Chapter 4.

The Greedy Approach

Foundations of Algorithms, 5^{th} Ed.

Richard E. Neapolitan

Contents

The Greedy Approach

- Greedy Algorithm
  - arrives at a solution by making *a sequence of choices*,
    - each of which simply *looks* *the best at the moment*.
  - That is, each choice is *locally optimal*.
  - The hope is that a *globally optimal* solution will be obtained,
    - but this is ***not always*** the case.
  - For a given (greedy) algorithm,
    - we *must determine* whether the (greedy) solution is *always optimal*.

The Greedy Approach

- The problem of *giving change* for a purchase:
  - Our goal is to give the correct change with *as few coins as possible*.
  - A ***greedy approach*** to the problem:
    - Initially, there are no coins in the change.
    - *(selection procedure)* Look for the larges coin (in value) you can find.
    - *(feasibility check)* If the total change does not exceed the amount owed,
      - add the coin to the change
    - *(solution check)* Check if the change is now equal to the amount owed.
    - If the values are not equal, *repeat the process until*
      - the value of the change equals the amount owed,
      - or there is no coins left.

The Greedy Approach

- High-level algorithm for the greedy approach:

```
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 amount owed)
        reject the coin;                 // feasibility check
    else
        add the coin to the change;
    if (the total value of the change equals the amount owed)
        the instance is solved;          // solution check
}
```

The Greedy Approach

- An example:
  - coins = [quarter, dime, dime, nickel, penny, penny] = [25, 10, 10, 5, 1, 1]
  - amount owed = 36 cents.
  - A greedy algorithm for giving change.
    - change = [25] < 36. Grab.
    - change = [25, 10] < 36. Grab.
    - change = [25, 10, ~~10~~] > 36. Reject.
    - change = [25, 10, ~~5~~] > 36. Reject.
    - change = [25, 10, 1] = 36. Grab and terminate.

The Greedy Approach

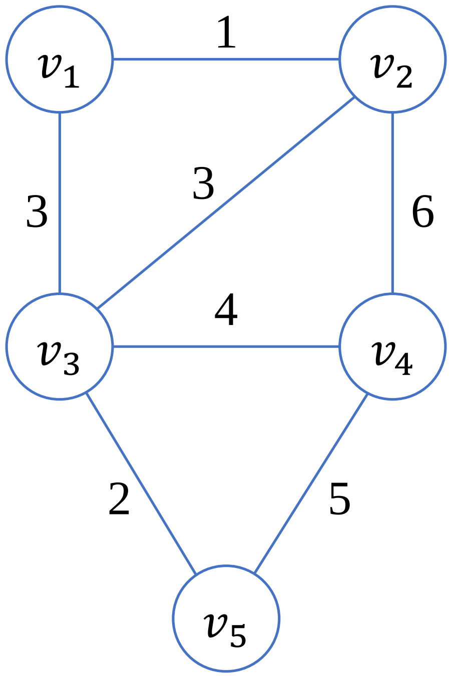
- Does it *always* result in an *optimal* solution?
  - Notice here that if we include a 12-cent coin with the U.S. coins,
    - the greedy algorithm does not always give an optimal solution.
  - coins = [12, 10, 5, 1, 1, 1, 1]
  - amount owed = 16 cents.
  - A greedy algorithm for giving change.
    - change = [12] < 16. Grab.
    - change = [12, ~~10~~] > 16. Reject.
    - change = [12, ~~5~~] > 16. Reject.
    - change = [12, 1, 1, 1, 1] = 16. Grab and terminate.
    - optimal change = [10, 5, 1]

- The ***Greedy Algorithm***
  - starts with an *empty set* and adds items to the set *in sequence*
    - until the set represents a solution to an instance of a problem.
  - Each iteration consists of three steps:
    1. *Selection Procedure*:
       - chooses the next item to add to the set.
    2. *Feasibility Check*:
       - determines if the new set if feasible.
    3. *Solution Check*:
       - determines whether the new set constitutes a solution.

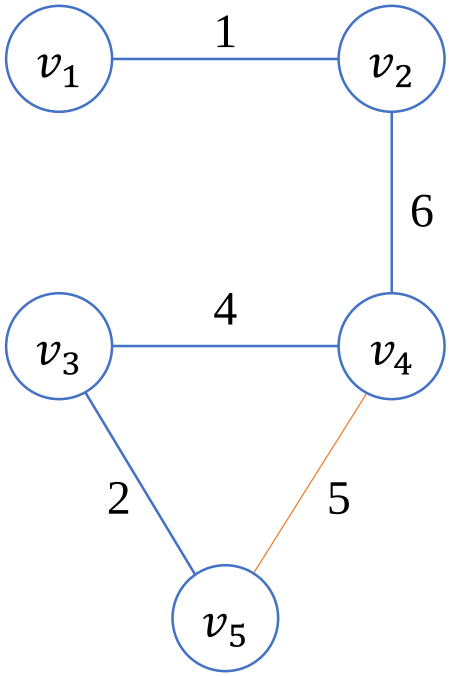
4.1 Minimum Spanning Trees

- *Minimum Spanning Tree* Problem:
  - The problem of removing edges
    - from a *connected*, *weighted*, *undirected* graph G
    - to form a *subgraph* such that *all the vertices* remains *connected*, and
    - the *sum of the weights* on the remaining edges is *as small as possible*.
  - A *spanning tree* for G is a connected subgraph
    - that contains all the vertices in G and is a tree.
  - A ***minimum spanning tree*** (MST) is
    - a spanning tree of *minimum weight*.

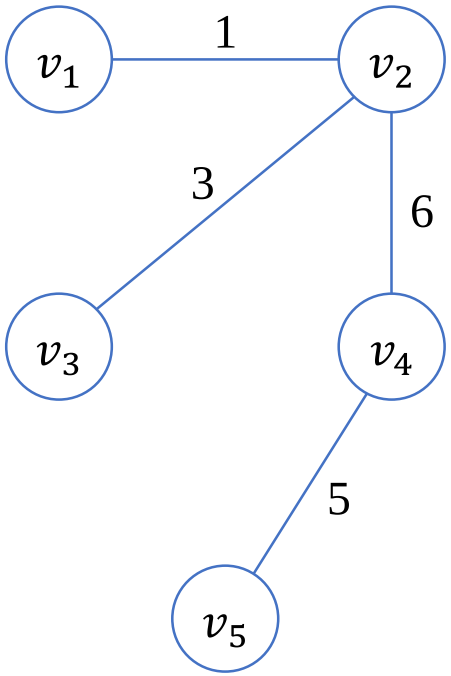
4.1 Minimum Spanning Trees



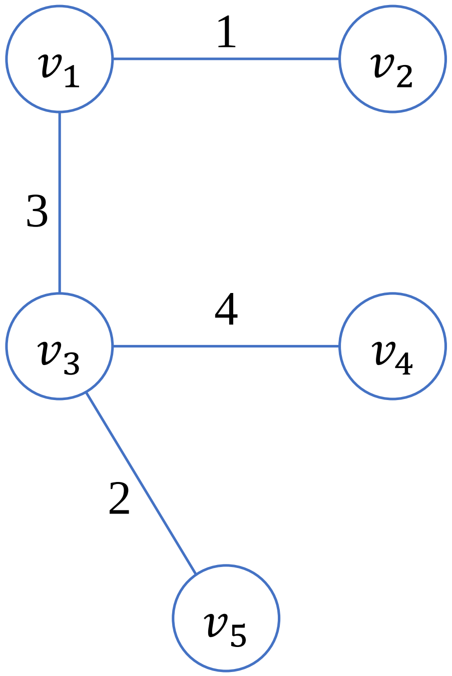
- A connected, weighted, undirected graph G.



- If (v_4, v_5) were removed, the graph would remain connected.



- A spanning tree for G.



- A minimum spanning tree for G.

- Formal Definition of the MST Problem:
  - Given a connected, weighted, undirected graph $G = (V, E)$.
  - A spanning tree T for G has the same vertices V as G,
    - but the set of edges of T is a subset F of E.
  - Denote a spanning tree by $T = (V, F)$.
  - Our problem is to find a subset F of E
    - such that $T = (V, F)$ is a *minimum spanning tree* for G.

