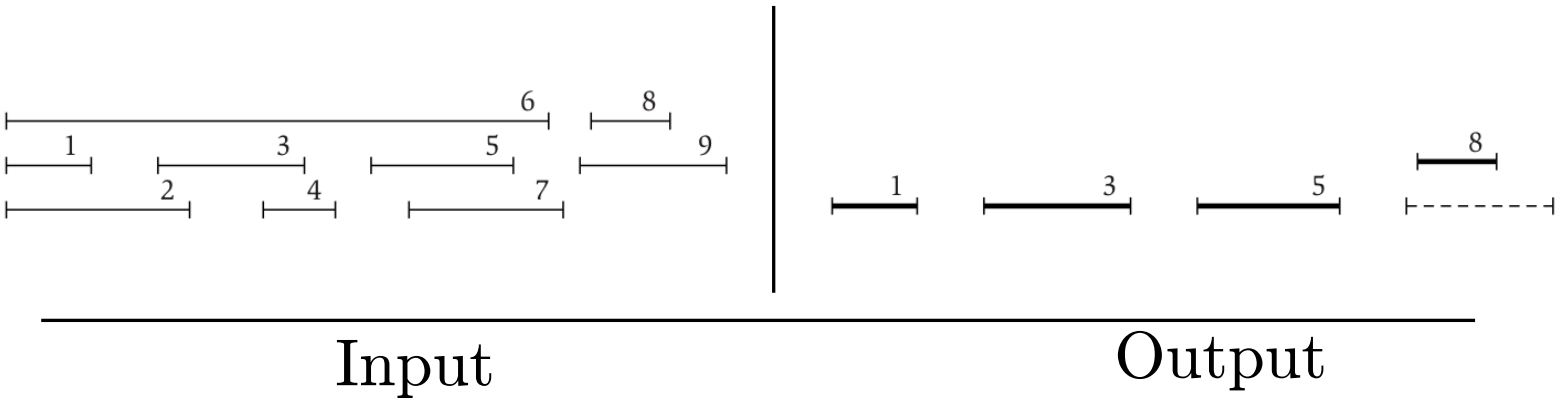
Chapter 4

Greedy Algorithms

Greedy Algorithms

- Algorithms for optimization problems typically go through a sequence of steps, with a set of choices at each step.
- For many optimization problems, using dynamic programming to determine the best choices is overkill.
- A greedy algorithm always makes the choice that looks best at the moment with the hope that this choice will lead to a globally optimal solution.
- Greedy algorithms do not always yield optimal solutions, but for many problems they do.

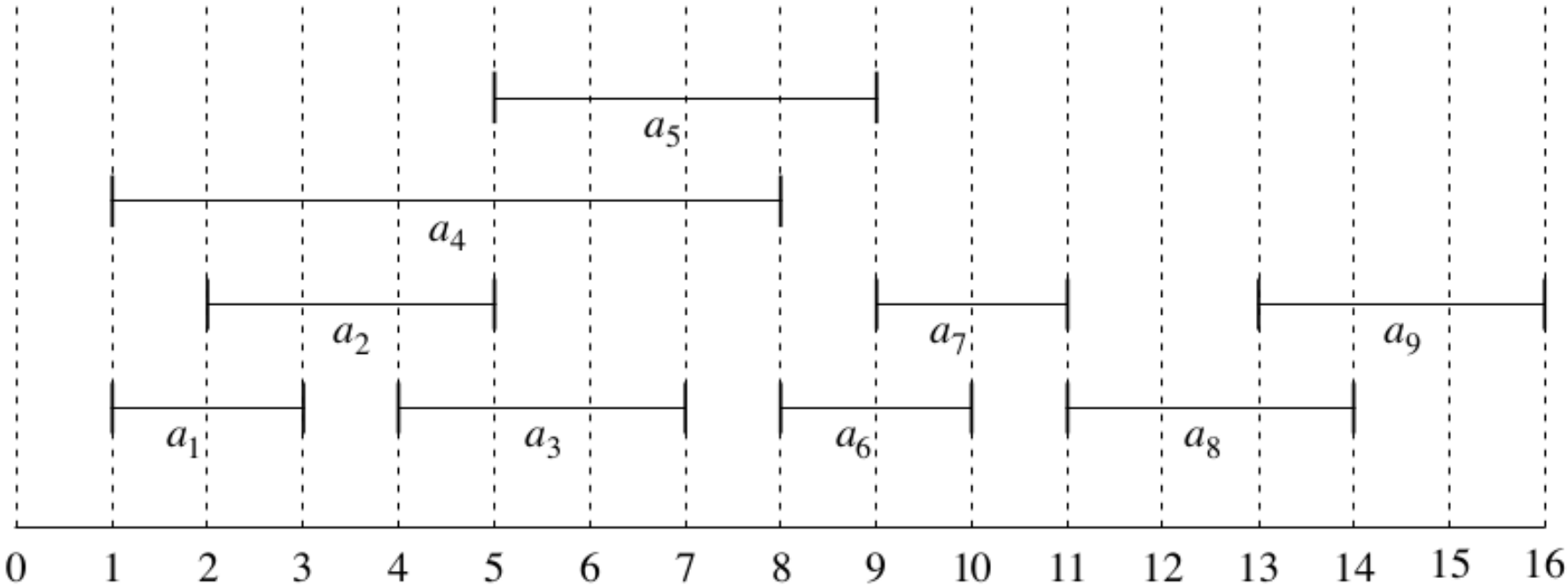
Activity-selection problem



Input description: a set $S = \{a_1, \cdots, a_n\}$ of n proposed *activities* that wish to use a resource. Each activity a_i has a start time s_i and a finish time f_i , where $0 \leq s_i < f_i < \infty$. If selected, activity a_i takes place during the half-open time interval $[s_i, f_i)$. We assume activities are ordered such that $f_1 \leq f_2 \leq \cdots \leq f_n$.
Problem description: Select the largest possible set of nonoverlapping (mutually compatible) activities.

Example: S sorted by finish time

i	1	2	3	4	5	6	7	8	9
s_i	1	2	4	1	5	8	9	11	13
f_i	3	5	7	8	9	10	11	14	16



Maximum-size mutually compatible set: $\{a_1, a_3, a_6, a_8\}$. Not unique: also $\{a_2, a_5, a_7, a_9\}$.

A Dynamic Programming Approach:

Let us denote by S_{ij} the set of activities that start after activity a_i finishes and that finish before activity a_j starts. If we denote the size of an optimal solution for the set S_{ij} by $c[i,j]$, we have the following recursion:

$$c[i,j] = \begin{cases} 0, & S_{ij} = \Phi \\ \max_{i<k<j, a_k \in S_{ij}} \{c[i,k] + 1 + c[k,j]\}, & S_{ij} \neq \Phi \end{cases}$$

Making the greedy choice

What if we could choose an activity to add to our optimal solution without having to first solve all the subproblems?

- Intuition suggests that we should choose an activity that leaves the resource available for as many other activities as possible.
- Our intuition tells us, therefore, to choose the activity in S with the earliest finish time, since that would leave the resource available for as many of the activities that follow it as possible.
- If we make the greedy choice, we have only one remaining subproblem to solve:

  finding activities that start after a_1 finishes.

Note: Choosing the first activity to finish is not the only way to think of making a greedy choice for this problem. What are other choices?

▶ **One big question:** is our intuition correct? Is the greedy choice–in which we choose the first activity to finish–always part of some optimal solution?

Theorem

Consider any nonempty subproblem $S_k = \{a_i \in S, s_i \geq f_k\}$, and let a_m be an activity in S_k with the earliest finish time. Then a_m is included in some maximum-size subset of mutually compatible activities of S_k.

Proof.

Let A_k be a maximum-size subset of mutually compatible activities in S_k, and let a_j be the activity in A_k with the earliest finish time.

1- If $a_j = a_m$, done (a_m is in some maximum-size subset).

2- Otherwise, construct $A'_k = (A_k - \{a_j\}) \bigcup \{a_m\}$ (replace a_j by a_m).

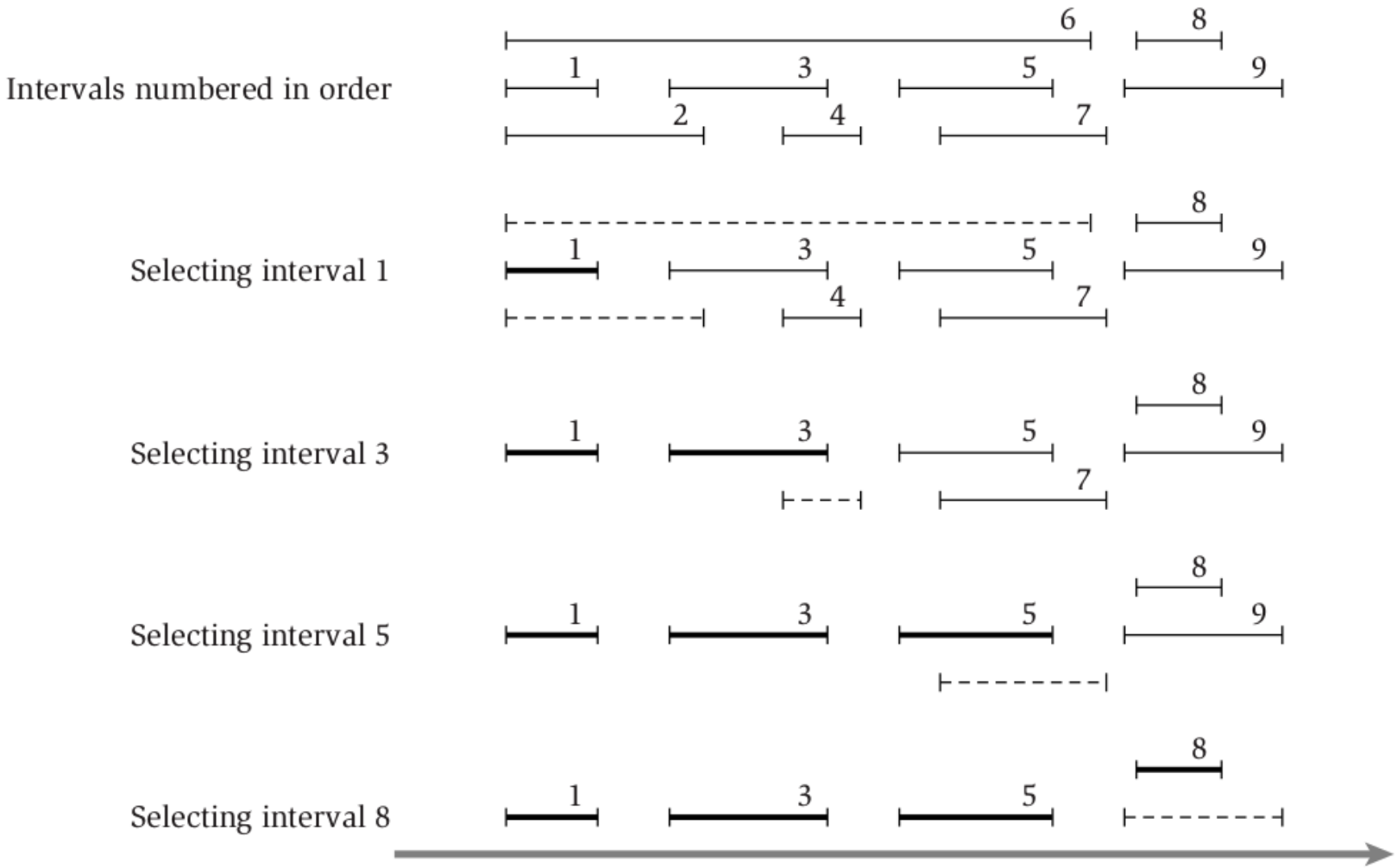
Claim: The activities in A'_k are disjoint

Proof: the activities in A_k are disjoint and a_j is the first activity in S_k to finish, and $f_m \leq f_j$ (so a_m doesn't overlap anything else in A'_k).

