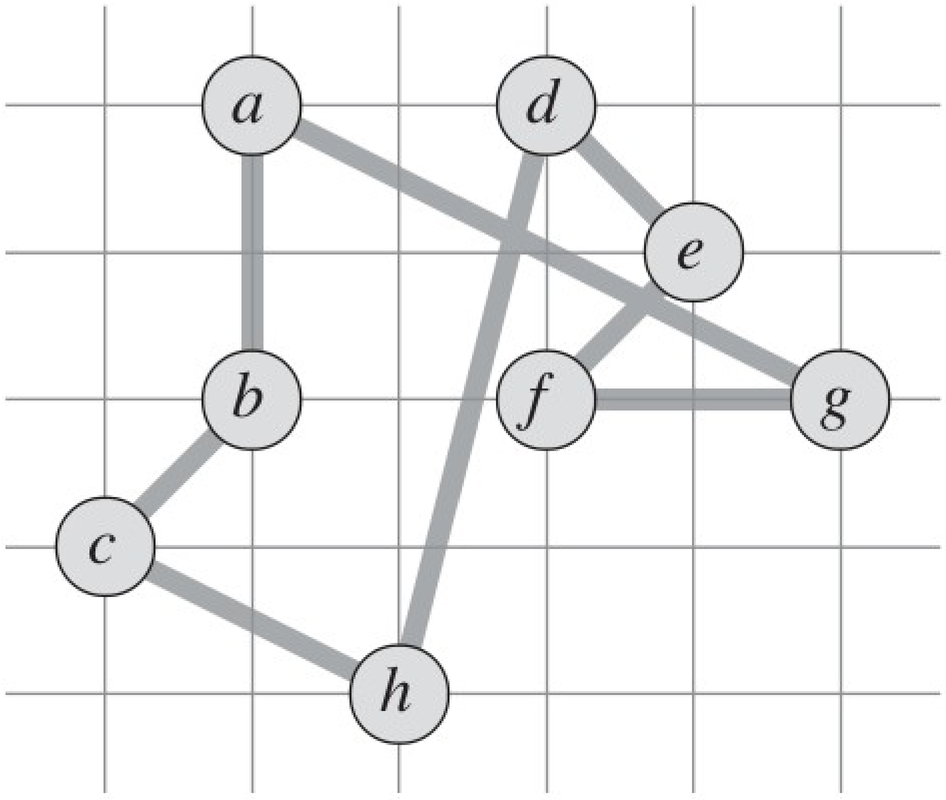
Traveling-Salesman Problem with the Triangle Inequality



(c) A walk of T , starting at a.

- **A full walk of the tree visits the vertices in the order a, b, c, b, h, b, a, d, e, f, e, g, e, d, a.**
- **A preorder walk of T lists a vertex just when it is first encountered, as indicated by the dot next to each vertex, yielding the ordering a, b, c, h, d, e, f, g.**

Traveling-Salesman Problem with the Triangle Inequality

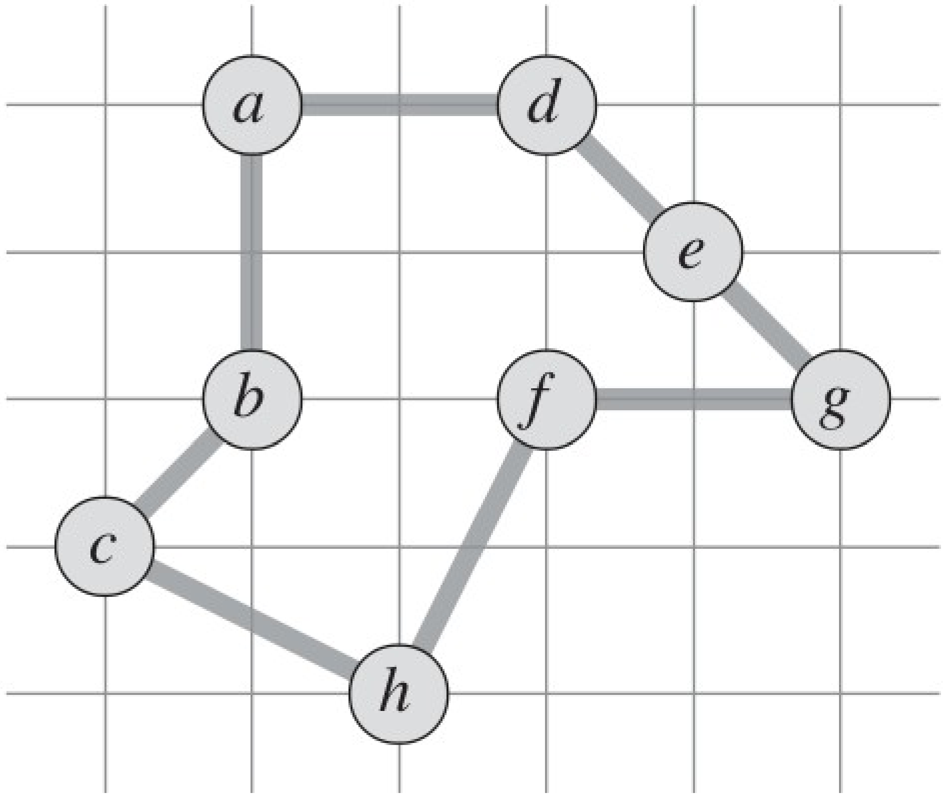


(d)

(d) A tour obtained by visiting the vertices in the order given by the preorder walk, which is the tour H returned by APPROX-TSP-TOUR.

- **Its total cost is approximately 19:074.**

Traveling-Salesman Problem with the Triangle Inequality



(e)

(e) An optimal tour H* for the original complete graph.

- **Its total cost is approximately 14.715.**

Traveling-Salesman Problem with the Triangle Inequality

- Even with a simple implementation of MST-PRIM, the running time of APPROX-TSP-TOUR is $\Theta(V^2)$.
- If the cost function for an instance of the traveling-salesman problem satisfies the triangle inequality, then APPROX-TSP-TOUR returns a tour whose cost is not more than twice the cost of an optimal tour.
- Refer to Page 1114 of “Introduction to Algorithms”

The General Traveling-Salesman Problem

- If we drop the assumption that the cost function c satisfies the triangle inequality, then we cannot find good approximate tours in polynomial time unless P = NP.

The Set-Covering Problem

- An instance (X, F) of the set-covering problem consists of a finite set X and a family F of subsets of X, such that every element of X belongs to at least one subset in F:

$$X = \bigcup_{S \in \mathcal{F}} S$$

- We say that a subset S ⊆ F .
- The problem is to find a minimum- size subset C ⊆ F whose members cover all of X:

$$X = \bigcup_{S \in \mathcal{C}} S$$

- We say that any C satisfying the above equation covers X.