The Greedy Approach

- High-level greedy algorithm for the MST problem

```
F = ∅;
while (the instance is not solved) {
   select an edge according to some locally optimal consideration;
   if (adding the edge to F does not create a cycle)
      add it;
   else
      add the coin to the change;
   if (T = (V, F) is a spanning tree)
      the instance is solved;
}
```

Prim's Algorithm

- starts with an *empty set* of edges F
  - and a *subset of vertices* Y initialized to contain an *arbitrary* vertex (v_1).
- A vertex *neatest* to Y is a vertex in $V - Y$
  - that is connected to a vertex in Y by an edge of *minimum weight*.
- The *vertex* that is *nearest* to Y is added to Y
  - and the *edge* is added to F. (Ties are broken arbitrarily)
- This process of adding nearest vertices is
  - repeated until $Y = V$.

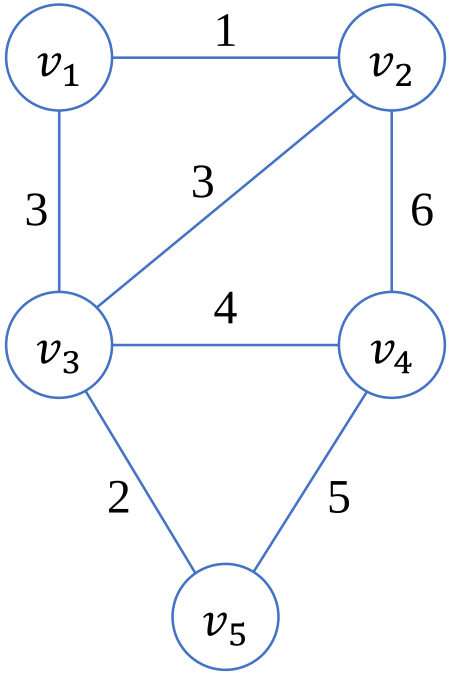
The Greedy Approach

- High-level pseudo-code for the Prim's algorithm

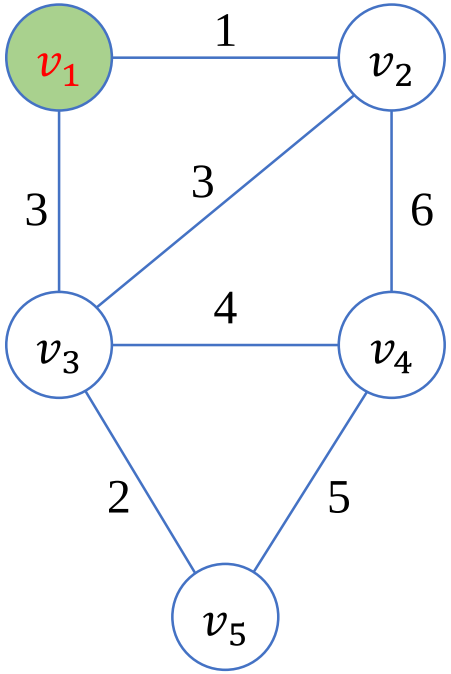
```
F = ∅;
Y = {v_1};
while (the instance is not solved) {
    select a vertex in V − Y that is nearest to Y;
    add the vertex to Y;
    add the edge to F;
    if (Y = V)
        the instance is solved;
}
```

4.1 Minimum Spanning Trees

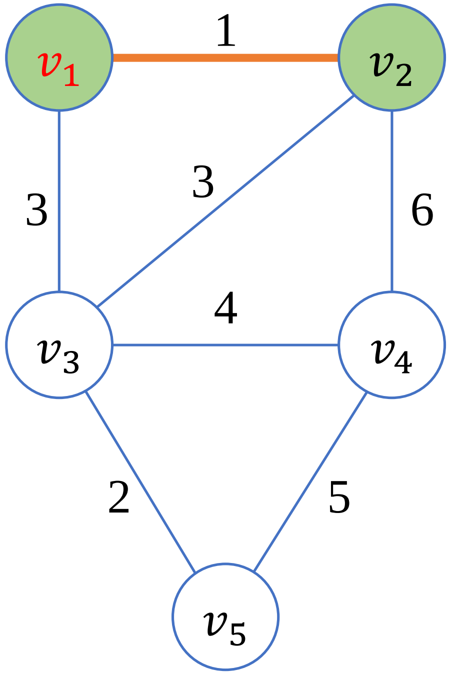
- Determine a minimum spanning tree



1. Vertex v_1 is selected first

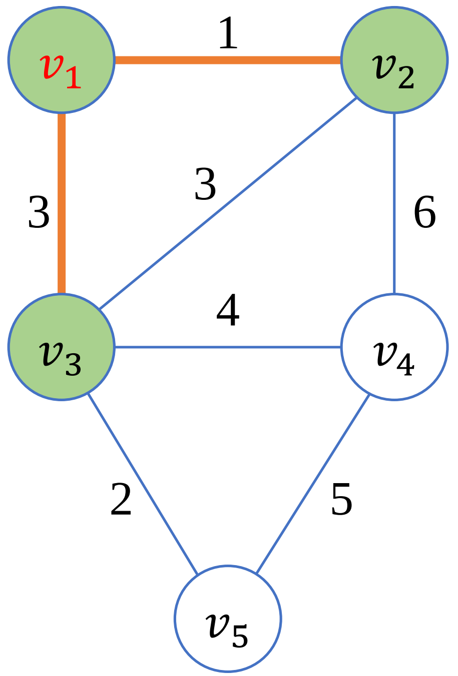


2. Vertex v_2 is selected because it is nearest to $\{v_1\}$

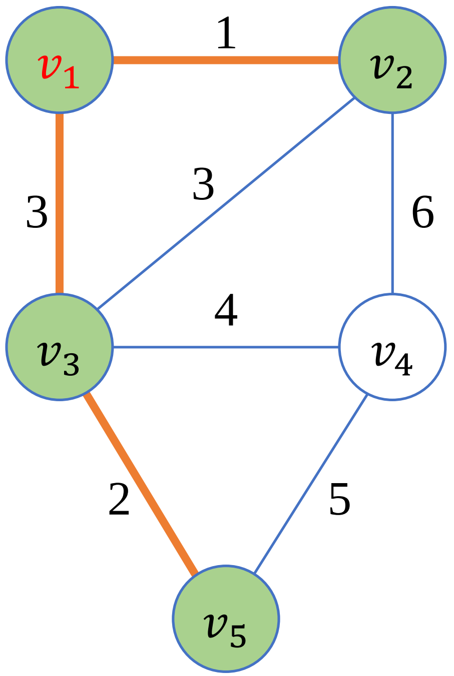


4.1 Minimum Spanning Trees

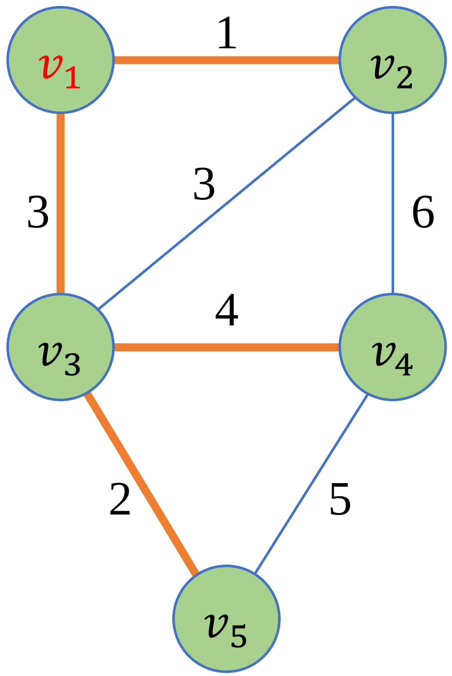
3. Vertex v_3 is selected because it is nearest to $\{v_1, v_2\}$



4. Vertex v_5 is selected because it is nearest to $\{v_1, v_2, v_3\}$



5. Vertex v_4 is selected because it is nearest to $\{v_1, v_2, v_3, v_5\}$



- Implementing the Prim's algorithm:
  - Represent a weighted graph by its adjacency matrix.
    - $W[i][j] = \begin{cases} weight\ on\ edge & \text{if there is an edge between } v_i \text{ and } v_j \\ \infty & \text{if there is no edge between } v_i \text{ and } v_j \\ 0 & \text{if } i = j \end{cases}$
  - We maintain two arrays, $nearest$ and $distance$, where, for $i = 2, \ldots, n$,
    - $nearest[i]$ = index of the vertex in Y nearest to v_i
    - $distance[i]$ = weight on edge between v_i and the vertex indexed by $nearest[i]$