Since $|A'_k| = |A_k|$, and A_k is a maximum-size subset, so is A'_k.

□

Example


Intervals numbered in order
Selecting interval 1
Selecting interval 3
Selecting interval 5
Selecting interval 8
1
2
3
4
5
6
7
8
9

- Thus, we see that although we might be able to solve the activity-selection problem with dynamic programming, we don't need to.
- Instead, we can repeatedly choose the activity that finishes first, keep only the activities compatible with this activity, and repeat until no activities remain.

```
GREEDY-ACTIVITY-SELECTOR(s, f)
1  n = s.length
2  A = {a_1}
3  k = 1
4  for m = 2 to n
5      if s[m] ≥ f[k]
6          A = A ∪ {a_m}
7          k = m
8  return A
```

Recall that activities are ordered based on their finish times!

- Time complexity is $\Theta(n)$

- The greedy algorithm does not need to work bottom-up, like a table-based dynamic-programming algorithm.
- Instead, it can work top-down, choosing an activity to put into the optimal solution and then solving the subproblem of choosing activities from those that are compatible with those already chosen.

Greedy algorithms typically have this top-down design: make a choice and then solve a subproblem, rather than the bottom-up technique of solving subproblems before making a choice

Elements of the greedy strategy

1. Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
2. Prove that there is always an optimal solution to the original problem that makes the greedy choice, so that the greedy choice is always safe.
3. Demonstrate optimal substructure by showing that, having made the greedy choice, what remains is a subproblem with the property that if we combine an optimal solution to the subproblem with the greedy choice we have made, we arrive at an optimal solution to the original problem.

Beneath every greedy algorithm, there is almost always a more cumbersome dynamic-programming solution.

Greedy & DP –Similarity:
A problem exhibits **optimal substructure** if an optimal solution to the problem contains within it optimal solutions to subproblems. This property is a **key ingredient** of assessing the applicability of dynamic programming as well as greedy algorithms.

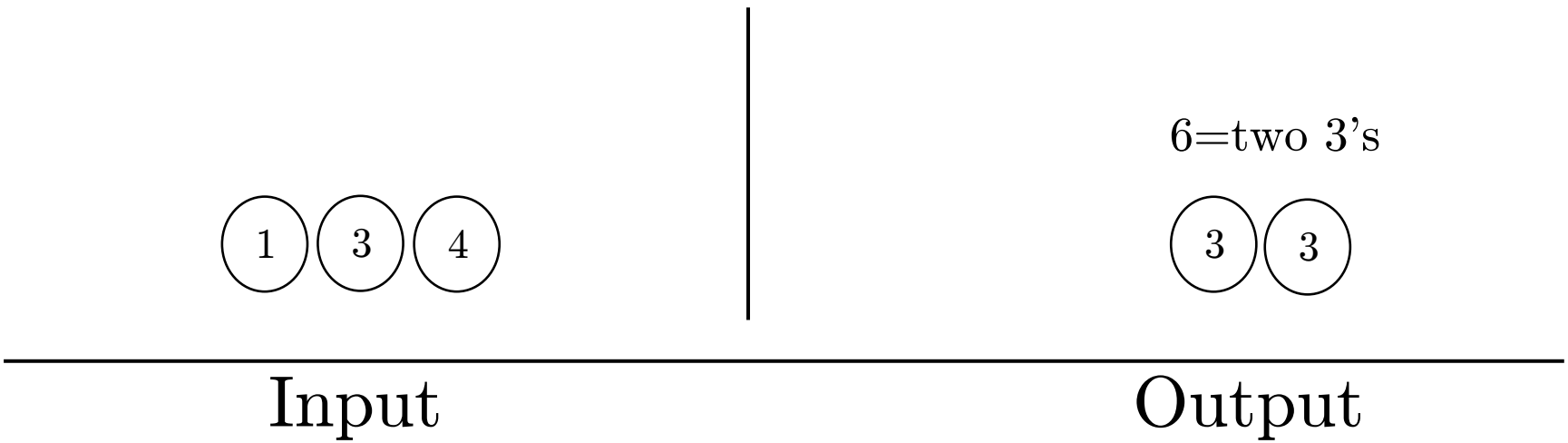
Greedy & DP –Differences:

- In **dynamic programming**, we make a choice at each step, but the choice usually depends on the solutions to subproblems.
- In a **greedy algorithm**, we make whatever choice seems best at the moment and then solve the subproblem that remains. The choice made by a greedy algorithm may depend on choices so far, but it cannot depend on any future choices or on the solutions to subproblems. Thus, unlike dynamic programming, which solves the subproblems before making the first choice, a greedy algorithm makes its first choice before solving any subproblems.
- A dynamic-programming algorithm proceeds bottom up, whereas a greedy strategy usually progresses in a top-down fashion, making one greedy choice after another, reducing each given problem instance to a smaller one.

Typical Structure of Greedy Algorithms:

- Selection procedure
- Feasibility check
- Solution check

Change-making problem



Input description: A set of coins of denominations $d_1 < d_2 < ... < d_m$. Assume availability of unlimited quantities of coins for each of the m denominations $d_1 < d_2 < ... < d_m$ where $d_1 = 1$.
Problem description: Give change for amount n using the minimum number of coins

In the last chapter, We have discussed about the DP approach to solve the change-making problem. A greedy algorithm to solve the same problem is as follows.

```
While (there are more coins and the instance is not solved) {
    Grab the largest remaining coin;                          // selection procedure
    If (adding the coin makes the change exceed the           // feasibility check
                                        amount owed)
        reject the coin;
    else
        add the coin to the change;
    If (the total value of the change equals the              // solution check
                                the amount owed)
        the instance is solved;
}
```

Does it result in an optimal solution?
Answer: Depends on the set of coins

For example, the widely used coin denominations in the United States are 100 cents (1$), 25 cents (quarter), 10 cents (dime), 5 cents (nickel), and 1 cent (penny). And we have the following theorem:

Theorem

The greedy algorithm is optimal for U.S. coins $\{1, 5, 10, 25, 100\}$.

Proof:

First consider the following properties of any optimal solution (for U.S. coin denominations)

- Number of pennies ≤ 4.
  Pf. Replace 5 pennies with 1 nickel.
- Number of nickels ≤ 1.
- Number of quarters ≤ 3.
- Number of nickels + number of dimes ≤ 2.
  Pf.
  - Recall: nickels ≤ 1.
  - Replace 3 dimes and 0 nickels with 1 quarter and 1 nickel;
  - Replace 2 dimes and 1 nickel with 1 quarter.

Now, consider optimal way to change $c_k \leq x < c_{k+1}$: greedy takes coin k.

► We claim that any optimal solution must take coin k.

- if not, it needs enough coins of type $c_1, \cdots, c_{k-1}$ to add up to x. But table below indicates no optimal solution can do this

k	c_k	all optimal solutions must satisfy	max value of coin denominations $c_1, c_2, \ldots, c_{k-1}$ in any optimal solution
1	1	$P \leq 4$	–
2	5	$N \leq 1$	4
3	10	$N + D \leq 2$	4 + 5 = 9
4	25	$Q \leq 3$	20 + 4 = 24
5	100	*no limit*	75 + 24 = 99

► Therefore, we cannot change x optimally without using c_k. As a result, the problem reduces to coin-changing $x - c_k$ cents, which, by induction, is optimally solved by greedy algorithm.

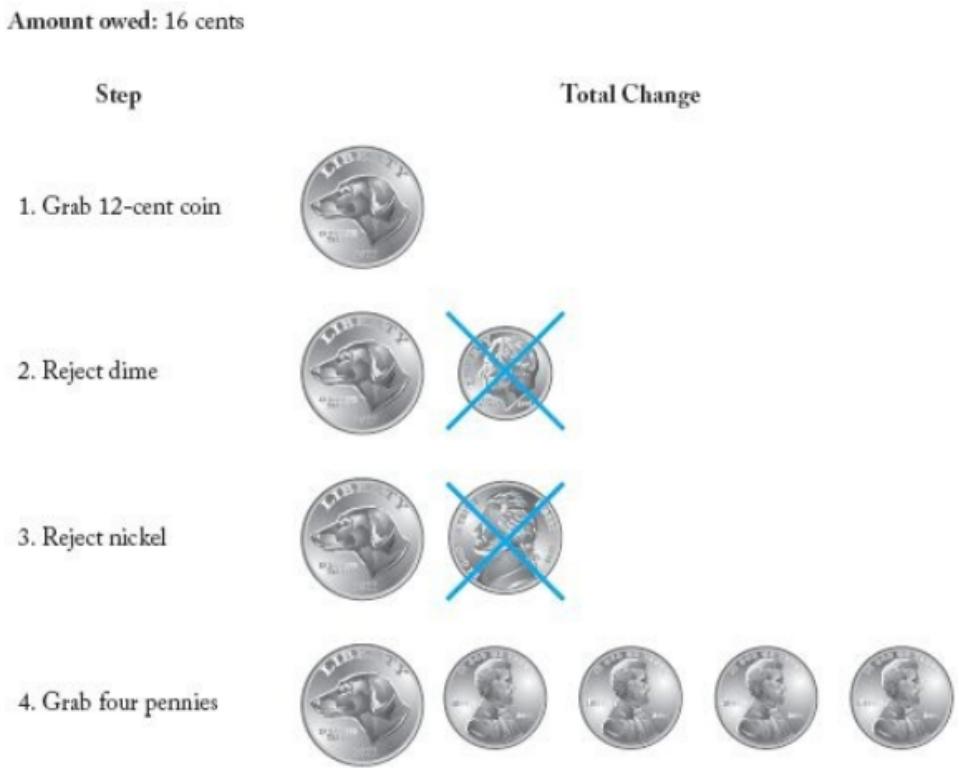
□

Example:

Amount owed: 36 cents

Step	Total Change
1. Grab quarter	
2. Grab first dime	
3. Reject second dime	
4. Reject nickel	
5. Grab penny	

But the greedy algorithm is not optimal if a 12-cent coin is included.