The Set-Covering Problem

- ✔ The size of C is the number of sets it contains, rather than the number of individual elements in these sets, since every subset C that covers X must contain all |X| individual elements.

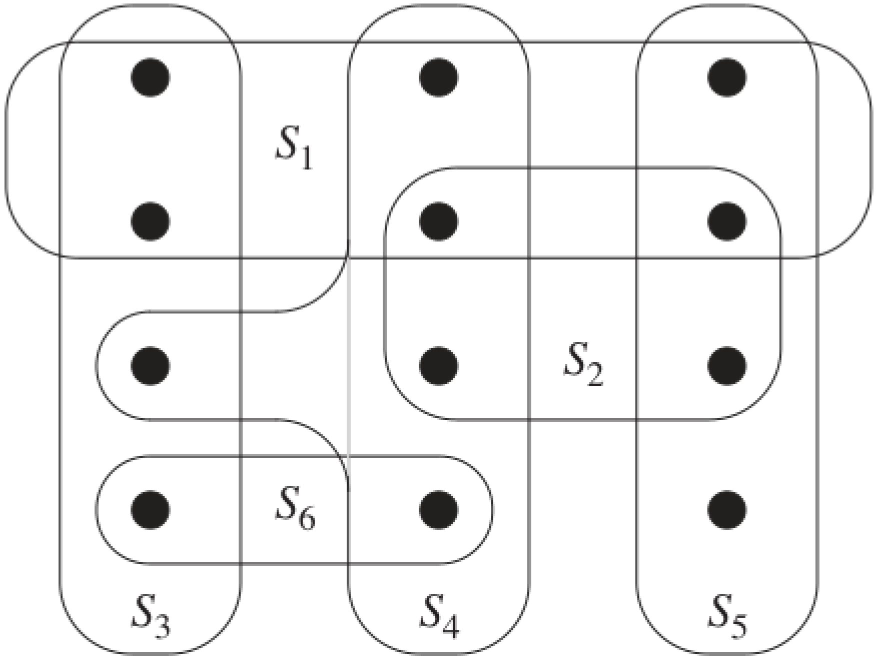


Fig: An instance (X, F) of the set-covering problem, where X consists of the 12 black points and F = {S_1, S_2, S_3, S_4, S_5, S_6}

✔ A minimum-size set cover is C = {S_3, S4, S_5}, with size 3.

The Set-Covering Problem

- The set-covering problem abstracts many commonly arising combinatorial problems.
- As a simple example, suppose that X represents a set of skills that are needed to solve a problem and that we have a given set of people available to work on the problem.
- We wish to form a committee, containing as few people as possible, such that for every requisite skill in X, at least one member of the committee has that skill.
- In the decision version of the set-covering problem, we ask whether a covering exists with size at most k, where k is an additional parameter specified in the problem instance.
- The decision version of the problem is NP-complete

The Set-Covering Problem

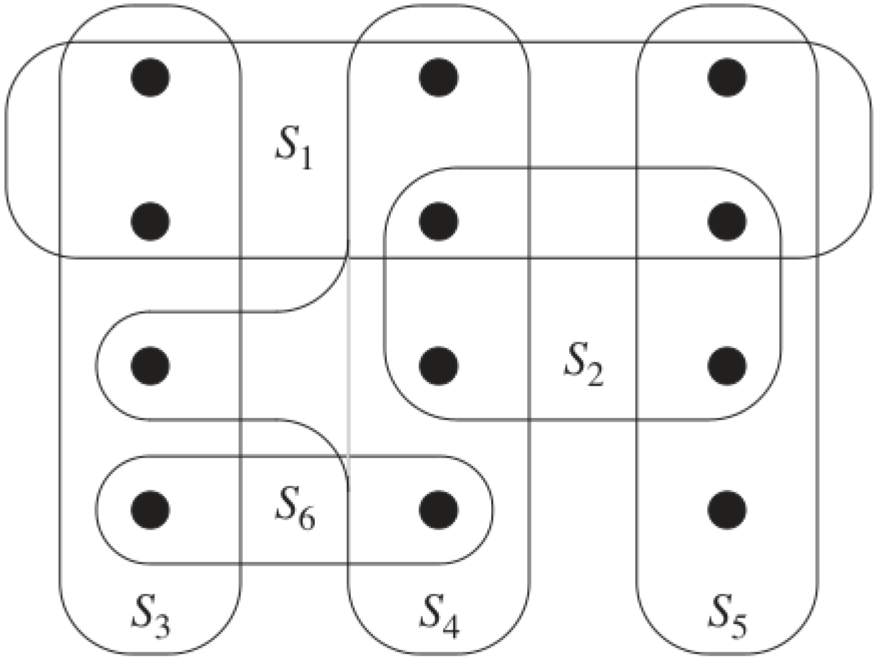
A greedy approximation algorithm

- The greedy method works by picking, at each stage, the set S that covers the greatest number of remaining elements that are uncovered.

```
GREEDY-SET-COVER$(X, \mathcal{F})$
$U = X$
$\mathcal{C} = \emptyset$
while $U \neq \emptyset$
    select an $S \in \mathcal{F}$ that maximizes $|S \cap U|$
    $U = U - S$
    $\mathcal{C} = \mathcal{C} \cup \{S\}$
return $\mathcal{C}$
```

The Set-Covering Problem

A greedy approximation algorithm



The greedy algorithm produces a cover of size 4 by selecting either the sets S_1, S_4, S_5, and S_3 or the sets S_1, S_4, S_5, and S_6, in order.

The Set-Covering Problem

```
GREEDY-SET-COVER$(X, \mathcal{F})$
1  $U = X$
2  $\mathcal{C} = \emptyset$
3  while $U \neq \emptyset$
4      select an $S \in \mathcal{F}$ that maximizes $|S \cap U|$
5      $U = U - S$
6      $\mathcal{C} = \mathcal{C} \cup \{S\}$
7  return $\mathcal{C}$
```

The algorithm works as follows.

- The set U contains, at each stage, the set of remaining uncovered elements.
- The set C contains the cover being constructed.
- Line 4 is the greedy decision making step, choosing a subset S that covers as many uncovered elements as possible (breaking ties arbitrarily).
- After S is selected, line 5 removes its elements from U , and line 6 places S into C.
- When the algorithm terminates, the set C contains a subfamily of F that covers X.