4.1 Minimum Spanning Trees

ALGORITHM 4.1: Prim's Algorithm

```
void prim(int n, int W[][MAX], set_of_edges &F) {
    int i, vnear, min, nearest[n + 1], distance[n + 1];
    edge_pointer e;

    F.clear(); // F = ∅;

    for (i = 2; i <= n; i++) {
        nearest[i] = 1;
        distance[i] = W[1][i];
    }
```

ALGORITHM 4.1: Prim's Algorithm (continued)

```
    repeat (n - 1 times) {
        min = ∞;
        for (i = 2; i <= n; i++)
            if (0 ≤ distance[i] < min) {
                min = distance[i];
                vnear = i;
            }
        e = edge connecting vertices indexed by vnear and nearest[vnear];
        add e to F;
        distance[vnear] = -1;
        for (i = 2; i <= n; i++)
            if (W[i][vnear] < distance[i]) {
                distance[i] = W[i][vnear];
                 nearest[i] = vnear;
            }
    }
}
```

4.1 Minimum Spanning Trees

```
typedef struct edge *edge_pointer;
typedef struct edge {
    int u;
    int v;
    float weight;
} edgetype;

typedef vector<edge_pointer> set_of_edges;
```

```
e = (edge_pointer)malloc(sizeof(edgetype));
e->u = vnear;
e->v = nearest[vnear];
e->weight = W[e->u][e->v];

F.push_back(e);
```

4.1 Minimum Spanning Trees

W	1	2	3	4	5
1	0	1	3	∞	∞
2	1	0	3	6	∞
3	3	3	0	4	2
4	∞	6	4	0	5
5	∞	∞	2	5	0

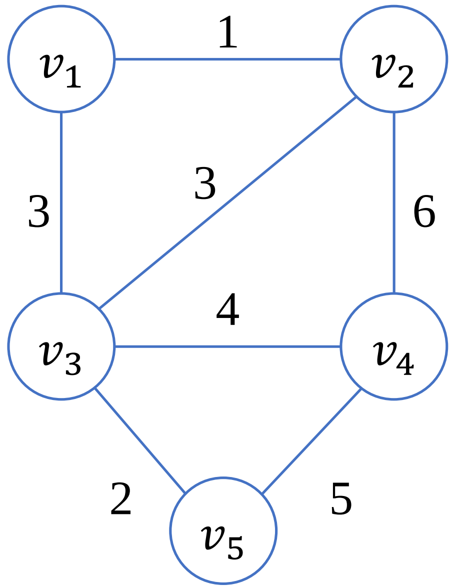
	i	2	3	4	5	*e*
init:	nearest[i]	1	1	1	1	
	distance[i]	1	3	∞	∞	
step 1:	nearest[i]	1	1	2	1	(2, 1, 1)
	distance[i]	-1	3	6	∞	
step 2:	nearest[i]	1	1	3	3	(3, 1, 3)
	distance[i]	-1	-1	4	2	
step 3:	nearest[i]	1	1	3	3	(5, 3, 2)
	distance[i]	-1	-1	4	-1	
step 4:	nearest[i]	1	1	3	3	(4, 3, 4)
	distance[i]	-1	-1	-1	-1	

4.1 Minimum Spanning Trees

- Time Complexity of Algorithm 4.1:
  - Basic Operation: the instructions inside each of two loops.
  - Input Size: n, the *number of vertices*.
  - Note that there are two (nested) loops,
    - and the *repeat* loop has n – 1 iterations.
  - Therefore,
    - $T(n) = 2(n-1)(n-1) \in \Theta(n^2)$

4.1 Minimum Spanning Trees

- Does it *always* produce an optimal solution?
  - We need to prove that
    - Prim's algorithm *always* produces a minimum spanning tree.
  - Given an undirected graph $G = (V, E)$,
    - A subset F and E is called *promising*
      - if edges can be added to it so as to form a minimum spanning tree.

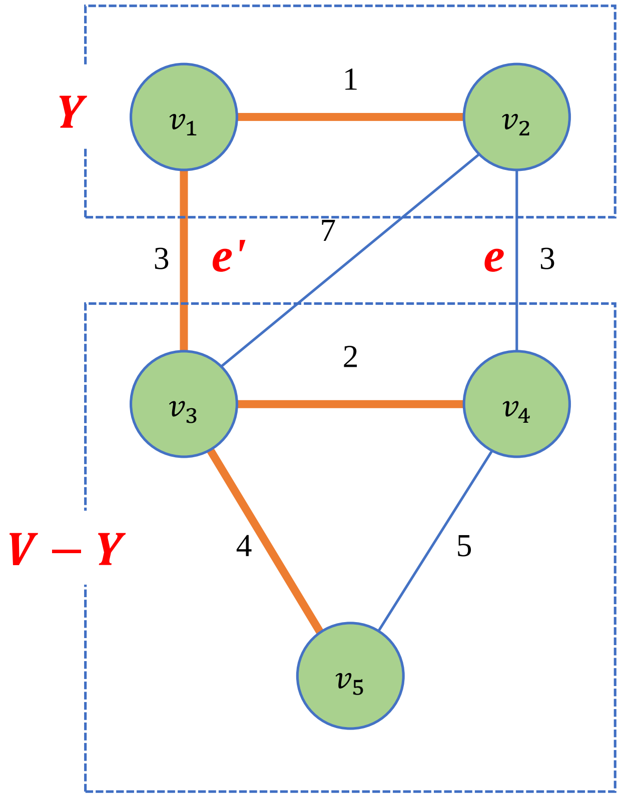


- The subset $\{(v_1, v_2), (v_1, v_3)\}$ is promising.
- The subset $\{(v_2, v_4)\}$ is not promising.

4.1 Minimum Spanning Trees

- Lemma:
  - If F is a *promising* subset of E
    - then $F \cup \{e\}$ is *promising*,
    - where e is *an edge of minimum weight* that
    - connects a vertex in Y and a vertex in $V - Y$.
  - Proof:
    - Let F' be a set edges in an MST s.t. $F \subseteq F'$.
    - If $e \in F'$, then $F \cup \{e\} \subseteq F'$.
    - If $e \notin F'$, then $F' \cup \{e\}$ must have a cycle.
    - There is an edge $e' \notin F'$ in the cycle
      - Remove e', then the cycle disappears.
      - Hence, $F' \cup \{e\} - \{e'\}$ is an MST.
    - Hence, $F \cup \{e\} \subseteq F' \cup \{e\} - \{e'\}$.

$F = \{(v_1, v_2)\}$



$F' = \{(v_1, v_2), (v_1, v_3), (v_3, v_4), (v_3, v_5)\}$

$F' \cup \{e\}$ has a cycle: $[v_1, v_2, v_4, v_3]$

4.1 Minimum Spanning Trees

- Theorem:
  - Prim's algorithm always produces a minimum spanning tree.
  - Proof:
    - Clearly, the *empty set* $\emptyset$ is *promising*.
    - Assume that, after a given iteration,
      - the selected edges F is *promising*.
    - The set $F \cup \{e\}$ is *promising*,
      - where e is the edge selected in the next iteration.
    - Because the e is an edge of minimum weight that
      - connects a vertex in Y to a vertex in $V - Y$. (by the Lemma)

4.1 Minimum Spanning Trees

- ***Kruskal's Algorithm***
  - starts by creating *disjoint subsets* of V,
    - one for *each vertex* and containing *only that vertex*.
  - If then, inspects the edge 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*.