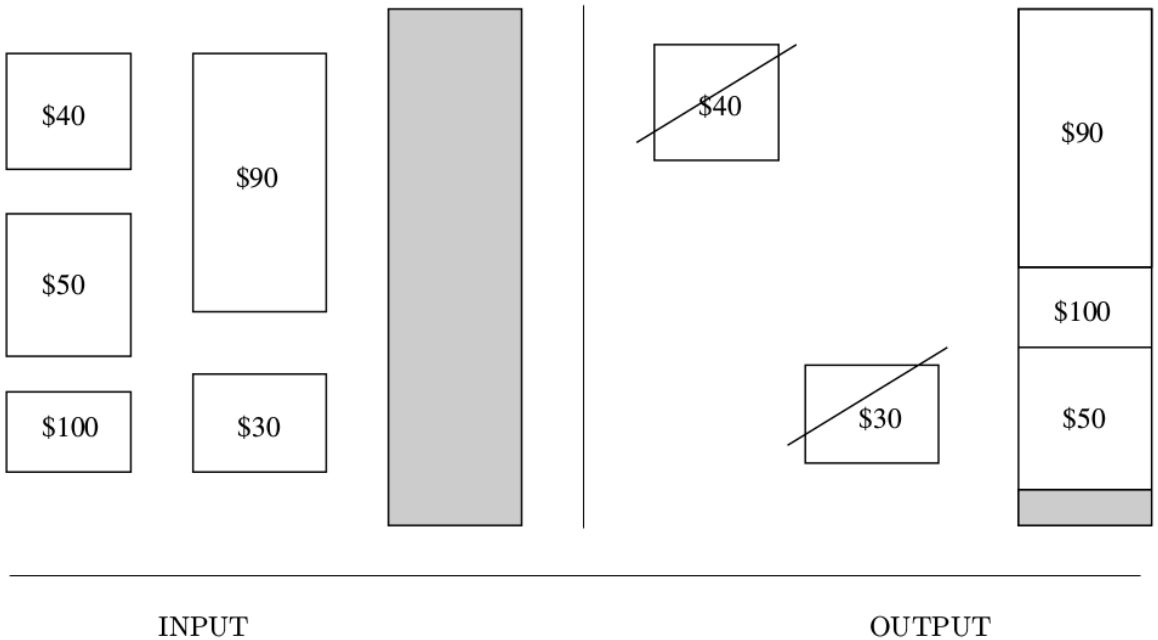


the greedy solution contains five coins, whereas the optimal solution, which consists of a dime, nickel, and penny, contains only three coins.

Discussion Question:

Show that the greedy approach always finds an optimal solution for the Change problem when the coins are in the denominations $D^0, D^1, D^2, ..., D^i$ for some integers $i > 0$ and $D > 0$.

Fractional Knapsack



Input description: n items of known weights $w_1, \cdots, w_n$ and values $v_1, \cdots, v_n$ and a knapsack of capacity W

Problem description: find the most valuable subset of the items that fit into the knapsack. **We are allowed to subdivide objects**.

Greedy Algorithms for the fractional Knapsack Problem

We can think of several greedy approaches to this problem:

- Highest Value First
  Sometimes fails: $W = 30$, and $(v_1 = \$10, W_1 = 25$ lb$)$, $(v_2 = \$9, W_2 = 10$ lb$)$, $(v_3 = \$9, W_3 = 10$ lb$) \rightarrow$ this greedy strategy would yield only a profit of _____,whereas the optimal solution is \$22
- Lightest First
  It also fails in instances where the light items have small profits compared with their weights.
- Highest Value-per-unit-Weight First
  we order the items in nonincreasing order according to value per unit weight, and select them in sequence.

Greedy algorithm for the continuous knapsack problem

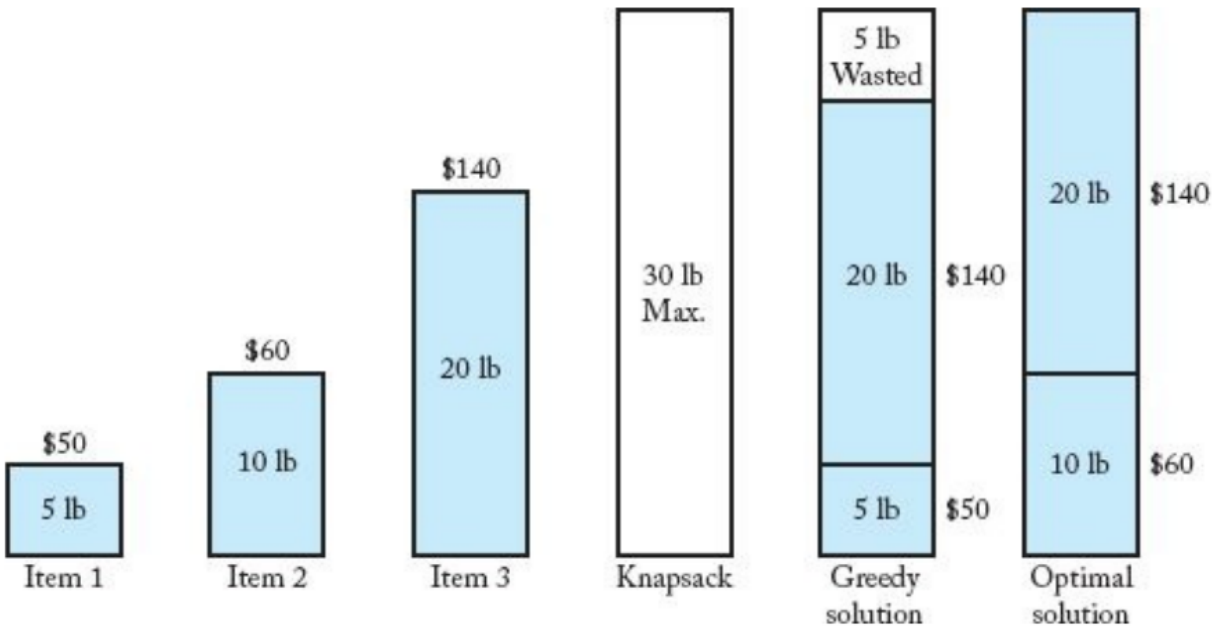
Step 1 Compute the value-to-weight ratios v_i/w_i, $i = 1, \ldots, n$, for the items given.

Step 2 Sort the items in nonincreasing order of the ratios computed in Step 1. (Ties can be broken arbitrarily.)

Step 3 Repeat the following operation until the knapsack is filled to its full capacity or no item is left in the sorted list: if the current item on the list fits into the knapsack in its entirety, take it and proceed to the next item; otherwise, take its largest fraction to fill the knapsack to its full capacity and stop.

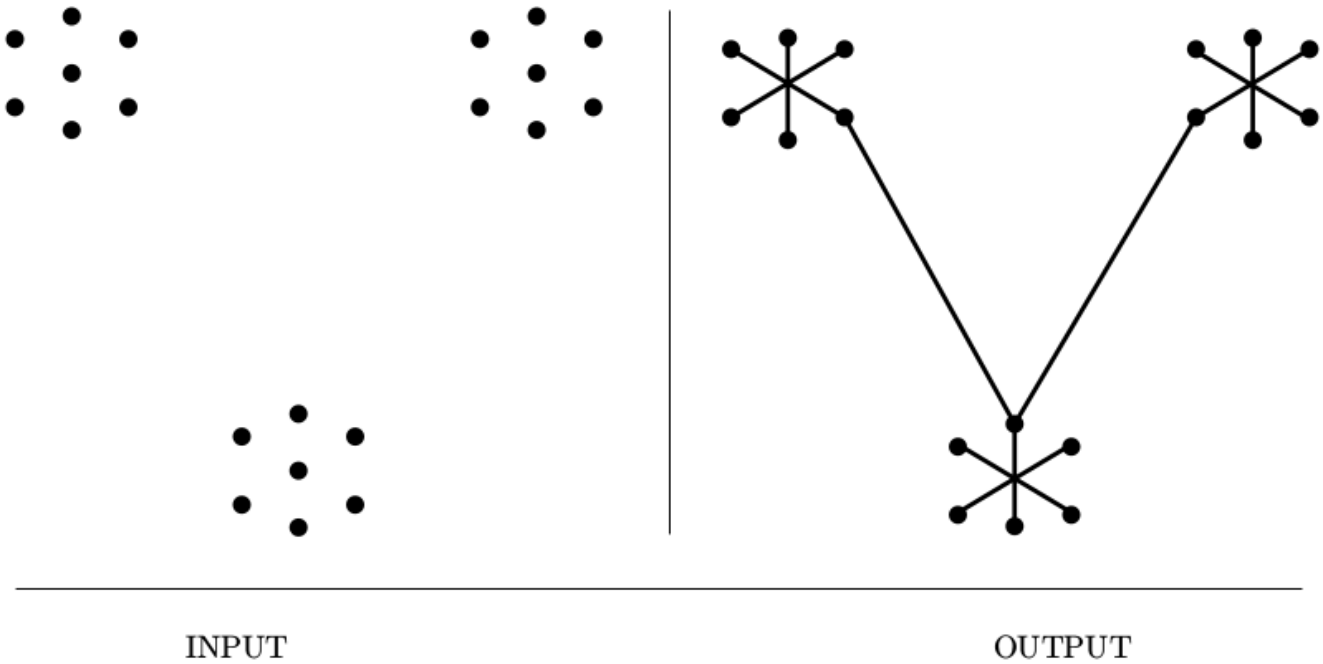
The items are ordered according to their efficiency in using the knapsack's capacity. If the first item on the sorted list has weight w_1 and value v_1, no solution can use w_1 units of capacity with a higher payoff than v_1. If we cannot fill the knapsack with the first item or its fraction, we should continue by taking as much as we can of the second-most efficient item, and so on. A formal rendering of this proof idea is left as an exercise.