The Set-Covering Problem

```
GREEDY-SET-COVER$(X, \mathcal{F})$
1  $U = X$
2  $\mathcal{C} = \emptyset$
3  while $U \neq \emptyset$
4      select an $S \in \mathcal{F}$ that maximizes $|S \cap U|$
5      $U = U - S$
6      $\mathcal{C} = \mathcal{C} \cup \{S\}$
7  return $\mathcal{C}$
```

- ✔ We can easily implement GREEDY-SET-COVER to run in time polynomial in |X| and |F|.
- ✔ Since the number of iterations of the loop on lines 3–6 is bounded from above by min(|X|, |F|), and we can implement the loop body to run in time O(|X| |F|), a simple implementation runs in time
  O(|X| |F| min(|X|, |F|))

The Subset-Sum Problem

- An instance of the subset-sum problem is a pair (S, t), where S is a set $\{x_1, x_2, ... x_n\}$ of positive integers and t is a positive integer.
- This **decision problem** asks whether there exists a subset of S that adds up exactly to the target value t.
- This problem is NP-complete.

The Subset-Sum Problem

- The **optimization problem** associated with this decision problem arises in practical applications.
- In the **optimization problem**, we wish to find a subset of $\{x_1, x_2 \ldots x_n\}$ whose sum is as large as possible but not larger than t.
- For example, we may have a truck that can carry no more than **t** pounds, and **n** different boxes to ship, the **i**th of which weighs $\mathbf{x_i}$ pounds.
- We wish to fill the truck with as heavy a load as possible without exceeding the given weight limit.

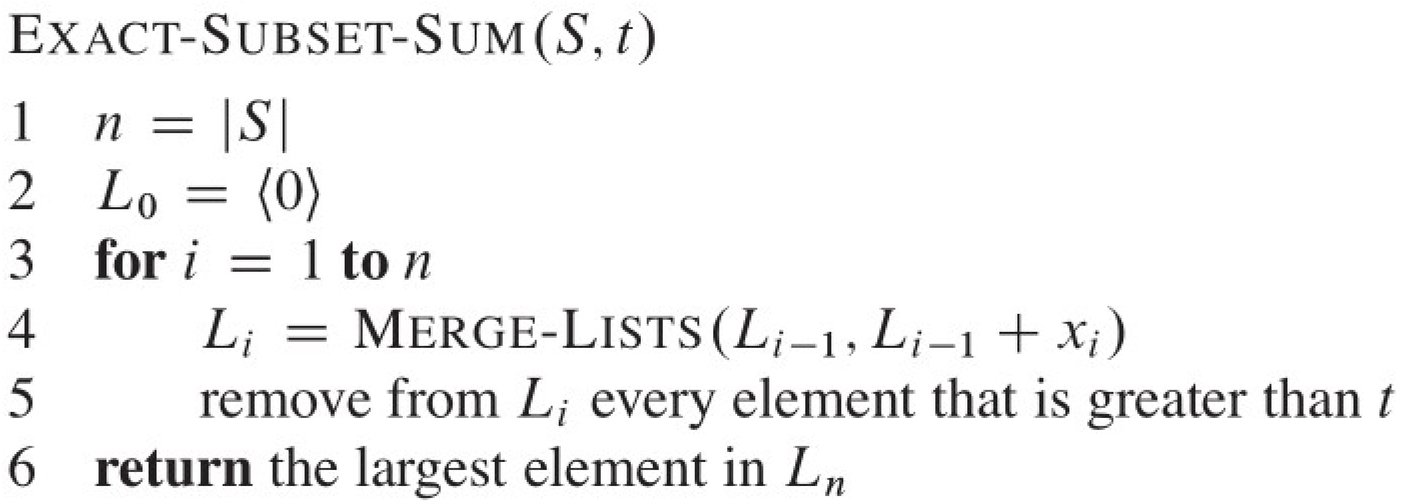
The Subset-Sum Problem

An exponential-time exact algorithm

```
EXACT-SUBSET-SUM$(S, t)$
$n = |S|$
$L_0 = \langle 0 \rangle$
for $i = 1$ to $n$
    $L_i =$ MERGE-LISTS$(L_{i-1}, L_{i-1} + x_i)$
    remove from $L_i$ every element that is greater than $t$
return the largest element in $L_n$
```

The Subset-Sum Problem

An exponential-time exact algorithm



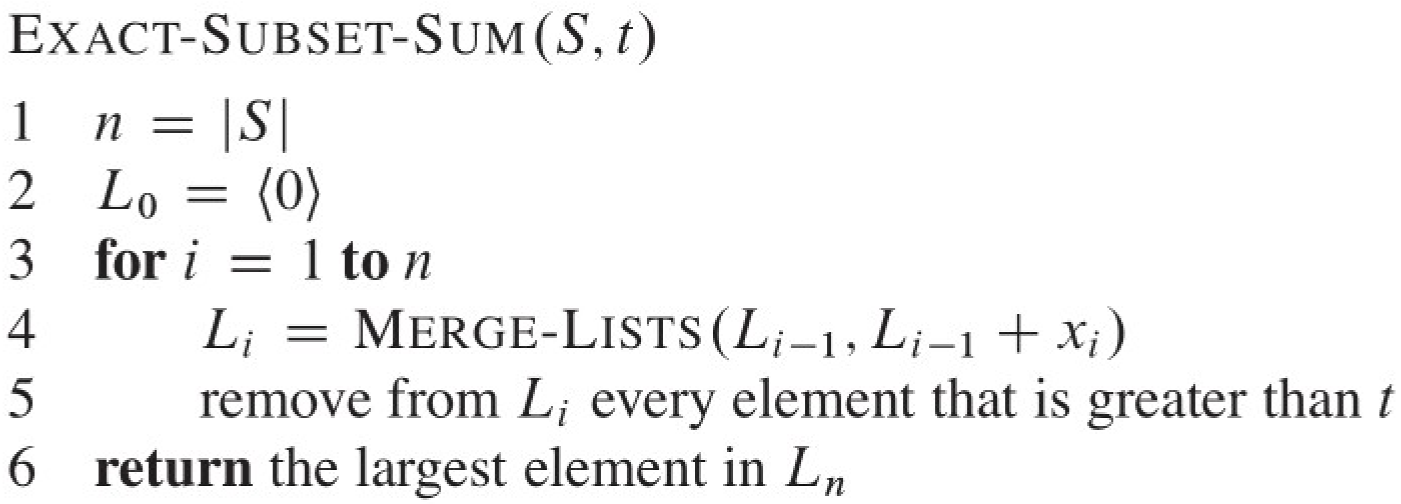
```
EXACT-SUBSET-SUM$(S, t)$
1   $n = |S|$
2   $L_0 = \langle 0 \rangle$
3   for $i = 1$ to $n$
4       $L_i = $ MERGE-LISTS$(L_{i-1}, L_{i-1} + x_i)$
5       remove from $L_i$ every element that is greater than $t$
6   return the largest element in $L_n$
```

- The procedure EXACT-SUBSET-SUM takes an input set S = {x_1, x_2 , . . . x_n} and a target value t.