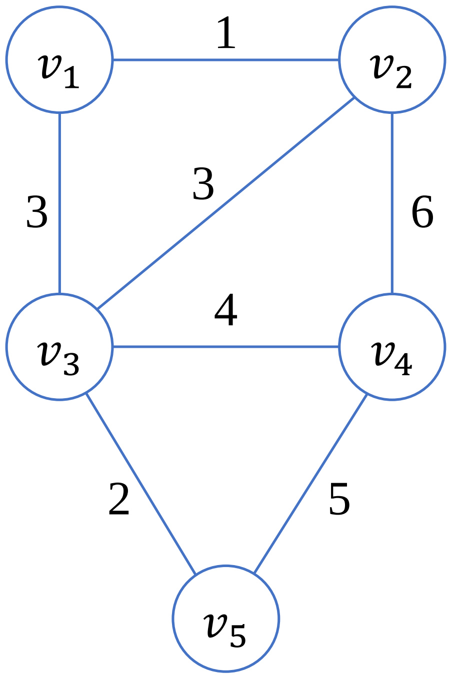
The Greedy Approach

- High-level pseudo-code for the Kruskal's algorithm

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

4.1 Minimum Spanning Trees

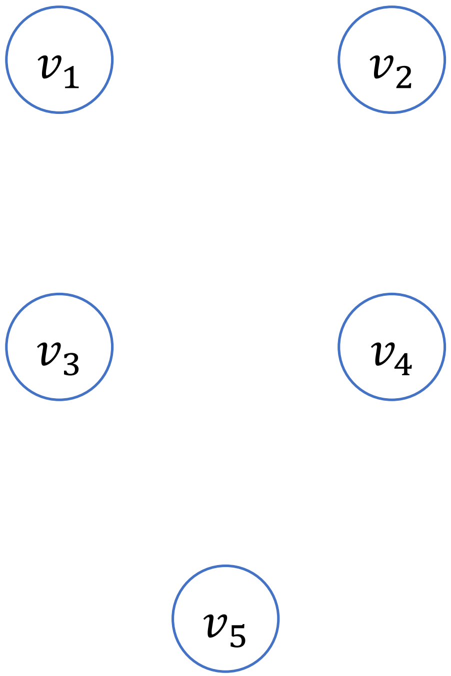
- Determine a minimum spanning tree.



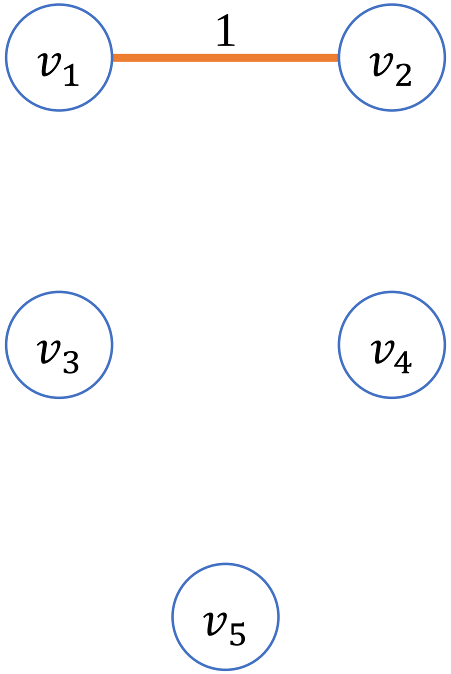
1. Edges are sorted by their weights.

edges	*weight*
(v_1, v_2)	1
(v_3, v_5)	2
(v_1, v_3)	3
(v_2, v_3)	3
(v_3, v_4)	4
(v_4, v_5)	5
(v_2, v_4)	6

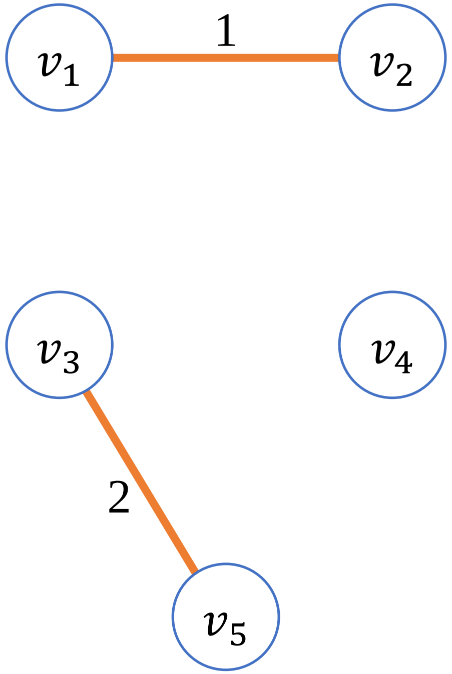
2. Disjoint sets are created.



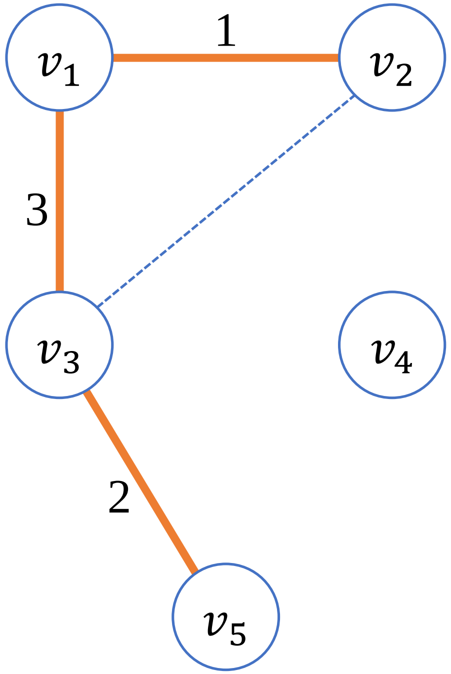
3. The first edge (v_1, v_2) is selected



4. Next edge (v_3, v_5) is selected

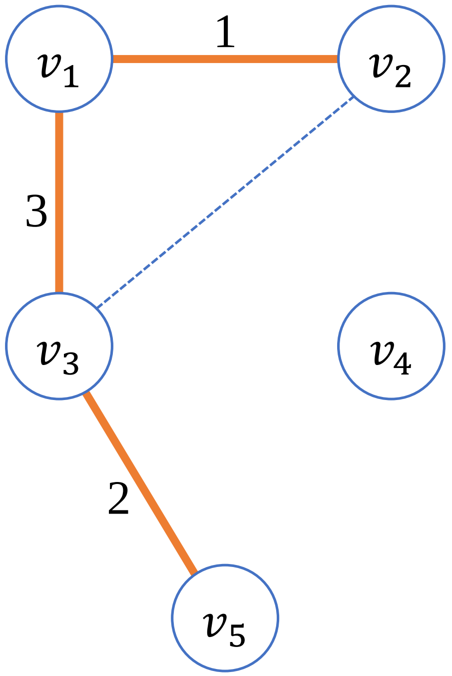


5. Next edge (v_1, v_3) is selected

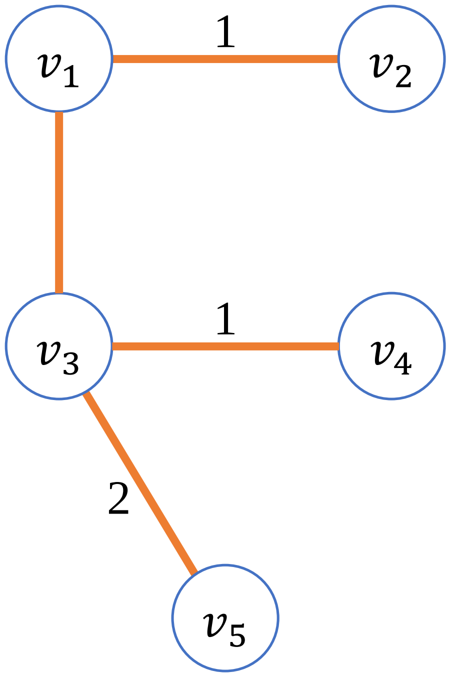


4.1 Minimum Spanning Trees

6. Next edge (v_2, v_3) is *discarded*, because it creates a cycle



7. Next edge (v_3, v_4) is selected



- (v_4, v_5) is *not considered*, because all the subsets are merged

- *Disjoint Set* Abstract Data Type
  - To write a formal version of Kruskal's algorithm,
    - we need a *disjoint set* abstract data type: Refer to *Appendix C*.
  - The ADT of the disjoint set defines two data types:
    - $index\ i$;
    - $set\_pointer\ p, q$;
  - Then the routines are defined:
    - $initial(n)$: initialzes n disjoint subsets.
    - $p = find(i)$: makes p point to the set containing index i.
    - $merge(p, q)$: merges the two sets, to which p and q point, into the set.
    - $equal(p, q)$: returns true if p and q both point to the same set.