In the last chapter, We have discussed about the DP approach to solve the 0/1 knapsack problem exactly (how efficient was that?). The greedy algorithm fails to find the correct solution for the same problem.



$$item_1 = \frac{\$50}{5} = \$10,\ item_2 = \frac{\$60}{10} = \$6, item_3 = \frac{\$140}{20} = \$7$$

Minimum Spanning Tree



Input description: A graph $G = (V, E)$ with weighted edges ($|V| = n, |E| = m, n \leq m \leq n(n-1)/2$)

Problem description: The minimum weight subset of edges $E' \subset E$ that form a tree on V

There are two famous greedy algorithms for the MST problem:

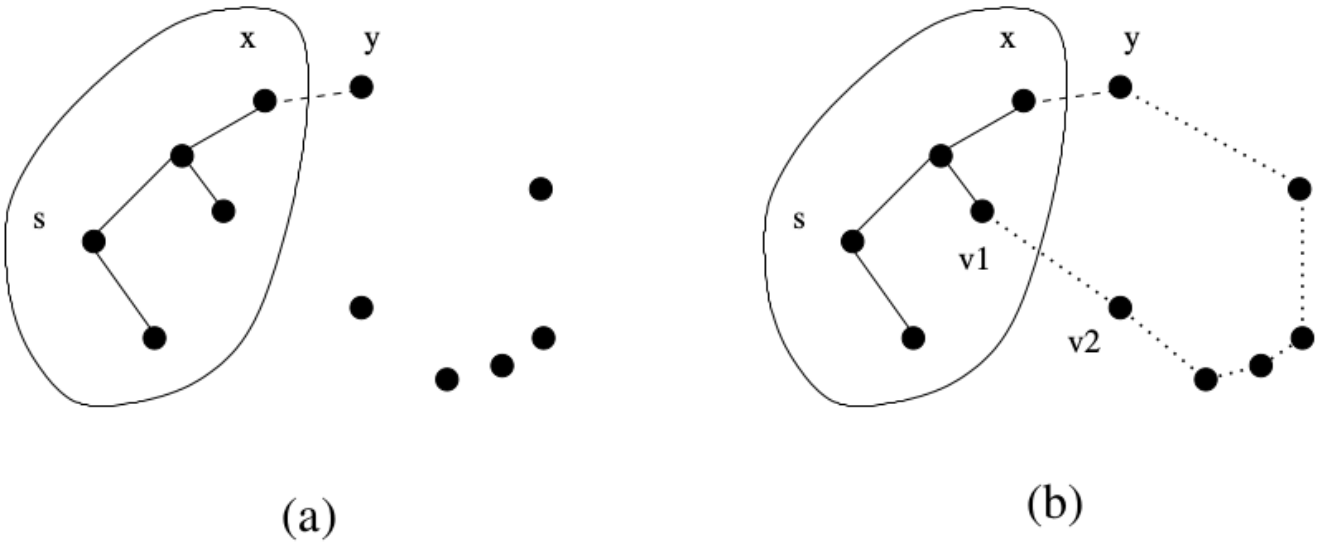
- Prim's Algorithm
- Kruskal's Algorithm

Prim's Algorithm

Prim's minimum spanning tree algorithm starts from one vertex and grows the rest of the tree one edge at a time until all vertices are included.
In each step, Prim's algorithm greedily selects the smallest weight edge that will enlarge the number of vertices in the tree.

Prim-MST(G)
    Select an arbitrary vertex s to start the tree from.
    While (there are still nontree vertices)
        Select the edge of minimum weight between a tree and nontree vertex
        Add the selected edge and vertex to the tree T_{prim}.

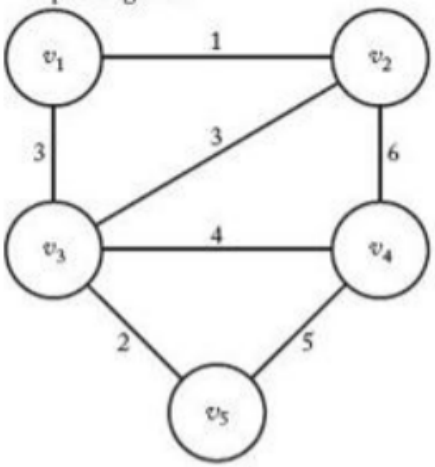
Proof by Contradiction ....



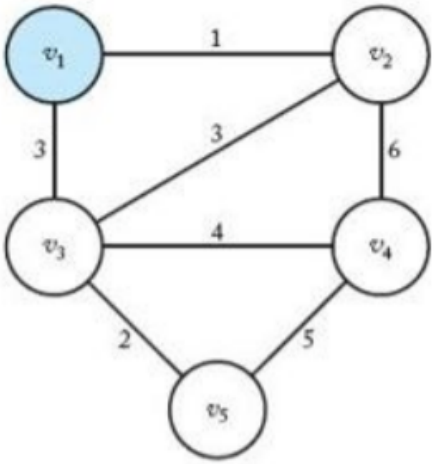
Let (x, y) be the first fatal error. Then there would be another path from x and y in the optimum tree like the one including (v_1, v_2) in figure b. But we can exchange (v_1, v_2) and (x, y) and be at least as good as the optimal (since $d(x, y) \leq d(v_1, v_2)$). So the Prim has not made a fatal error! □

Example

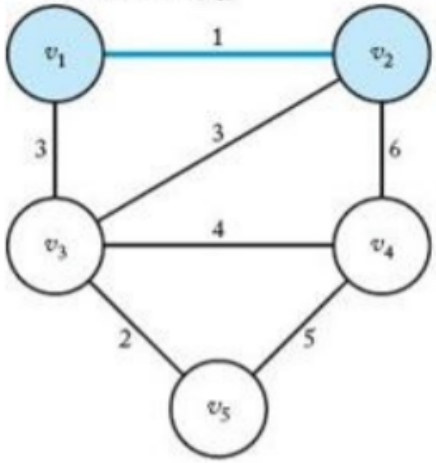
Determine a minimum spanning tree.



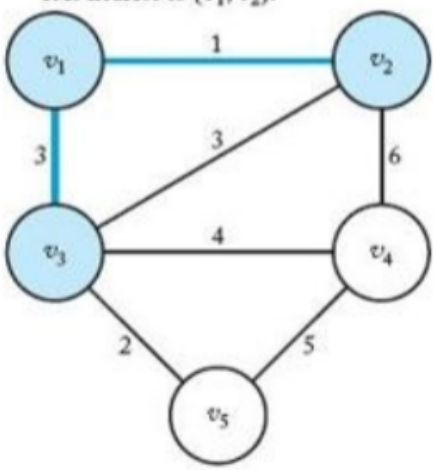
1. Vertex v_1 is selected first.



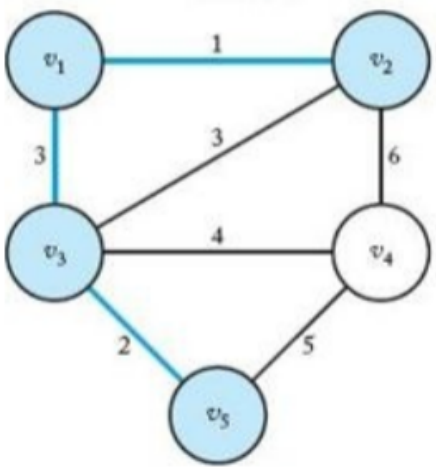
2. Vertex v_2 is selected because it is nearest to $\{v_1\}$.



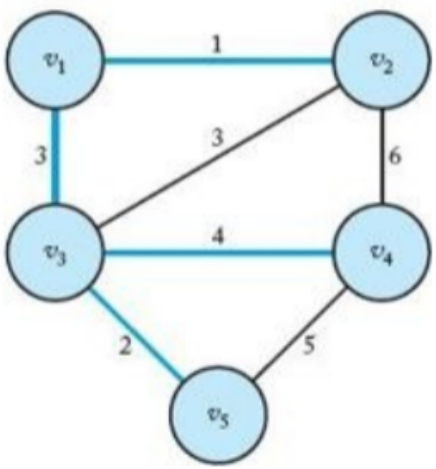
3. Vertex v_3 is selected because it is nearest to $\{v_1, v_2\}$.



4. Vertex v_2 is selected because it is nearest to $\{v_1, v_2, v_3\}$.



5. Vertex v_4 is selected.



Time Complexity Analysis

```
void prim (int n,
           const number W[][],
           set_of_edges& F)
{
  index i, vnear;
  number min;
  edge e;
  index nearest[2..n];
  number distance[2..n];

  F = ∅;
  for (i = 2; i <= n; i++){
     nearest[i] = 1;              // For all vertices, initialize v1
     distance[i] = W[1][i];       // to be the nearest vertex in
  }                               // Y and initialize the distance
                                  // from Y to be the weight
                                  // on the edge to v1.
  repeat (n - 1 times){
                                        // Add all n - 1 vertices to Y.
     min = ∞;
     for (i = 2; i <= n; i++)           // Check each vertex for
        if (0 ≤ distance[i] < min){     // being nearest to Y.
           min = distance[i];
           vnear = i;
        }
     e = edge connecting vertices indexed
         by vnear and nearest[vnear];
     add e to F;
     distance[vnear] = -1;              // Add vertex indexed by
     for (i = 2; i <= n; i++)           // vnear to Y.
        if (W[i][vnear] < distance[i]){ // For each vertex not in
           distance[i] = W[i][vnear];   // Y, update its distance
           nearest[i] = vnear;          // from Y.
        }
  }
}
```

Every-case Time Complexity:

Basic operation: There are two loops, each with $n-1$ iterations, inside the repeat loop. Executing the instructions inside each of them can be considered to be doing the basic operation once.

Input size: n, the number of vertices.

$\Rightarrow T(n) = 2(n-1)(n-1) \in \Theta(n^2)$

Discussion Question

Fastest implementation of Prim's algorithm: $O(m + n \log n)$. Which data structure?