- This procedure iteratively computes L_i, the list of sums of all subsets of {$x_{1\,...}x_i$} that do not exceed t, and then it returns the maximum value in L_n.

The Subset-Sum Problem

An exponential-time exact algorithm



```
EXACT-SUBSET-SUM$(S, t)$
$n = |S|$
$L_0 = \langle 0 \rangle$
**for** $i = 1$ **to** $n$
    $L_i =$ MERGE-LISTS$(L_{i-1}, L_{i-1} + x_i)$
    remove from $L_i$ every element that is greater than $t$
**return** the largest element in $L_n$
```

- We also use an auxiliary procedure MERGE-LISTS(L, L’), which returns the sorted list that is the merge of its two sorted input lists L and L’ with duplicate values removed.
- MERGE-LISTS runs in O(|L| + |L’|) time.

The Subset-Sum Problem

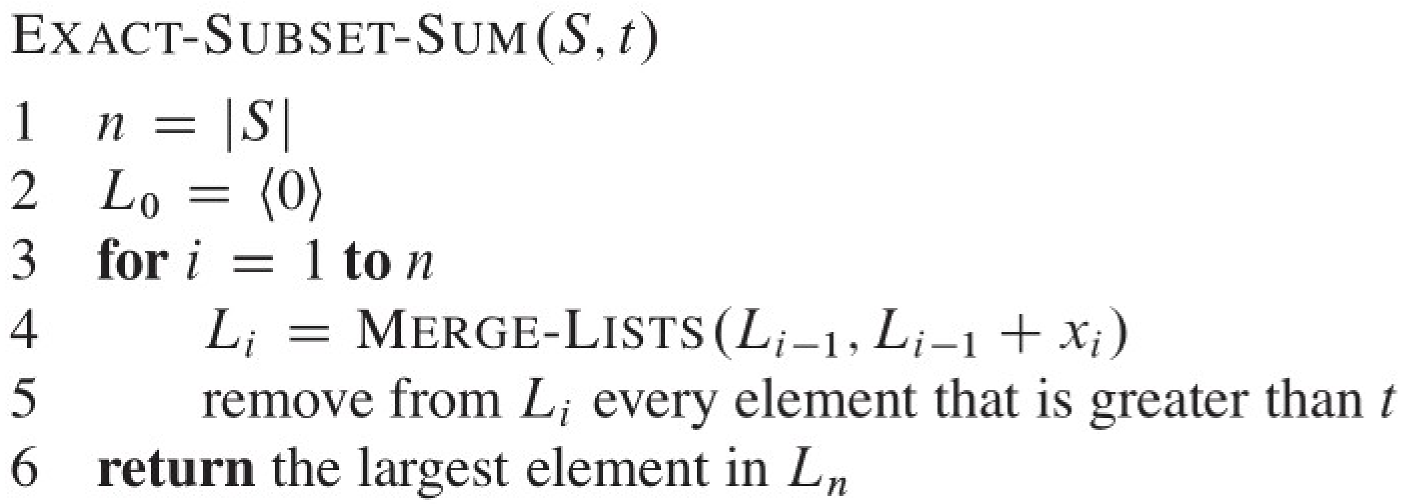
An exponential-time exact algorithm

```
EXACT-SUBSET-SUM($S, t$)
1  $n = |S|$
2  $L_0 = \langle 0 \rangle$
3  for $i = 1$ to $n$
4      $L_i$ = MERGE-LISTS($L_{i-1}, L_{i-1} + x_i$)
5      remove from $L_i$ every element that is greater than $t$
6  return the largest element in $L_n$
```

- Suppose, S = {104, 102, 201, 101} and t = 308
- n = |S| = 4
- L_0 = (0)

The Subset-Sum Problem

An exponential-time exact algorithm



```
EXACT-SUBSET-SUM$(S, t)$
1  $n = |S|$
2  $L_0 = \langle 0 \rangle$
3  for $i = 1$ to $n$
4      $L_i = $ MERGE-LISTS$(L_{i-1}, L_{i-1} + x_i)$
5      remove from $L_i$ every element that is greater than $t$
6  return the largest element in $L_n$
```

- Suppose, S = {104, 102, 201, 101} and t = 308
- i = 1

$L_0 = (0)$

- L_1 = MERGE-LISTS(L_0 , $L_0 + x_1$) = MERGE-LISTS((0), (0+104)) = (0, 104)

The Subset-Sum Problem

An exponential-time exact algorithm

```
EXACT-SUBSET-SUM$(S, t)$
1  $n = |S|$
2  $L_0 = \langle 0 \rangle$
3  for $i = 1$ to $n$
4      $L_i = $ MERGE-LISTS$(L_{i-1}, L_{i-1} + x_i)$
5      remove from $L_i$ every element that is greater than $t$
6  return the largest element in $L_n$
```

- Suppose, S = {104, 102, 201, 101} and t = 308
- i = 2

$L_1 = (0, 104)$

- L_2 = MERGE-LISTS(L_1 , $L_1 + x_2$) = MERGE-LISTS((0, 104), (102, 206)) = (0, 102, 104, 206)

The Subset-Sum Problem

An exponential-time exact algorithm

```
EXACT-SUBSET-SUM($S, t$)
1  $n = |S|$
2  $L_0 = \langle 0 \rangle$
3  for $i = 1$ to $n$
4      $L_i$ = MERGE-LISTS($L_{i-1}, L_{i-1} + x_i$)
5      remove from $L_i$ every element that is greater than $t$
6  return the largest element in $L_n$
```

- Suppose, S = {104, 102, 201, 101} and t = 308
- i = 3

  L_2 = (0, 102, 104, 206)
- L_3 = MERGE-LISTS(L_2 , $L_2 + x_3$)

  = MERGE-LISTS((0, 102, 104, 206), (201, 303, 305, 407))

  = (0, 102, 104, 201, 206, 303, 305, 407)
- 407 is removed from L_3 as it is greater than 308.