- Let $U = \{\, A, B, C, D, E \,\}$ be a universe of elements

```
for i in U:
    initial(i);
```

$\{A\}$ $\{B\}$ $\{C\}$ $\{D\}$ $\{E\}$ **(disjoint sets)**

```
p = find(B);
q = find(C);
```

p → $\{B\}$, q → $\{C\}$

```
merge(p, q);
```

$\{A\}$ $\{B, C\}$ $\{D\}$ $\{E\}$

```
p = find(C);
q = find(E);
```

p → $\{B, C\}$, q → $\{E\}$

`equal(C, E);` *returns false;*

```
merge(p, q);
```

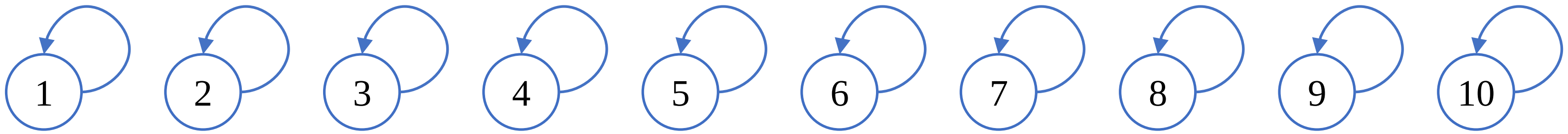
$\{A\}$ $\{B, C, E\}$ $\{D\}$

```
p = find(C);
q = find(E);
```

p → $\{B, C, E\}$, q → $\{B, C, E\}$

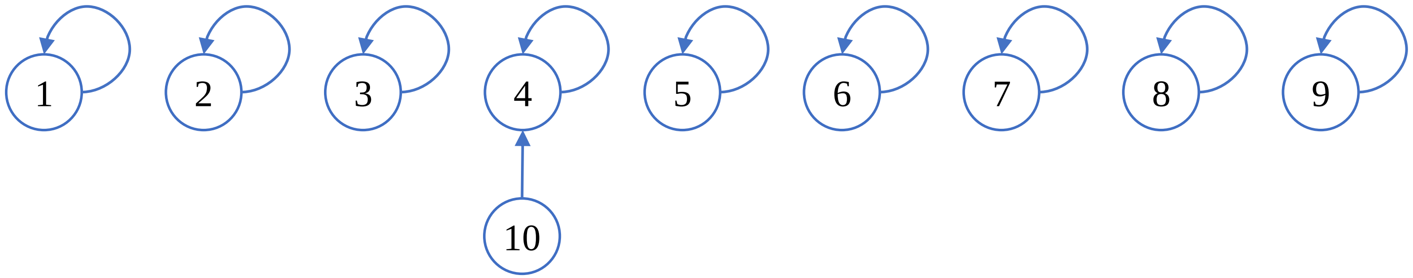
`equal(C, E);` *returns true;*

```
initial(10);
```



1	2	3	4	5	6	7	8	9	10
$U[1]$	$U[2]$	$U[3]$	$U[4]$	$U[5]$	$U[6]$	$U[7]$	$U[8]$	$U[9]$	$U[10]$

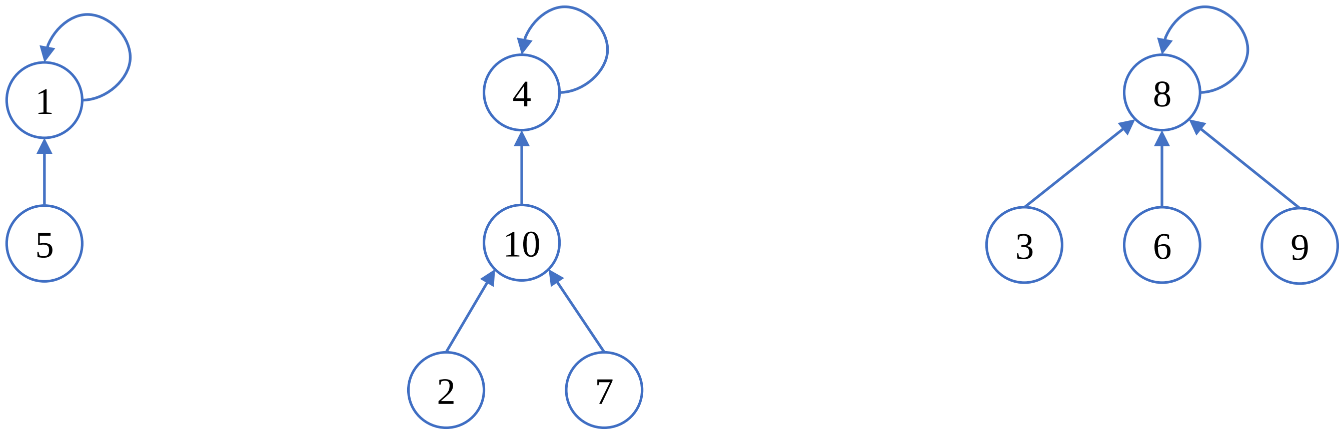
```
merge(find(4), find(10);
```



1	2	3	4	5	6	7	8	9	4
$U[1]$	$U[2]$	$U[3]$	$U[4]$	$U[5]$	$U[6]$	$U[7]$	$U[8]$	$U[9]$	$U[10]$

4.1 Minimum Spanning Trees

```
After several union and find:
```



1	10	8	**4**	1	8	10	**8**	8	4
$U[1]$	$U[2]$	$U[3]$	$U[4]$	$U[5]$	$U[6]$	$U[7]$	$U[8]$	$U[9]$	$U[10]$

- Analyze the complexity of ***union***(merge) and ***find***,
  - and improve the efficiency to $\Theta(m \lg m)$,
    - where m is the *number of passes* to call the routines($merge$ and $find$).

```
int U[MAX];

void initial(int n) {
    for (int i = 1; i <= n; i++)
        U[i] = i;
}

set_pointer find(int i) {
    int j = i;
    while (U[j] != j)
        j = U[j];
    return j;
}
```

```
bool equal(set_pointer p, set_pointer q) {
    return (p == q) ? true: false;
}

void merge(set_pointer p, set_pointer q) {
    if (p < q)
        U[q] = p;
    else
        U[p] = q;
}
```

ALGORITHM 4.2: Kruskal's Algorithm

```
void kruskal(int n, int m, set_of_edges E, set_of_edges &F) {
    int i, j;
    set_pointer p, q;
    edgetype e;

    sort the m edges in E by weight in nondecreasing order;
    F.clear(); // F = ∅;
    initial(n);
    while (number of edges in F is less than n - 1) {
        e = edge with least weight not yet considered;
        i, j = indices of vertices connected by e;
        p = find(i);
        q = find(j);
        if (!equal(p, q)) {
            merge(p, q);
            add e to F;
        }
    }
}
```

4.1 Minimum Spanning Trees

```
sort(E.begin(), E.end(), compare);


vector<edge_pointer>::iterator iter = E.begin();
while (F.size() < n - 1) {
    edge_pointer e = *iter++;
    p = find(e->u);
    q = find(e->v);
    if (!equal(p, q)) {
        merge(p, q);
        F.push_back(e);
    }
}
```

4.1 Minimum Spanning Trees

- Time Complexity of Algorithm 4.2:
  - Basic Operation: a *comparison* instruction.
  - Input Size: n, the *number of vertices*, and m, the *number of edges*.
  - Three considerations in this algorithm:
    1. The time to sort the edges: $\Theta(m \lg m)$
    2. The time to initialize n disjoint sets: $\Theta(n)$.
    3. The time in the while loop: $\Theta(m \lg m)$
       - The time complexity of *Union-Find* (Appendix C)
  - Since $m \geq n - 1$, $W(m, n) \in \Theta(m \lg m)$
  - In worst-case, the number of edges is $m = n(n-1)/2$
    - $w(m, n) \in \Theta(n^2 \lg n^2) = \Theta(n^2 \lg n)$

4.1 Minimum Spanning Trees

- Comparing Prim's Algorithm with Kruskal's Algorithm:
  - The time complexity of two algorithms:
    - Prim's: $T(n) \in \Theta(n^2)$
    - Kruskal's: $W(m,n) \in \Theta(m \lg m) = \Theta(n^2 \lg n)$
  - We can show that $n - 1 \leq m \leq \frac{n(n-1)}{2}$.
  - For a *sparse* graph,
    - whose number of edges m is near the low end of these limits,
    - Kruskal's algorithm is $\Theta(n \lg n)$, which is *faster than Prim's*.
  - For a *dense* graph,
    - whose number of edges m is near the high end of those limits,
    - Kruskal's algorithm is $\Theta(n^2 \lg n)$, which is *slower than Prim's*.