- For a sparse graph, this is $\Theta(n \log(n))$, and for a dense graph it is $\Theta(n^2)$

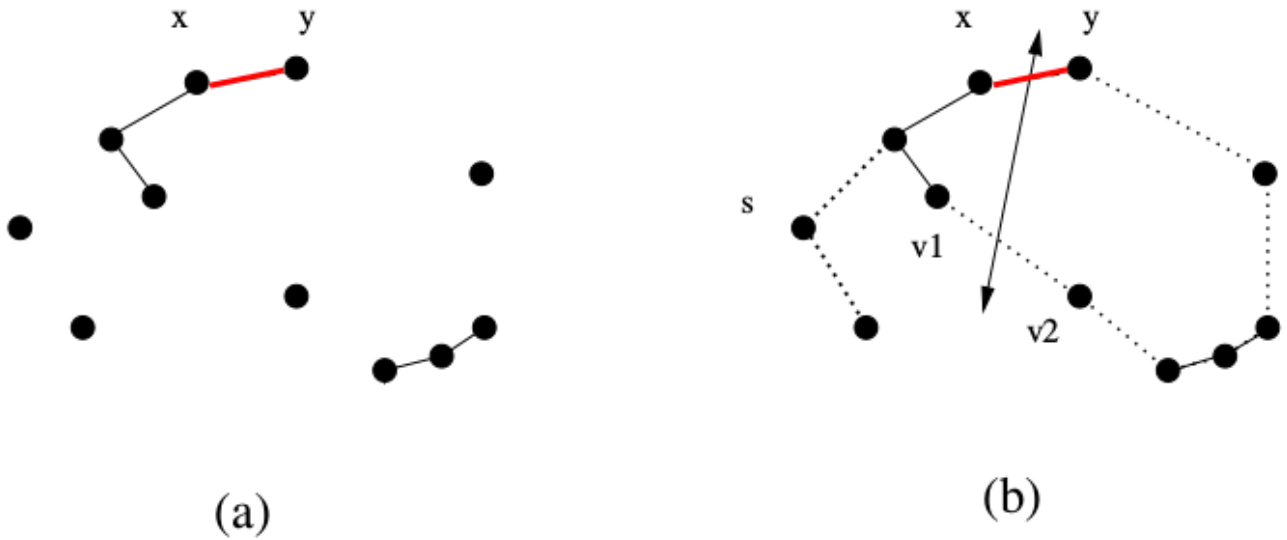
Kruskal's Algorithm

Kruskal's Algorithm starts by creating disjoint subsets of V, one for each vertex and containing only that vertex. It then inspects the edges according to nondecreasing weight (ties are broken arbitrarily). If an edge connects two vertices in disjoint subsets, the edge is added and the subsets are merged into one set. This process is repeated until all the subsets are merged into one set.

```
F = ∅;                                          // Initialize set of
                                                // edges to empty.
create disjoint subsets of V, one for each
vertex and containing only that vertex;

sort the edges in E in nondecreasing order;

while (the instance is not solved){

  select next edge;                             // selection procedure

  if (the edge connects two vertices in         // feasibility check
                    disjoint subsets){
      merge the subsets;
      add the edge to F;
  }
  if (all the subsets are merged)               // solution check
     the instance is solved;
}
```

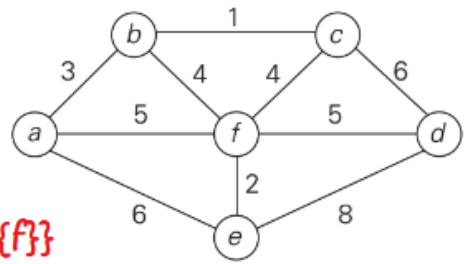
► This algorithm clearly creates a spanning tree for any connected graph. But why must this be a minimum spanning tree?

Proof by Contradiction ....



Let (x, y) be the first fatal error. Then there would be another path from x to y in the optimum tree (T_{opt}) like the one in figure b. Inserting the edge (x, y) into T_{opt} will create a cycle. Since x and y were in different components at the time of inserting (x, y), we have at least one edge (say (v_1, v_2)) on the path from x to y in T_{opt} that $w(v_1, v_2) \geq w(x, y)$, so exchanging the two edges yields a tree of weight at most equal to the weight of T_{opt}. So the Kruskal's algorithm has not made a fatal error! □

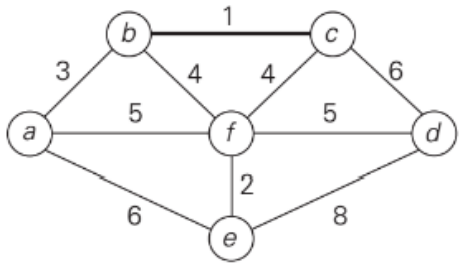
Example



{{a},{b},{c},{d},{e},{f}}

Tree edges	Sorted list of edges	Illustration
	bc ef ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	

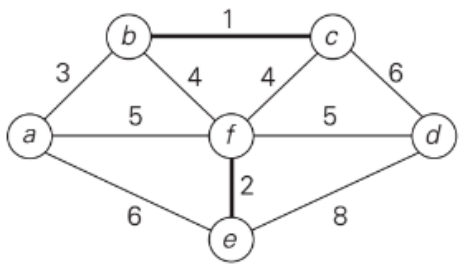
{{a},{b,c},{d},{e},{f}}



bc
1

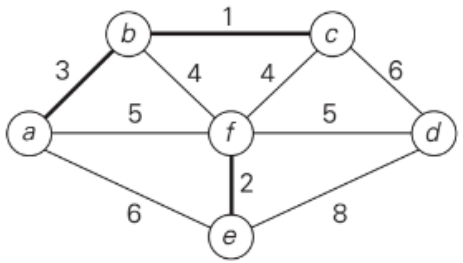
bc **ef** ab bf cf af df ae cd de
1 2 3 4 4 5 5 6 6 8

{{a},{b,c},{d},{e,f}}



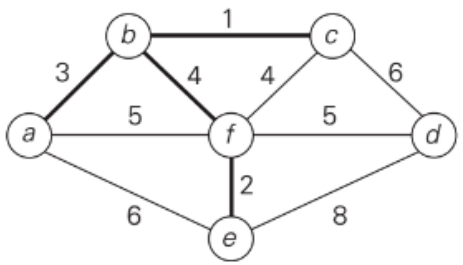
ef	bc	ef	**ab**	bf	cf	af	df	ae	cd	de
2	1	2	3	4	4	5	5	6	6	8

{{a,b,c},{d},{e,f}}



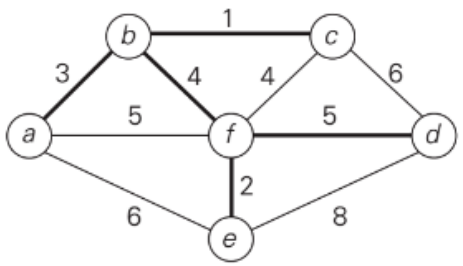
ab	bc	ef	ab	**bf**	cf	af	df	ae	cd	de
3	1	2	3	4	4	5	5	6	6	8

{{a,b,c,e,f},{d}}



bf	bc	ef	ab	bf	cf	af	**df**	ae	cd	de
4	1	2	3	4	4	5	5	6	6	8

{{a,b,c,d,e,f}}



df
5

Implementation: The Union-Find Data Structure

A set partition is a partitioning of the elements of some universal set (say the integers 1 to n) into a collection of disjointed subsets.

► Kruskal's algorithm is one of a number of applications that require a dynamic partition of some n element set S into a collection of disjoint subsets $S_1, S_2, \cdots, S_k$.

- After being initialized as a collection of n one-element subsets, each containing a different element of S, the collection is subjected to a sequence of intermixed **union** and **find** operations.

Thus we are dealing here with an abstract data type of a collection of disjoint subsets of a finite set with the following operations:

- MAKE-SET(x) creates a one-element set $\{x\}$. It is assumed that this operation can be applied to each of the elements of set S only once.
- FIND-SET(x) returns a subset containing x
- UNION(x, y) constructs the union of the disjoint subsets S_x and S_y containing x and y, respectively, and adds it to the collection to replace S_x and S_y, which are deleted from it.

Example

let $S = \{1, 2, 3, 4, 5, 6\}$. Then MAKE-SET(i) creates the set $\{i\}$ and applying this operation six times initializes the structure to the collection of six singleton sets:

$$\{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}.$$

Performing UNION$(1, 4)$ and UNION$(5, 2)$ yields

$$\{1, 4\}, \{5, 2\}, \{3\}, \{6\},$$

and, if followed by UNION$(4, 5)$ and then by UNION$(3, 6)$, we end up with the disjoint subsets

$$\{1, 4, 5, 2\}, \{3, 6\}.$$

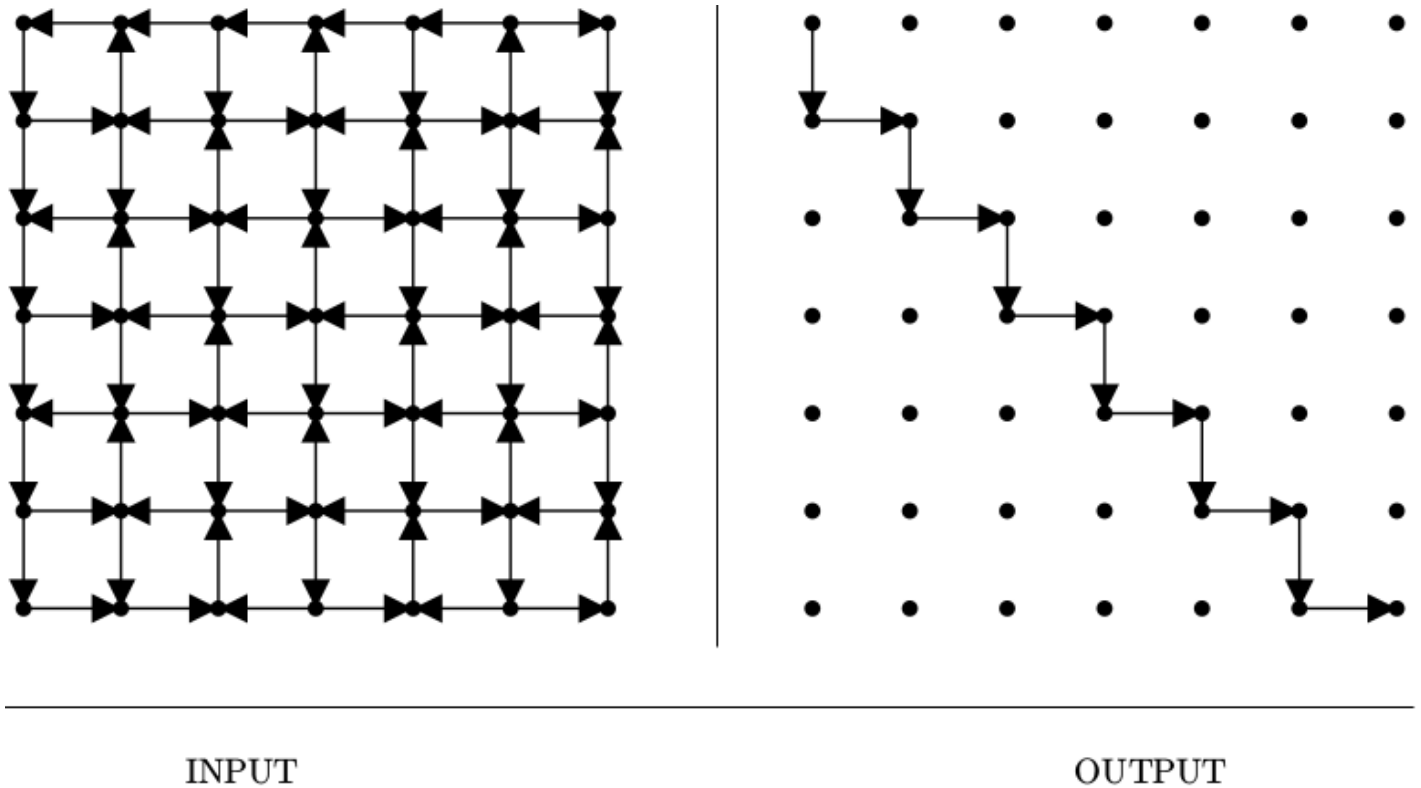
```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ G.V
3       MAKE-SET(v)
4  sort the edges of G.E into nondecreasing order by weight w
5  for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
6       if FIND-SET(u) ≠ FIND-SET(v)
7            A = A ∪ {(u, v)}
8            UNION(u, v)
9  return A
```

Worst-Case Time-Complexity: depends on how we implement the disjoint-set data structure. In one of the alternatives

- Each MAKE-SET and FIND-SET operation takes $O(1)$ time
- Any legitimate sequence (must be $\leq n-1$) of UNION operation is $O(n \log n)$

Therefore, initialization (lines 2, 3) takes $O(n)$, sorting takes $m \log m$, lines 5,...,8 take $O(m + n \log n) \Rightarrow W(n, m) \in O(m \log m)$

Shortest Path



Input description: An edge-weighted graph G
Problem description: finding the shortest paths from one vertex to all other vertices

Shortest path problem has the subproblem optimality structure:

Theorem (Subpaths of shortest paths are shortest paths)

Given a weighted, directed graph $G = (V, E)$ with weight function $w : E \to R$, let $p =< v_0, v_1, \cdots, v_k >$ be a shortest path from vertex v_0 to vertex v_k and, for any i and j such that $0 \leq i \leq j \leq k$, let $p_{ij} =< v_i, v_{i+1}, \cdots, v_j >$ be the subpath of p from vertex v_i to vertex v_j. Then, p_{ij} is a shortest path from v_i to v_j.