  Thus, L_3 = (0, 102, 104, 201, 206, 303, 305)

The Subset-Sum Problem

An exponential-time exact algorithm

```
EXACT-SUBSET-SUM($S, t$)
1  $n = |S|$
2  $L_0 = \langle 0 \rangle$
3  for $i = 1$ to $n$
4      $L_i$ = MERGE-LISTS($L_{i-1}, L_{i-1} + x_i$)
5      remove from $L_i$ every element that is greater than $t$
6  return the largest element in $L_n$
```

- Suppose, S = {104, 102, 201, 101} and t = 308
- i = 4

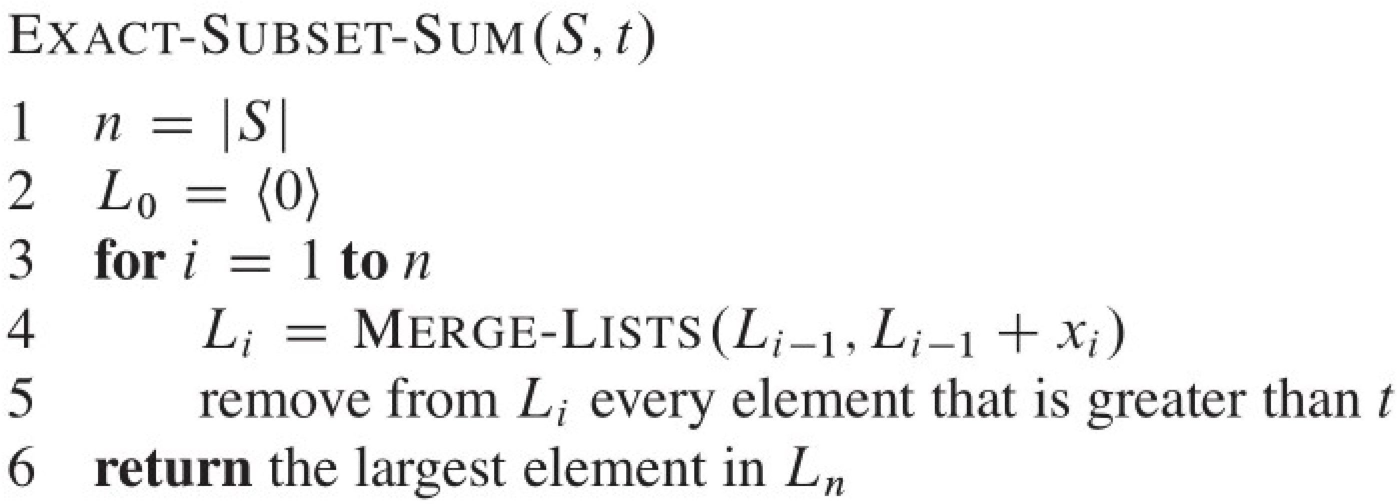
  L_3 = (0, 102, 104, 201, 206, 303, 305)
- L_4 = MERGE-LISTS(L_3 , $L_3 + x_4$)
  - = MERGE-LISTS((0, 102, 104, 201, 206, 303, 305), (101, 203, 205, 302, 307, 404, 406))

    = (0, 101, 102, 104, 201, 203, 205, 206, 302, 303, 305, 307, 404, 406)
- 404 and 406 are removed from L_4 as it is greater than 308.

  Thus, L_4 = (0, 101, 102, 104, 201, 203, 205, 206, 302, 303, 305, 307)

The Subset-Sum Problem

An exponential-time exact algorithm



```
EXACT-SUBSET-SUM(S, t)
1  $n = |S|$
2  $L_0 = \langle 0 \rangle$
3  for $i = 1$ to $n$
4      $L_i$ = MERGE-LISTS($L_{i-1}, L_{i-1} + x_i$)
5      remove from $L_i$ every element that is greater than $t$
6  return the largest element in $L_n$
```

$L_4 = (0, 101, 102, 104, 201, 203, 205, 206, 302, 303, 305, 307)$

- Therefore, the algorithm will return 307.
- Since the length of L_i can be as much as 2^i, EXACT-SUBSET-SUM is an exponential-time algorithm in general.

The Subset-Sum Problem

A fully polynomial-time approximation scheme

- We can derive a fully polynomial-time approximation scheme for the subset-sum problem by “trimming” each list L_i after it is created.
- The idea behind trimming is that if two values in L are close to each other, then since we want just an approximate solution, we do not need to maintain both of them explicitly.

The Subset-Sum Problem

A fully polynomial-time approximation scheme

- More precisely, we use a trimming parameter δ such that 0 < δ < 1.
- When we trim a list L by, we remove as many elements from L as possible, in such a way that if L_0 is the result of trimming L, then for every element y that was removed from L, there is an element still in L_0 that approximates y, that is

$$\frac{y}{1+\delta} \leq z \leq y$$

- We can think of such a **z** as “representing” y in the new list L’.

The Subset-Sum Problem

A fully polynomial-time approximation scheme

- For example, if $\delta = 0.1$ and $L = \{10, 11, 12, 15, 20, 21, 22, 23, 24, 29\}$ then we can trim L to obtain

  $$L' = \{10, 12, 15, 20, 23, 29\}$$

  where the deleted value 11 is represented by 10, the deleted values 21 and 22 are represented by 20, and the deleted value 24 is represented by 23.

- Because every element of the trimmed version of the list is also an element of the original version of the list, trimming can dramatically decrease the number of elements kept while keeping a close (and slightly smaller) representative value in the list for each deleted element.

The Subset-Sum Problem

A fully polynomial-time approximation scheme

- The following procedure trims list L = $\{y_1, y_2 . . . . y_m\}$ in time Θ(m), given L and δ, and assuming that L is sorted into monotonically increasing order. The output of the procedure is a trimmed, sorted list.

```
TRIM($L, \delta$)
let $m$ be the length of $L$
$L' = \langle y_1 \rangle$
$last = y_1$
for $i = 2$ to $m$
    if $y_i > last \cdot (1 + \delta)$        // $y_i \geq last$ because $L$ is sorted
        append $y_i$ onto the end of $L'$
        $last = y_i$
return $L'$
```

- The procedure scans the elements of L in monotonically increasing order. A number is appended onto the returned list L’ only if it is the first element of L or if it cannot be represented by the most recent number placed into L’.

The Subset-Sum Problem

A fully polynomial-time approximation scheme

- Given the procedure TRIM, we can construct our approximation scheme as follows. This procedure takes as input a set $S = \{x_1, x_2 \ldots x_n\}$of n integers (in arbitrary order), a target integer t, and an “approximation parameter” ε, where

$$\mathbf{0 < \varepsilon < 1}$$

- It returns a value z whose value is within a 1 + ε factor of the optimal solution.