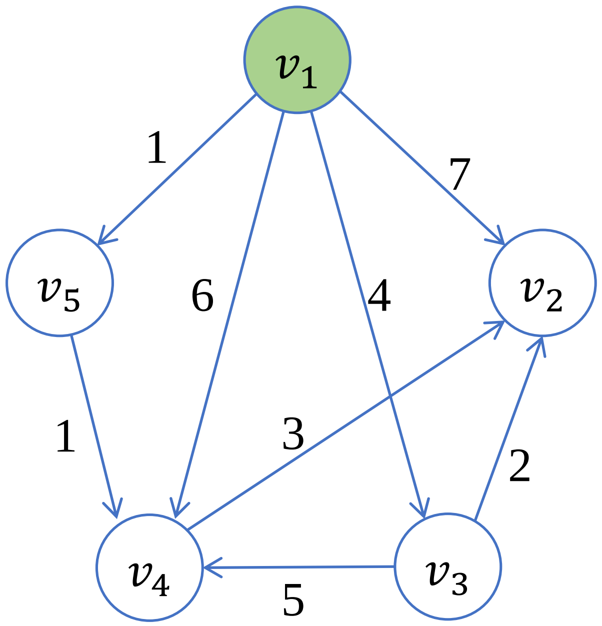
- The Problem of *Single-Source-Shortest-Paths*
  - Find a shortest path from *one particular vertex* to *all the others*.
  - *Dijkstra's Algorithm* uses the *greedy approach*
    - to develop a $\Theta(n^2)$ algorithm for this problem.

- Dijkstra's Algorithm
  - initializes a set Y to contain only the source vertex v_1,
    - and initializes a set F of edges to being empty.
  - First, choose a vertex v that is *nearest* to v_1,
    - add it to Y, and add the edge $\langle v_1, v \rangle$ (a *shortest edge*) to F.
  - Next, check the paths from v_1 to the vertices in $V - Y$
    - that allow only vertices in Y as intermediate vertices.
  - The vertex at the end of such a path is added to Y,
    - and the edge (on the path) that *touches* that vertex is added to F.
  - Continue this procedure, until Y equals V.
    - At this point, F contains the edges in *shortest paths*.

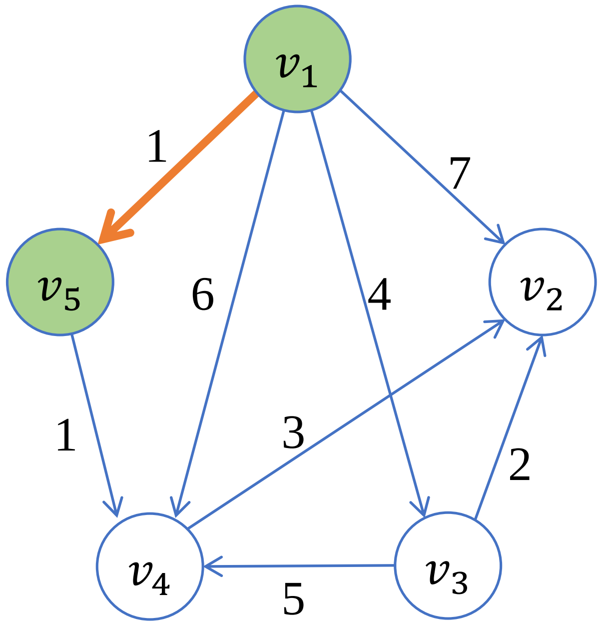
- High-level pseudo-code for the Dijkstra's algorithm:

```
Y = {v_1};
F = ∅;
while (the instance is not solved) {
    select a vertex v from V − Y that has a shortest path from v_1,
          using only vertices in Y as intermediates;
    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;
}
```

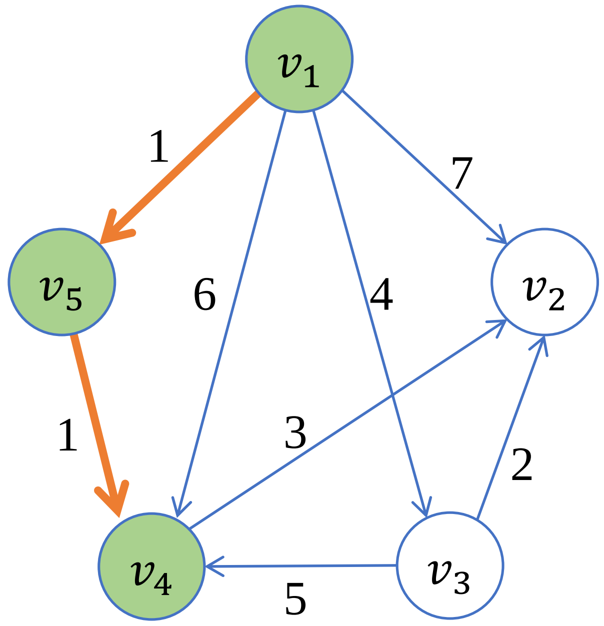
- Compute shortest paths from v_1.



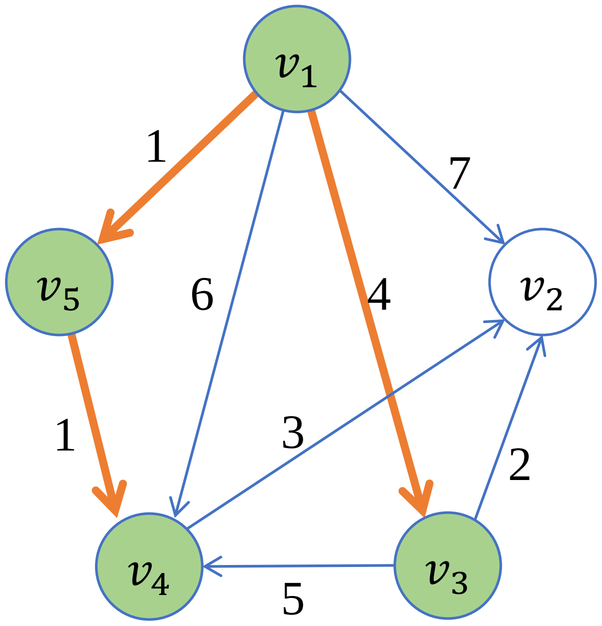
1. Vertex v_5 is selected because it is nearest to v_1.



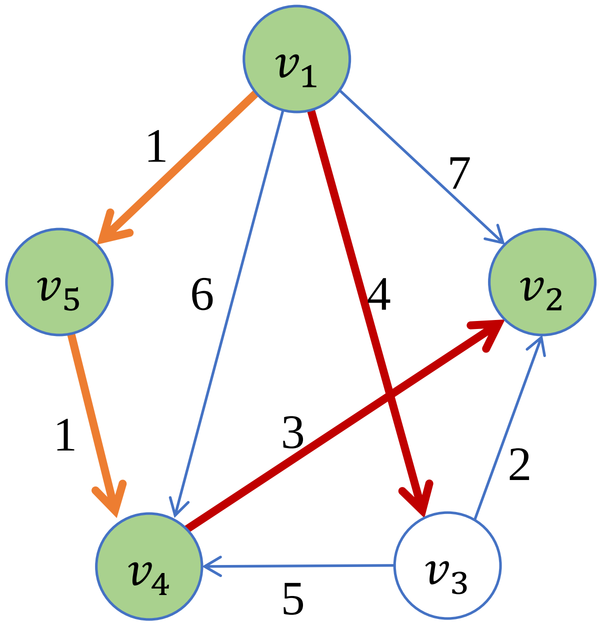
2. Vertex v_4 is selected because it has the shortest path from v_1 using only vertices in $\{v_5\}$ as intermediates.



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 $[v_1, v_5, v_4, v_2]$.



Implementation of Dijkstra's Algorithm

- It is very similar to Prim's Algorithm.
- The difference is that, instead of the arrays *nearest* and *distance*,
  - we have arrays *touch* and *length*, where for $i = 2, \cdots, n$.
- Let us define:
  - $touch[i] =$ index of vertex v in Y such that the edge $\langle v, v_i \rangle$ is the last edge on the current shortest path from v_1 to v_i using only vertices in Y as intermediates.
  - $length[i] =$ length of the current shortest path from v_1 to v_i using only vertices in Y as intermediates.