Recall that optimal substructure is one of the key indicators that dynamic programming and the greedy method might apply.

- the Floyd-Warshall algorithm, which finds shortest paths between all pairs of vertices, is a dynamic-programming algorithm.
- Dijkstra's algorithm, which we shall see in this Chapter, is a greedy algorithm (applicable to graphs with non-negative weights).

- Floyd-Warshal:
  - the shortest paths from each vertex to all other vertices
  - $\Theta(n^3)$ Complexity
  - overkill for one particular vertex to all the others

Here, we will use the greedy approach to develop a $\Theta(n^2)$ algorithm for this problem (called the **Single-Source Shortest Paths problem**). This algorithm is due to Dijkstra (1959).

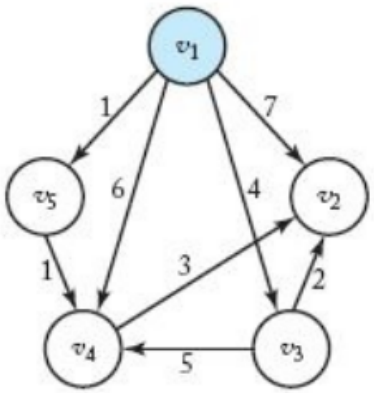
Dijkstra's Algorithm

This algorithm is similar to Prim's algorithm for the Minimum Spanning Tree problem:

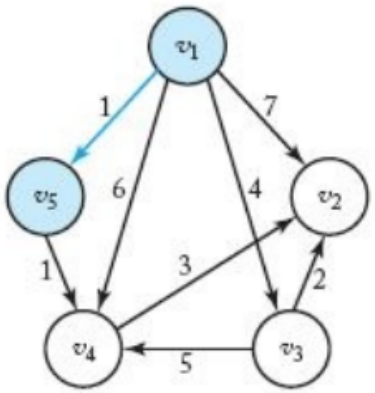
```
Y = {v1};
F = ∅;

while (the instance is not solved){

  select a vertex v from V − Y, that has a    // selection
  shortest path from v1, using only vertices  // procedure and
  in Y as intermediates;                      // feasibility check

  add the new vertex v to Y;
  add the edge (on the shortest path) that touches v to F;

  if (Y == V)
    the instance is solved;                   // solution check
}
```

Example

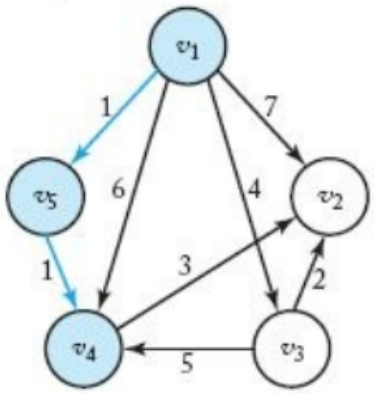
Compute shortest paths from v_1.



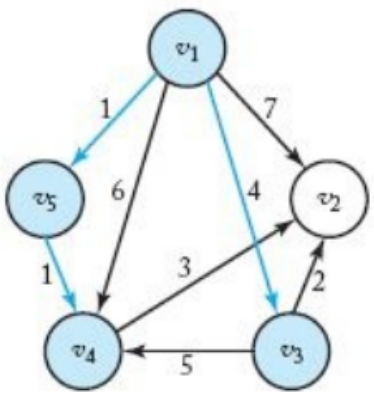
1. Vertex v_5 is selected because it is nearest to v_1.



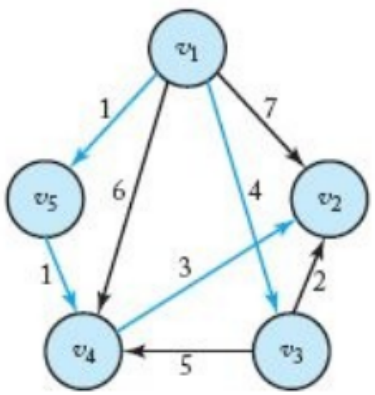
2. Vertex v_4 is selected because it has the shortest path from v_1 using only vertices in $\{v_5\}$ as intermediates.



3. Vertex v_3 is selected because it has the shortest path from v_1 using only vertices in $\{v_4, v_5\}$ as intermediates.



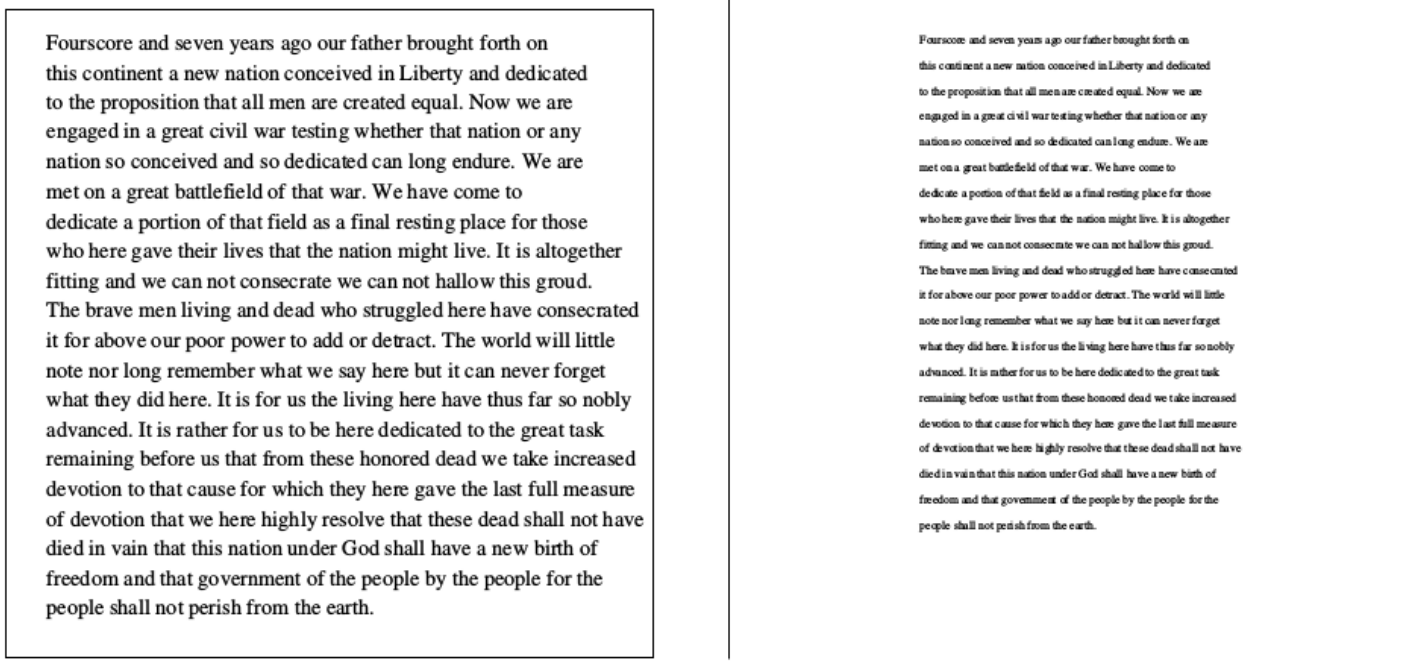
4. The shortest path from v_1 to v_2 is $[v_1, v_5, v_4, v_2]$.



```
void dijkstra (int n, const number W[][], set_of_edges& F)
{
  index i, vnear;
  edge e;
  index touch[2..n];
  number length[2..n];

  F = ∅;
  for (i = 2; i <= n; i++){                // For all vertices, initialize v1
     touch[i] = 1;                         // to be the last vertex on the
     length[i] = W[1][i];                  // current shortest path from
  }                                        // v1, and initialize length of
                                           // that path to be the weight
                                           // on the edge from v1.
  repeat (n − 1 times){                    // Add all n − 1 vertices to Y.
     min = ∞;
     for (i = 2; i <= n; i++)              // Check each vertex for
        if (0 ≤ length[i] < min){          // having shortest path.
           min = length[i];
           vnear = i;
        }
     e = edge from vertex indexed by touch[vnear]
         to vertex indexed by vnear;
     add e to F;
     for (i = 2; i <=n; i++)
        if (length[vnear] + W[vnear][i] < length[i]){
           length[i] = length[vnear] + W[vnear][i];
           touch[i] = vnear;               // For each vertex not in Y,
        }                                  // update its shortest path.
     length[vnear] = −1;                   // Add vertex indexed by vnear
  }                                        // to Y.
}
```

A Dynamic Programming Approach to the Shortest Path Problem:
Bellman-Ford Algorithm.

Text Compression: Huffman Code



Input description: A text string S.

Problem description: Create a shorter text string S' such that S can be correctly reconstructed from S'

Motivation:
Secondary storage devices fill up quickly on every computer system, even though their capacity continues to double every year. Decreasing storage prices only seem to have increased interest in data compression, probably because there is more data to compress than ever before. Data compression is the algorithmic problem of finding space-efficient encodings for a given data file. The rise of computer networks provided a new mission for data compression, that of increasing the effective network bandwidth by reducing the number of bits before transmission.

Variable-Length vs. Fixed-Length Codes

We have many options for how to represent information. Here, we consider the problem of designing a binary character code (or code for short) in which each character is represented by a unique binary string, which we call a **codeword**.