The Subset-Sum Problem

A fully polynomial-time approximation scheme

```
APPROX-SUBSET-SUM$(S, t, \epsilon)$
$n = |S|$
$L_0 = \langle 0 \rangle$
**for** $i = 1$ **to** $n$
    $L_i = $ MERGE-LISTS$(L_{i-1}, L_{i-1} + x_i)$
    $L_i = $ TRIM$(L_i, \epsilon/2n)$
    remove from $L_i$ every element that is greater than $t$
let $z^*$ be the largest value in $L_n$
**return** $z^*$
```

The Subset-Sum Problem

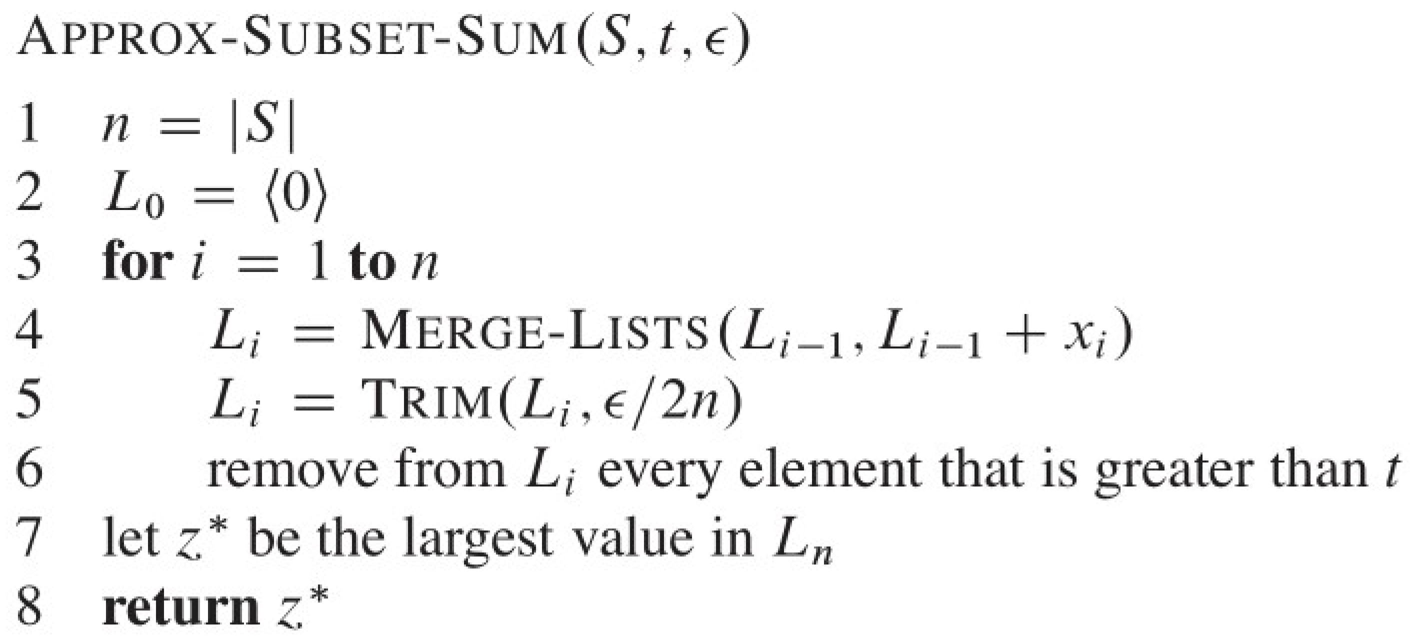
A fully polynomial-time approximation scheme

```
APPROX-SUBSET-SUM($S, t, \epsilon$)
$n = |S|$
$L_0 = \langle 0 \rangle$
for $i = 1$ to $n$
    $L_i$ = MERGE-LISTS($L_{i-1}, L_{i-1} + x_i$)
    $L_i$ = TRIM($L_i, \epsilon/2n$)
    remove from $L_i$ every element that is greater than $t$
let $z^*$ be the largest value in $L_n$
return $z^*$
```

- As an example, suppose we have the instance S = {104, 102, 201, 101} with t = 308 and ε = 0.40.
- The trimming parameter is δ = ε/8 = 0.05.

The Subset-Sum Problem

A fully polynomial-time approximation scheme



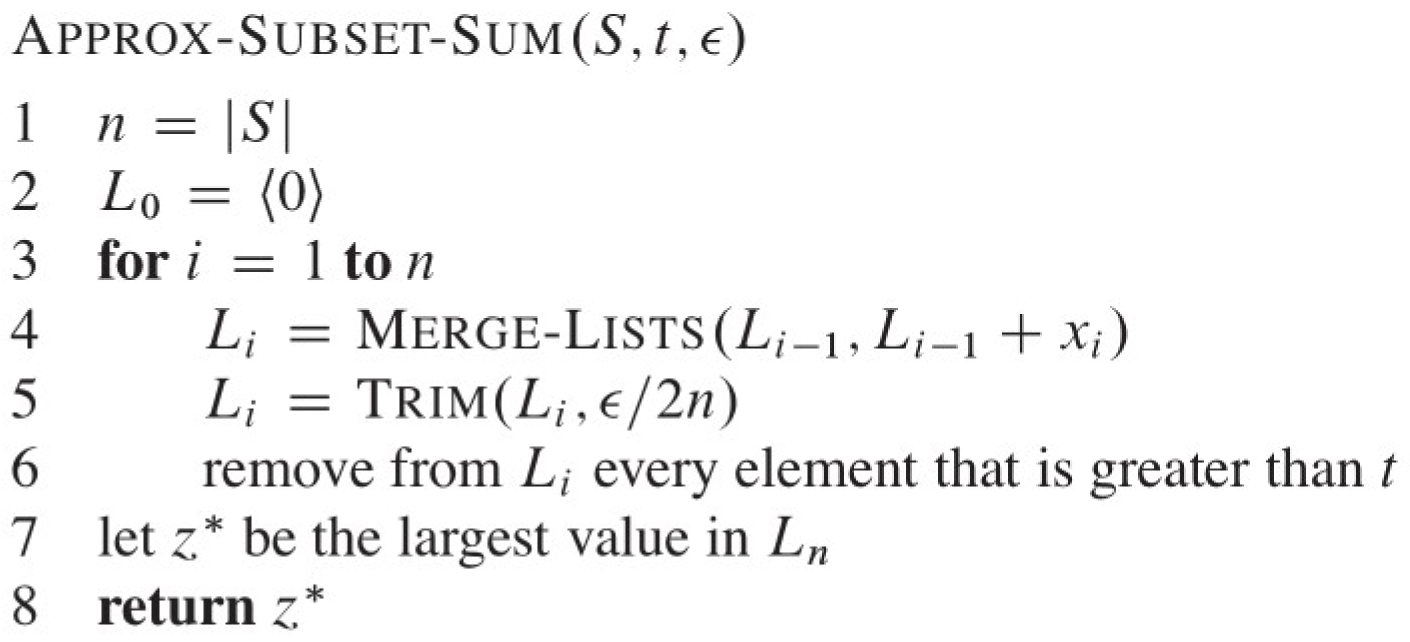
```
APPROX-SUBSET-SUM($S, t, \epsilon$)
1  $n = |S|$
2  $L_0 = \langle 0 \rangle$
3  for $i = 1$ to $n$
4      $L_i$ = MERGE-LISTS($L_{i-1}, L_{i-1} + x_i$)
5      $L_i$ = TRIM($L_i, \epsilon/2n$)
6      remove from $L_i$ every element that is greater than $t$
7  let $z^*$ be the largest value in $L_n$
8  return $z^*$
```