ALGORITHM 4.3: Dijkstra's Algorithm

```
void dijkstra(int n, int W[][MAX], set_of_edges &F) {
    int i, vnear, min, touch[n + 1], length[n + 1];
    edge_pointer e;

    F.clear(); // F = Ø;
    for (i = 2; i <= n; i++) {
        touch[i] = 1;
        length[i] = W[1][i];
    }
```

ALGORITHM 4.3: Dijkstra's Algorithm (continued)

```
        repeat (n - 1 times) {
            min = INF;
            for (i = 2; i <= n; i++)
                if (0 <= length[i] && length[i] < min) {
                    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;
                }
            length[vnear] = -1;
        }
    }
```

4.2 Dijkstra's Algorithm for Single-Source Shortest Paths

W	1	2	3	4	5
1	0	7	4	6	1
2	∞	0	∞	∞	∞
3	∞	2	0	5	∞
4	∞	3	∞	0	∞
5	∞	∞	∞	1	0

	i	2	3	4	5	*e*
init:	touch[i]	1	1	1	1	
	length[i]	7	4	6	1	
step 1:	touch[i]	1	1	5	1	(1, 5, 1)
	length[i]	7	4	2	-1	
step 2:	touch[i]	4	1	5	1	(5, 4, 1)
	length[i]	5	4	-1	-1	
step 3:	touch[i]	4	1	5	1	(1, 3, 4)
	length[i]	5	-1	-1	-1	
step 4:	touch[i]	4	1	5	1	(4, 2, 3)
	length[i]	-1	-1	-1	-1	

- The Lengths of Shortest Paths:
  - Algorithm 4.3 determines only the edges in the shortest paths.
    - It does not produce *the lengths of those paths*.
  - These lengths could be obtained from the edges.
    - Alternatively, they can be computed and stored in an array as well.

- Time Complexity of Algorithm 4.3
  - is the same with that of Algorithm 4.1 (Prim's Algorithm)
  - $T(n) = 2(n-1)^2 \in \Theta(n^2)$

4.3 Scheduling Problem

- The ***Scheduling*** Problem:
  - The *time in the system* is
    - the time spent both waiting and being served.
  - The problem of *minimizing* the *total time in the system*
    - has many applications.
    - ex) scheduling users' access to a bank counter or a disk drive.
  - You would learn it in detail
    - when you study the schedulers of operating system.

4.3 Scheduling Problem

- ***Scheduling with Deadlines***:
  - Another scheduling problem
    - occurs when *each job* takes the *same amount of time* to complete,
    - but has a *deadline* by which it must start to *yield a profit*
      - associated with the job.
  - The goal is
    - to schedule the jobs to *maximize* the *total profit*.

4.3 Scheduling Problem

- The Problem of *Scheduling with Deadlines*:
  - Each job *takes one unit* of time to finish
    - and has a *deadline* and a *profit*.
  - If the job starts *before or at* its deadline, the profit is obtained.
    - Therefore, not all jobs have to be scheduled.
  - A schedule is called ***impossible***,
    - if a job is scheduled *after its deadline*.
  - We need not consider any impossible schedule
    - because the job in that schedule does not yield any profit.

4.3 Scheduling Problem

Job	Deadline	Profit
1	2	30
2	1	35
3	2	25
4	1	40

1	2	3	4

Job 1	Job 2	(impossible)
Job 1	Job 3	
Job 1	Job 4	(impossible)
Job 2	Job 1	
Job 2	Job 3	
Job 2	Job 4	(impossible)

Schedule	Total Profit
[1, 3]	55
[2, 1]	65
[2, 3]	60
[3, 1]	55
[4, 1]	70 **(optimal)**
[4, 3]	65

$$profit([1,3]) = 30 + 25 = 55$$

$$profit([4,1]) = 40 + 30 = 70$$

- **The Greedy Approach to the Problem:**
  - To consider all schedules, a brute-force approach,
    - takes *factorial* time. (worse than exponential)
  - A reasonable greedy approach for solving the problem would be:
    - First, *sort* the jobs in *non-increasing* order *by profit*.
    - Next, *inspect* each job in sequence
      - and *add* it to the *schedule* if it is *possible*.

4.3 Scheduling Problem

- The Greedy Approach to the Problem:
  - A *sequence* is called a ***feasible sequence***
    - if all the jobs in the sequence *start by their deadlines*.
    - ex) [4, 1]: feasible sequence, [1, 4]: not a feasible sequence.
  - A *set of jobs* is called a ***feasible set***
    - if there exists *at least one feasible sequence* for the jobs in the set.
    - ex) {1, 4}: feasible set, {2, 4} not a feasible set.
  - Our goal is to find an *optimal* sequence,
    - which is *a feasible sequence with maximum total profit*.
  - An *optimal set of jobs* is the set of jobs in an optimal sequence.

4.3 Scheduling Problem

- High-level greedy algorithm for the problem:

```
sort the jobs in nonincreasing order by profit;
S = ∅;
while (the instance is not solved) {
    select next job;
    if (S is feasible with this job added)
        add this job to S;
    if (there are no more jobs)
        the instance is solved;
}
```

4.3 Scheduling Problem

Job	Deadline	Profit
1	3	40
2	1	35
3	1	30
4	3	25
5	1	20
6	3	15
7	2	10

1. $S = \phi$
2. $S = \{1\}$, [1] is feasible
3. $S = \{1, 2\}$, [2, 1] is feasible
4. $S = \{1, 2, 3\}$, rejected
   there is no feasible sequence for this set : $S = \{1, 2\}$
5. $S = \{1, 2, 4\}$, [2, 1, 4] is feasible
6. $S = \{1, 2, 4, 5\}$, rejected
   $S = \{1, 2, 4\}$
7. $S = \{1, 2, 4, 6\}$, rejected
   $S = \{1, 2, 4\}$
8. $S = \{1, 2, 4, 7\}$, rejected
   $S = \{1, 2, 4\}$ (feasible set) [2, 1, 4] (feasible sequence)

4.3 Scheduling Problem

- An efficient way to *determine* whether a set is *feasible*:
  - Lemma:
    - Let S be *a set of jobs*, then S is *feasible*
      - *if and only if* the sequence obtained by ordering
      - the jobs in S according to *nondecreasing deadlines*
      - is *feasible*.

$S = \{1,\ 2,\ 4,\ 7\}$: feasible?

not feasible

We need only check the feasibility of the sequence:

[2, 7, 1, 4]

1 2 3 3

not feasible

4.3 Scheduling Problem

ALGORITHM 4.4: Scheduling with Deadlines

```
void schedule(int n, int dealine[], sequence_of_integer &J) {
    int i;
    sequence_of_integer K;

    J = [1];
    for (i = 2; i <= n; i++) {
        K = J with i added according to nondecreasing values of deadline[i];
        if (K is feasible)
            J = K;
    }
}
```

4.3 Scheduling Problem

Job	Deadline	Profit
1	3	40
2	1	35
3	1	30
4	3	25
5	1	20
6	3	15
7	2	10

The jobs are already sorted by the profit

1. $J = [1]$
2. $K = [2,1]$, K is feasible
   $J = [2,1]$
3. $K = [2,3,1]$ is rejected, because K is not feasible
4. $K = [2,1,4]$, K is feasible
   $J = [2,1,4]$
5. $K = [2,5,1,4]$ is rejected
6. $K = [2,1,4,6]$ is rejected
7. $K = [2,7,1,4]$ is rejected

- $J = [2, 1, 4]$ is the final result
- $Total\ Profit = 35 + 40 + 25 = 100$

4.3 Scheduling Problem

```
typedef vector<int> sequence_of_integer;

bool is_feasible(sequence_of_integer K, int deadline[]) {
    vector<int>::iterator it = K.begin();
    for (int i = 1; it != K.end(); it++, i++)
        if (i > deadline[*it])
            return false;
    return true;
}
```

```
// K = J
K.resize(J.size());
copy(J.begin(), J.end(), K.begin());
// with i added according to nondecreasing values of deadline[i];
vector<int>::iterator it = K.begin();
while (deadline[*it] <= deadline[i])
    it++;
if (it > K.end())
    K.push_back(i);
else
    K.insert(it, i);

if (is_feasible(K, deadline)) {
    // J = K
    J.resize(K.size());
    copy(K.begin(), K.end(), J.begin());
}
```