- Variable length codes replace each alphabet symbol by a variable-length code string. Using eight bits-per-symbol to encode English text (e.g., using ASCII code) is wasteful, since certain characters (such as "e") occur far more frequently than others (such as "q"). A good variable-length code assigns "e" a short code word, and "q" a longer one to compress text.

Example:

Suppose we have a 100,000-character data file that we wish to store compactly. We observe that the characters in the file occur with the frequencies given by following table. That is, only 6 different characters appear, and the character "a" occurs 45,000 times.

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

- If we assign each character a 3-bit codeword, we can encode the file in 300,000 bits.
- Using the variable-length code shown, we can encode the file in only 224,000 bits (why?) $\Rightarrow$ **a savings of approximately 25%**.

Prefix codes: No codeword is prefix of any other codeword.

- Encoding is always simple for any binary character code; we just concatenate the codewords representing each character of the file.
  - For example, with the variable-length prefix code of the last example, we code the 3-character file abc as $0 \cdot 101 \cdot 100 = 0101100$, where "$\cdot$" denotes concatenation.
- Prefix codes are desirable because they simplify decoding. Since no codeword is a prefix of any other, the codeword that begins an encoded file is unambiguous.

How to decode a Prefix Code:

Scan the bit sequence from left to right.

- As soon as you've seen enough bits to match the encoding of some letter, output this as the first letter of the text. This must be the correct first letter, since no shorter or longer prefix of the bit sequence could encode any other letter.
- Now delete the corresponding set of bits from the front of the message and iterate.
- example, the string 001011101 parses uniquely as $0 \cdot 0 \cdot 101 \cdot 1101$, which decodes to $aabe$.

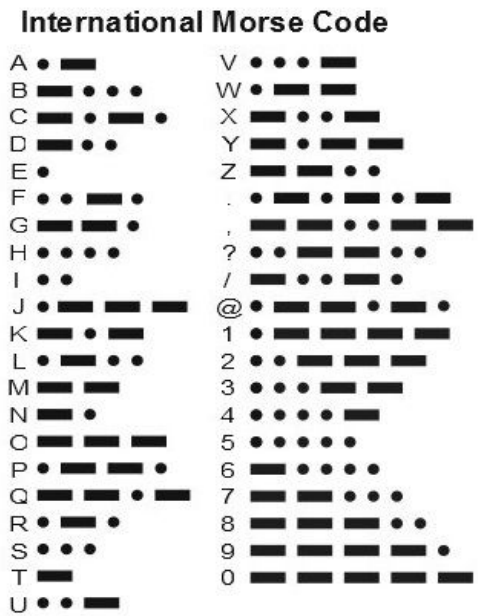
Example:

1- What is wrong with this code:

$$a \to 01, \quad b \to 0 \quad c \to 00$$

Answer: No uniqueness without separating character

Morse Codes:



For our purposes, we can think of dots and dashes as zeros and ones (Thus, Morse code maps e to 0 (a single dot), t to 1 (a single dash), a to 01 (dot-dash)). Then, the string 0101 could correspond to any of the sequences of letters eta, aa, $etet$, or aet.

To deal with this ambiguity, Morse code transmissions involve short pauses between letters (so the encoding of aa would actually be dot-dash-pause-dot-dash-pause). we've actually encoded it using a three-letter alphabet of 0, 1, and "pause".

Example:

2- What is wrong with this code:

$$a \to 0, \quad b \to 01 \quad c \to 11$$

Answer: It has uniqueness, but it is **inefficient in decoding**. Assume the coded string is

$$01 \quad \underbrace{1111111111111111...}$$

The number of 1's: even or odd?

The advantage of a prefix code is that we do not need to look ahead when parsing a coded string

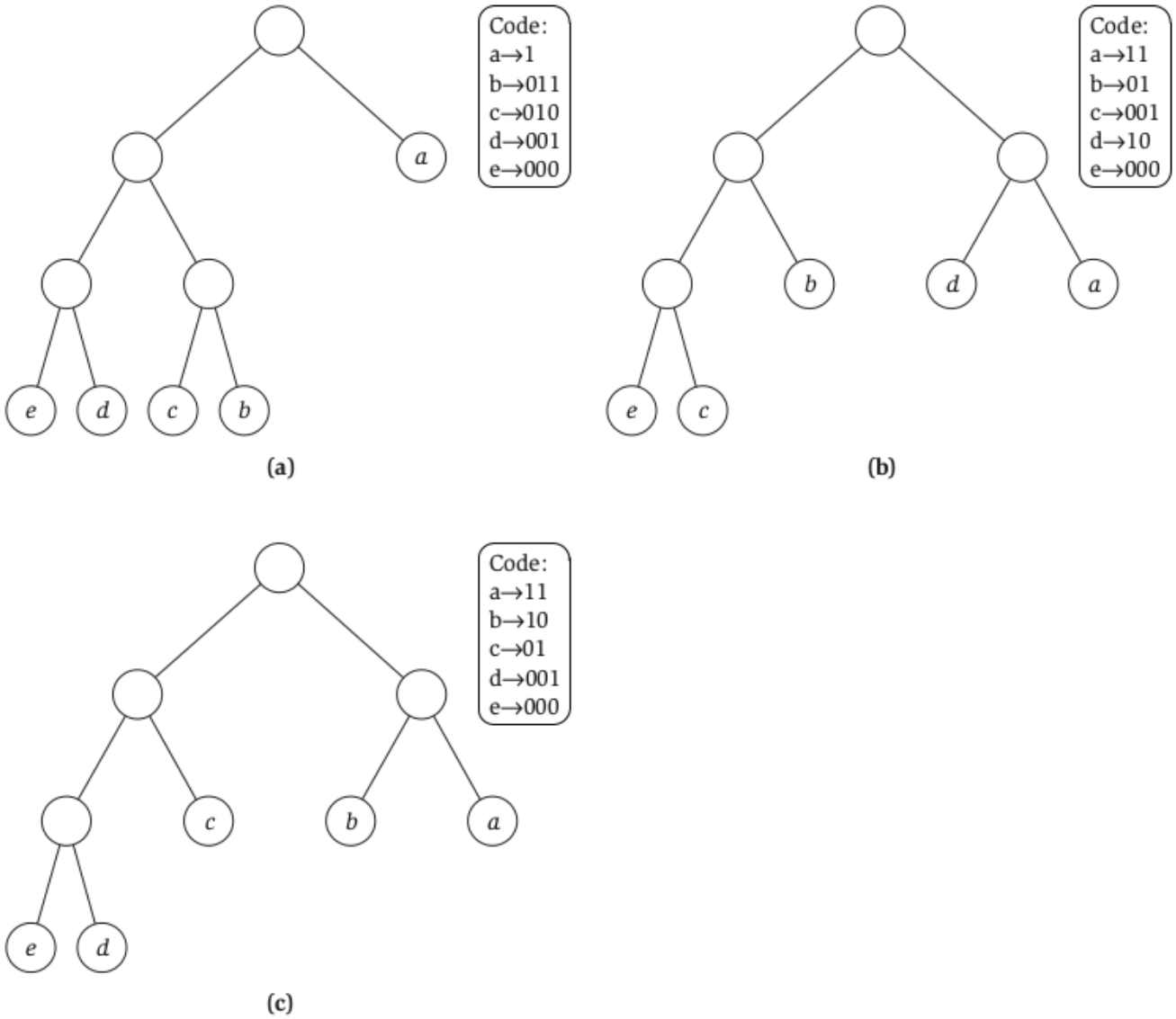
Representing Prefix Codes Using Binary Trees

In a binary tree, suppose that the number of leaves is equal to the size of the alphabet C, and we label each leaf with a distinct letter in C. Such a labeled binary tree T naturally describes a prefix code, as follows.

- For each letter $x \in C$, we follow the path from the root to the leaf labeled x;
- each time the path goes from a node to its left child, we write down a 0, and each time the path goes from a node to its right child, we write down a 1. We take the resulting string of bits as the encoding of x.

The encoding of C constructed from T is a prefix code. (Why?)

Example: different prefix codes for the alphabet $C = \{a, b, c, d, e\}$



Constructing an optimal binary tree

Let's denote by f_x, the fraction of letters in the text (which we want to compress) that are equal to x. In other words, assuming there are n letters total, nf_x of these letters are equal to x. We notice that the $\sum_{x \in C} f_x = 1$. Then, we have:

$$\text{encoding length} = \sum_{x \in C} n f_x \text{depth}(x) = n \sum_{x \in C} f_x \text{depth}(x)$$

- Observation 1: The binary tree corresponding to the optimal prefix code is full (every nonleaf node has exactly two children). (why?)
- Observation 2: In the binary tree corresponding to the optimal prefix code, if $f_x \geq f_y \rightarrow \text{depth}(x) \leq \text{depth}(y)$. (why?)

Huffman codes – Huffman invented a greedy algorithm that constructs an optimal prefix code called a Huffman code. The essence of the Huffman's approach to make the optimal coding with corresponding tree T is

There is an optimal prefix code, in which the two lowest-frequency letters are assigned to leaves that are siblings in T.

```
HUFFMAN(C)
1  n = |C|
2  Q = C
3  for i = 1 to n − 1
4       allocate a new node z
5       z.left = x = EXTRACT-MIN(Q)
6       z.right = y = EXTRACT-MIN(Q)
7       z.freq = x.freq + y.freq
8       INSERT(Q, z)
9  return EXTRACT-MIN(Q)      // return the root of the tree
```

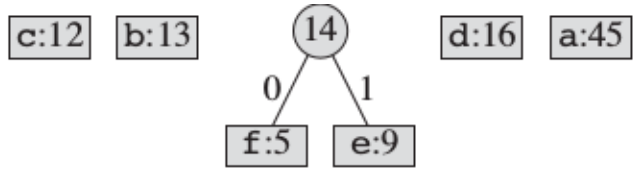
we assume that C is a set of n characters and that each character $c \in C$ is an object with an attribute $c.freq$ giving its frequency.

- Line 2 initializes the min-priority queue Q with the characters in C.
- The **for** loop in lines 3–8 repeatedly extracts the two nodes x and y of lowest frequency from the queue, replacing them in the queue with a new node, z, representing their merger.
- The frequency of z is computed as the sum of the frequencies of x and y in line 7. The node z has x as its left child and y as its right child. (This order is arbitrary; switching the left and right child of any node yields a different code of the same cost.)
- After $n - 1$ mergers, line 9 returns the one node left in the queue, which is the root of the code tree.