- As an example, suppose we have the instance S = {104, 102, 201, 101} with t = 308 and ε = 0.40.
- The trimming parameter is δ = ε/8 = 0.05.

- Line 2: L_0 = (0),

The Subset-Sum Problem

A fully polynomial-time approximation scheme



```
APPROX-SUBSET-SUM$(S, t, \epsilon)$
1  $n = |S|$
2  $L_0 = \langle 0 \rangle$
3  for $i = 1$ to $n$
4      $L_i$ = MERGE-LISTS$(L_{i-1}, L_{i-1} + x_i)$
5      $L_i$ = TRIM$(L_i, \epsilon/2n)$
6      remove from $L_i$ every element that is greater than $t$
7  let $z^*$ be the largest value in $L_n$
8  return $z^*$
```

- As an example, suppose we have the instance S = {104, 102, 201, 101} with t = 308 and ε = 0.40.
- The trimming parameter is δ = ε/8 = 0.05.

$L_0 = (0)$,

- Line 4: $L_1 = (0, 104)$,
- Line 5: $L_1 = (0, 104)$,
- Line 6: $L_1 = (0, 104)$,

The Subset-Sum Problem

A fully polynomial-time approximation scheme

```
APPROX-SUBSET-SUM($S, t, \epsilon$)

1  $n = |S|$
2  $L_0 = \langle 0 \rangle$
3  for $i = 1$ to $n$
4      $L_i$ = MERGE-LISTS($L_{i-1}, L_{i-1} + x_i$)
5      $L_i$ = TRIM($L_i, \epsilon/2n$)
6      remove from $L_i$ every element that is greater than $t$
7  let $z^*$ be the largest value in $L_n$
8  return $z^*$
```

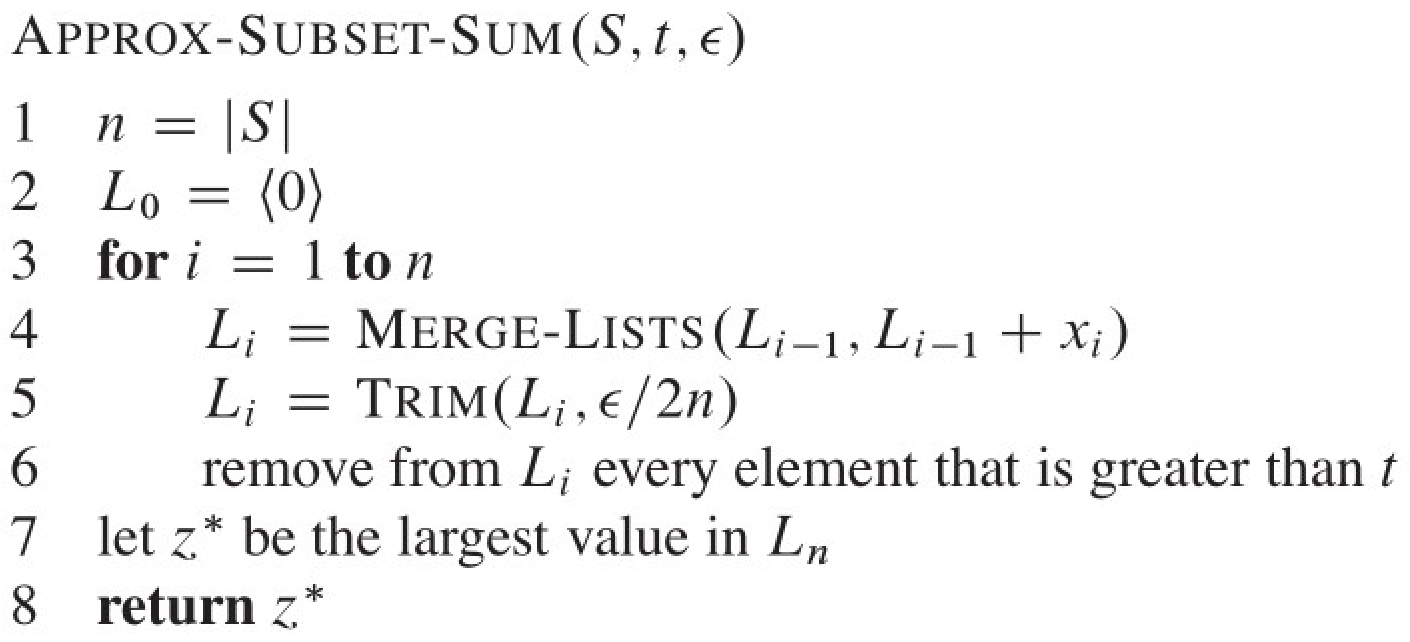
- As an example, suppose we have the instance S = {104, 102, 201, 101} with t = 308 and ε = 0.40.
- The trimming parameter is δ = ε/8 = 0.05.

$L_1 = (0, 104)$,

- Line 4: $L_2 = (0, 102, 104, 206)$,
- Line 5: $L_2 = (0, 102, 206)$,
- Line 6: $L_2 = (0, 102, 206)$,

The Subset-Sum Problem

A fully polynomial-time approximation scheme



```
APPROX-SUBSET-SUM$(S, t, \epsilon)$
1  $n = |S|$
2  $L_0 = \langle 0 \rangle$
3  for $i = 1$ to $n$
4      $L_i =$ MERGE-LISTS$(L_{i-1}, L_{i-1} + x_i)$
5      $L_i =$ TRIM$(L_i, \epsilon/2n)$
6      remove from $L_i$ every element that is greater than $t$
7  let $z^*$ be the largest value in $L_n$
8  return $z^*$
```

- As an example, suppose we have the instance S = {104, 102, 201, 101} with t = 308 and ε = 0.40.
- The trimming parameter is δ = ε/8 = 0.05.