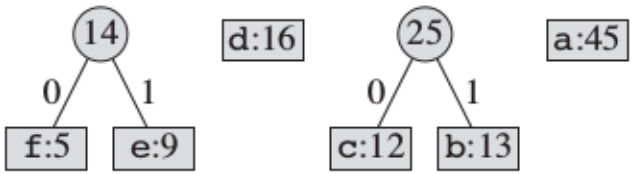
Example:

(a)

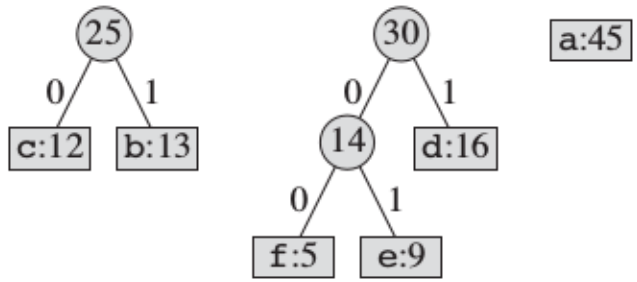
(b)



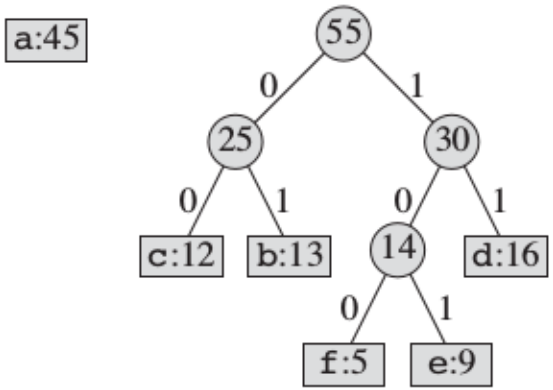
(c)



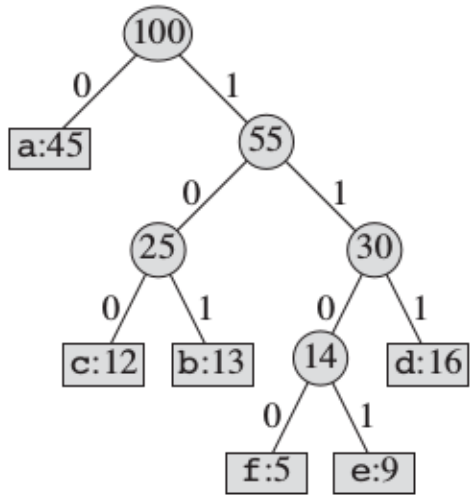
(d)



(e)



(f)



Time Complexity:
we assume that Q is implemented as a binary min-heap.

- For a set C of n characters, we can initialize Q in line 2 in $O(n)$ time.
- The **for** loop in lines 3–8 executes exactly $n-1$ times, and since each heap operation requires time $O(\log n)$, the loop contributes $O(n \log n)$ to the running time.

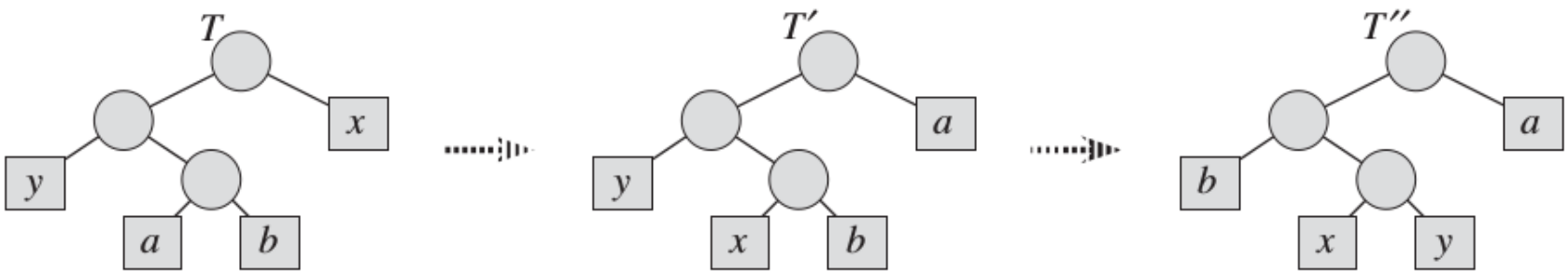
$$\Rightarrow T(n) \in O(n \log n)$$

Correctness Analysis:

Lemma (Huffman Greedy Choice Property)

Let C be an alphabet in which each character $c \in C$ has frequency $c.freq$. Let x and y be two characters in C having the lowest frequencies. Then there exists an optimal prefix code for C in which the codewords for x and y appear as sibling leaves of maximum depth in the corresponding binary tree.

Proof:



Lemma (Optimal Subproblem Property)

Let C be a given alphabet with frequency $c.freq$ defined for each character $c \in C$. Let x and y be two characters in C with minimum frequency. Let C' be the alphabet C with the characters x and y removed and a new character z added, so that $C' = C - \{x, y\} \cup \{z\}$, and $z.freq = x.freq + y.freq$. Let T' be any tree representing an optimal prefix code for the alphabet C'. Then the tree T, obtained from T' by replacing the leaf node for z with an internal node having x and y as children, represents an optimal prefix code for the alphabet C.

Theorem

Procedure HUFFMAN produces an optimal prefix code.

Proof: Immediate from Lemmas □

Stable Matching

favorite ↓ least favorite ↓

	1st	2nd	3rd
Atlanta	Xavier	Yolanda	Zeus
Boston	Yolanda	Xavier	Zeus
Chicago	Xavier	Yolanda	Zeus

hospitals' preference lists

favorite ↓ least favorite ↓

	1st	2nd	3rd
Xavier	Boston	Atlanta	Chicago
Yolanda	Atlanta	Boston	Chicago
Zeus	Atlanta	Boston	Chicago

students' preference lists

Input

	1st	2nd	3rd
Atlanta	Xavier	Yolanda	Zeus
Boston	Yolanda	Xavier	Zeus
Chicago	Xavier	Yolanda	Zeus

	1st	2nd	3rd
Xavier	Boston	Atlanta	Chicago
Yolanda	Atlanta	Boston	Chicago
Zeus	Atlanta	Boston	Chicago

a stable matching M = { A–X, B–Y, C–Z }

Output

Input description: A set of n hospitals H and a set of n students S.

- Each hospital $h \in H$ ranks students.
- Each student $s \in S$ ranks hospitals.

Problem description: Design a self-reinforcing, also called stable matching, admissions process.

self-reinforcing: individual self-interest will prevent any doctor/hospital deal from being made behind the scenes.

- Unstable pair. Hospital h and student s form an unstable pair if both:
  - h prefers s to one of its admitted students.
  - s prefers h to assigned hospital.
- Stable assignment. Assignment with no unstable pairs.
  - Natural and desirable condition.
  - Individual self-interest prevents any hospital–student side deal.

Example:

	1st	2nd	3rd
Atlanta	Xavier	Yolanda	Zeus
Boston	Yolanda	Xavier	Zeus
Chicago	Xavier	Yolanda	Zeus

	1st	2nd	3rd
Xavier	Boston	Atlanta	Chicago
Yolanda	Atlanta	Boston	Chicago
Zeus	Atlanta	Boston	Chicago

A–Y is an unstable pair for matching M = { A–Z, B–Y, C–X }

Quiz:

Which pair is unstable in the matching $\{A-X, B-Z, C-Y\}$?

A. $A-Y$.

B. $B-X$.

C. $B-Z$.

D. None of the above.

	1st	2nd	3rd
Atlanta	Xavier	Yolanda	Zeus
Boston	Yolanda	Xavier	Zeus
Chicago	Xavier	Yolanda	Zeus

	1st	2nd	3rd
Xavier	Boston	Atlanta	Chicago
Yolanda	Atlanta	Boston	Chicago
Zeus	Atlanta	Boston	Chicago

It is a surprising fact that no matter how the doctors and hospitals rate each other, there is always at least one stable matching. Further, such a matching can be found in $O(n^2)$ time

Check the stable-roommate-problem ...

Gale–Shapley acceptance algorithm:

GALE–SHAPLEY (*preference lists for hospitals and students*)

INITIALIZE M to empty matching.

WHILE (some hospital h is unmatched and hasn't proposed to every student)

  $s \leftarrow$ first student on h's list to whom h has not yet proposed.

  IF (s is unmatched)

    Add h–s to matching M.

  ELSE IF (s prefers h to current partner h')

    Replace h'–s with h–s in matching M.

  ELSE

    s rejects h.

RETURN stable matching M.

Observation 1. Hospitals propose to students in decreasing order of preference.

Observation 2. Once a student is matched, the student never becomes unmatched; only "trades up."

► **Claim.** Algorithm terminates after at most n^2 iterations of WHILE loop.

Proof: Each time through the WHILE loop, a hospital proposes to a new student. Thus, there are at most n^2 possible proposals. □

▶ **Claim.** Gale–Shapley outputs a perfect matching

Proof: the main thing we need to show is the following:

• *It is not possible that the algorithm finishes, and a hospital is not pair with any student.*

Suppose there comes a point when h is unmatched but has already proposed to every student (otherwise, the algorithm is not terminated!).

I) Based on **Observation 2**, all students are matched.

II) Since a hospital proposes only if unmatched $\Rightarrow$
a matched hospital is matched to exactly 1 student.

But there are only n hospital total, and h is not matched, so this is a contradiction. □

Theorem (Gale–Shapley 1962)

The Gale–Shapley algorithm guarantees to find a stable matching for any problem instance.

Proof.

Let's denote the output of the G-S algorithm by M^*. We prove the theorem by contradiction. Suppose that (h, s) is an unstable pair for M^*. Also assume that $(h', s), (h, s') \in M^*$. This means that

1. h prefers s to its assigned student $s' \rightarrow h$ has offered to s.
2. s prefers h to its assigned hospital h'. $\rightarrow$ if h offers to s, s is never paired with h'

$\rightarrow (s, h') \notin M^*$. But this is in contradiction to our assumption.

□

2012 Nobel Prize in Economics

Lloyd Shapley. Stable matching theory and Gale–Shapley algorithm.

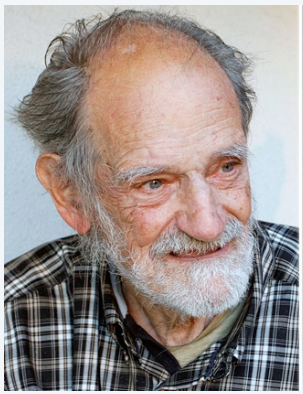
COLLEGE ADMISSIONS AND THE STABILITY OF MARRIAGE

D. GALE* AND L. S. SHAPLEY, Brown University and the RAND Corporation

1. Introduction. The problem with which we shall be concerned relates to the following typical situation: A college is considering a set of n applicants of which it can admit a quota of only q. Having evaluated their qualifications, the admissions office must decide which ones to admit. The procedure of offering admission only to the q best-qualified applicants will not generally be satisfactory, for it cannot be assumed that all who are offered admission will accept.

← original applications: college admissions and opposite-sex marriage

Alvin Roth. Applied Gale–Shapley to matching med-school students with hospitals, students with schools, and organ donors with patients.



Lloyd Shapley

Alvin Roth