$L_2 =$ (0, 102, 206)

- Line 4: $L_3=$ (0, 102, 201, 206, 303, 407)
- Line 5: $L_3 =$ (0, 102, 201, 303, 407)
- Line 6: $L_3 =$ (0, 102, 201, 303)

The Subset-Sum Problem

A fully polynomial-time approximation scheme

```
APPROX-SUBSET-SUM$(S, t, \epsilon)$
1  $n = |S|$
2  $L_0 = \langle 0 \rangle$
3  for $i = 1$ to $n$
4      $L_i$ = MERGE-LISTS$(L_{i-1}, L_{i-1} + x_i)$
5      $L_i$ = TRIM$(L_i, \epsilon/2n)$
6      remove from $L_i$ every element that is greater than $t$
7  let $z^*$ be the largest value in $L_n$
8  return $z^*$
```

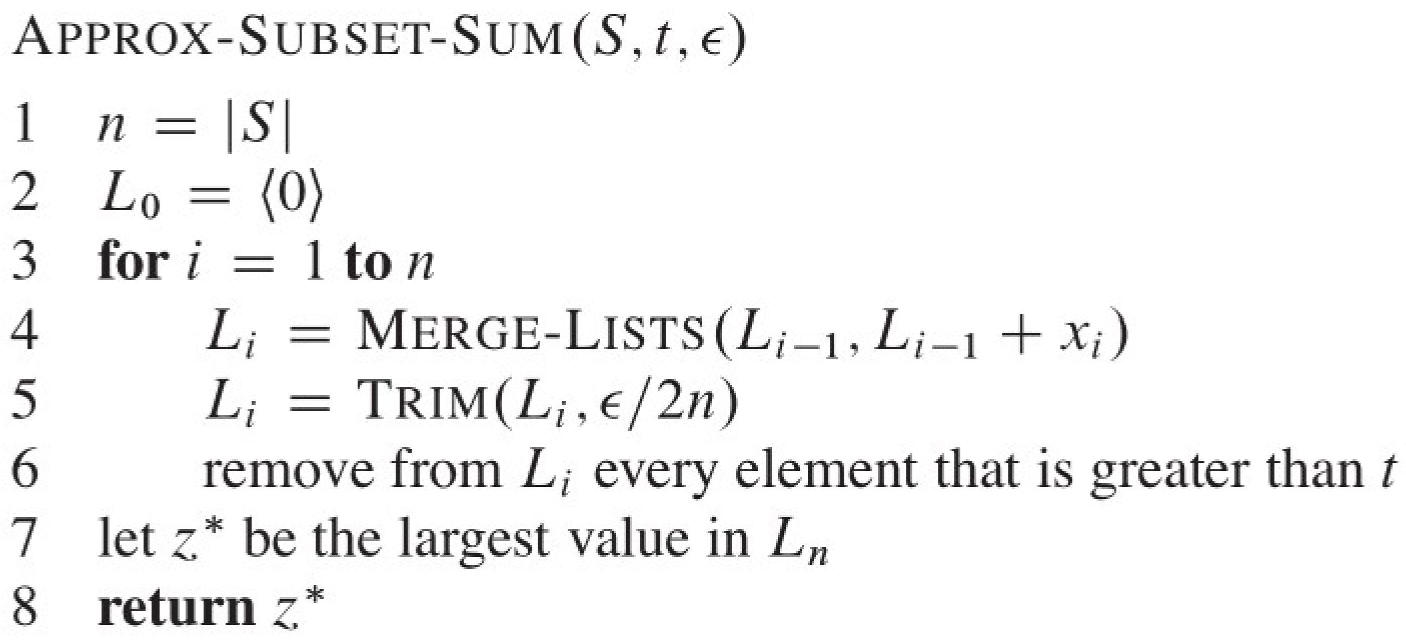
- As an example, suppose we have the instance S = {104, 102, 201, 101} with t = 308 and ε = 0.40.
- The trimming parameter is δ = ε/8 = 0.05.

L_3 = (0, 102, 201, 303)

- Line 4: L_4= (0, 101, 102, 201, 203, 302, 303, 404)
- Line 5: L_4 = (0, 101, 201, 302, 404)
- Line 6: L_4 = (0, 101, 201, 302)

The Subset-Sum Problem

A fully polynomial-time approximation scheme



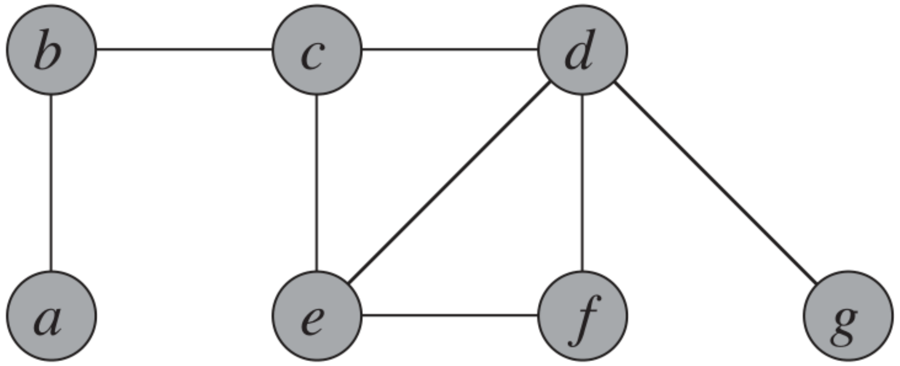
```
APPROX-SUBSET-SUM($S, t, \epsilon$)
$n = |S|$
$L_0 = \langle 0 \rangle$
for $i = 1$ to $n$
    $L_i$ = MERGE-LISTS($L_{i-1}, L_{i-1} + x_i$)
    $L_i$ = TRIM($L_i, \epsilon/2n$)
    remove from $L_i$ every element that is greater than $t$
let $z^*$ be the largest value in $L_n$
return $z^*$
```

L_4 = (0, 101, 201, 302)

- The algorithm returns z* = 302 as its answer, which is well within ε = 40% of the optimal answer 307 = 104 + 102 + 101; in fact, it is within 2%.

Review Questions

1) Why are approximation algorithms important?Write an algorithm that computes the approximate solution for the vertex cover problem and apply it on the following graph.



2) How can we obtain an approximate solution for the traveling salesman problem? Explain with an example.

3) What do you mean by an approximation algorithm? Explain the approximation algorithm for the set cover problem.

4) Give an approximation algorithm for the subset sum problem that works in polynomial time and show it's working with an example.