4.3 Scheduling Problem

- Time Complexity of Algorithm 4.4 (Worst-Case)
  - Basic Operation: the operation of comparisons
    to sort, to do $K = J$, and to check if K is feasible.
  - Input Size: n, the *number of jobs*.
  - In each iteration of the `for`$-i$ loop, we need to do
    - at most $i - 1$ comparisons to add the ith job of K,
    - and at most i comparisons to check if K is feasible.
  - Therefore, the worst case is

$$W(n) = \sum_{i=2}^{n} [(i-1) + i] = n^2 - 1 \in \Theta(n^2)$$

4.4 Huffman Code

- The problem of *data compression*
  - is to find an *efficient method* for *encoding a data file*.
  - A *binary code* is a common way to represent a data file.
  - A *codeword* is a *unique binary string*
    - representing *each character* in a binary code.
  - A ***fixed-length*** binary code
    - represents each character using the *same number of bits*.
  - A ***variable-length*** binary code is a more efficient coding.
    - It represents different characters using *different number of bits*.

4.4 Huffman Code

$$File = ababcbbbc, character\ set = \{a, b, c\}$$

Character	Fixed-lengh Binary Code
a	00
b	01
c	10

a	b	a	b	c	b	b	b	c
00	01	00	01	10	01	01	01	10

- It takes 18 bits with this encoding

Character	Variable-length Binary Code
a	10
b	0
c	11

a	b	a	b	c	b	b	b	c
10	0	10	0	11	0	0	0	11

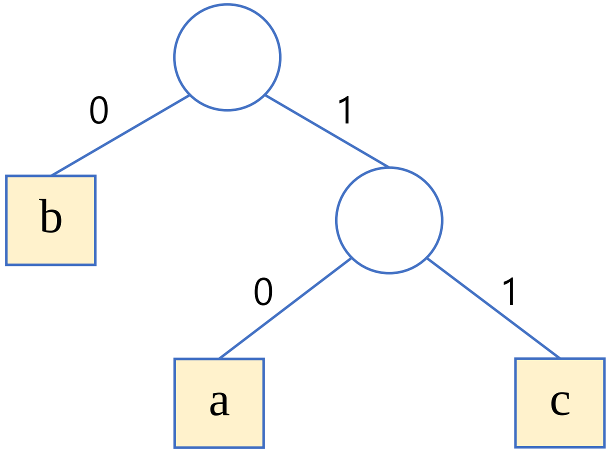
- 'b' occurs *most frequently*:
  - Encode 'b' with one bit (0)
- Encode 'a' and 'c' starting with 1 to distinguish from ' bb'
  - 'a': 10, 'c': 11
- It takes only 13 bits with this encoding

4.4 Huffman Code

- The Problem of the ***Optimal Binary Code***:
  - Given a file (or a string of characters),
    - find a *binary character code* for the characters in the file,
    - which represents the file in the *least number of bits*.
  - We discuss the encoding method, called *Huffman code*,
    - then we develop *Huffman's algorithm* for solving this problem.

- Prefix Code
  - is one particular type of variable-length code.
  - In a prefix code, *no codeword* for one character
    - constitutes the *beginning of the codeword* for another character.
  - Every prefix code can be represented by
    - a *binary tree* whose *leaves* are *the characters* that are to be encoded.
  - The advantage of a prefix code is that
    - we *need not look ahead* when parsing the file.

- b: 0
- a: 10
- c: 11



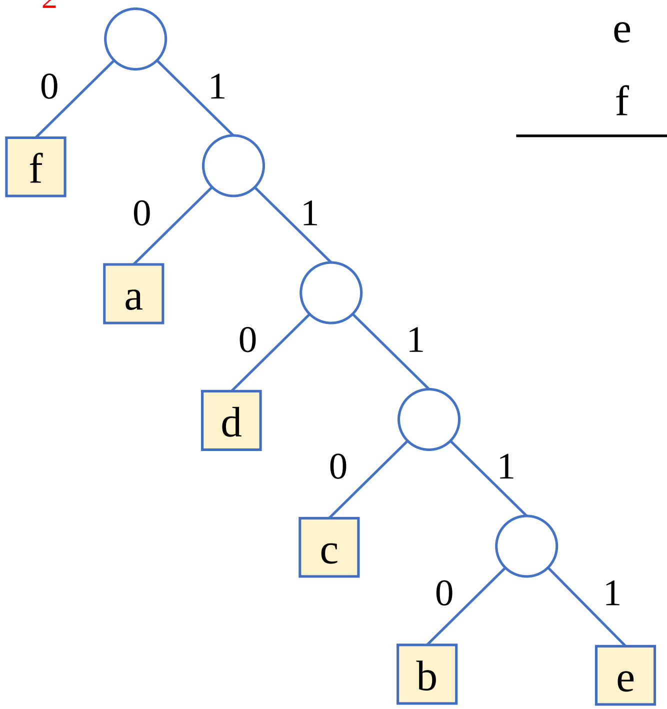
4.4 Huffman Code

$S = \{a, b, c, d, e, f\}$

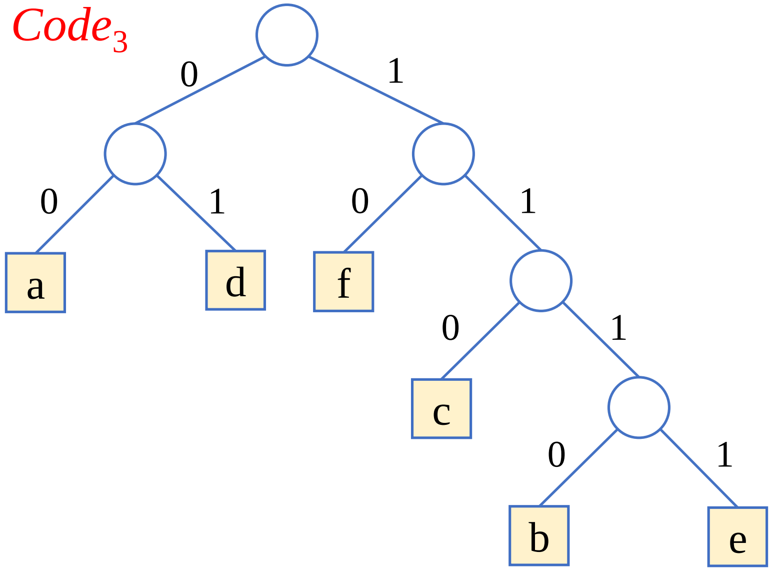
Character	Frequency	$Code_1$	$Code_2$	$Code_3$
a	**16**	000	10	00
b	**5**	001	11110	1110
c	**12**	010	1110	110
d	**17**	011	110	01
e	**10**	100	11111	1111
f	**25**	101	0	10

- $Code_1$: Fixed-Length
- $Code_3$: Huffman code

$Code_2$



$Code_3$



4.4 Huffman Code

- Computing the number of bits for encoding:
  - Given the binary tree T corresponding to some code
    - the number of bits it takes to encode a file is given by
    - $bits(T) = \sum_{i=1}^{n} frequency(v_i) \times depth(v_i)$
      - where $\{v_1, v_2, \cdots, v_n\}$ is the set of characters in the file,
      - $frequency(v_i)$ is the number of times v_i occurs in the file,
      - and $depth(v_i)$ is the depth of v_i in T.

- $bits(Code_1) = 255$
- $bits(Code_2) = 231$
- $bits(Code_3) = 212$

4.4 Huffman Code

- Huffman's Algorithm
  - Huffman developed a *greedy* algorithm
    - that produces an *optimal binary character code* by constructing
    - a *Huffman code*, a *binary tree* corresponding to an *optimal code*.

  - We need a *type declaration* for the node of binary tree.
  - We also need to use a *priority queue*
    - in which the character with the *lowest frequency* is *removed next*.
    - It can be implemented as a *min-heap*.

```c
typedef struct node *node_pointer;
typedef struct node {
    char symbol;    // the value of a character.
    int frequency;  // the number of times the character is in the file.
    node_pointer left;
    node_pointer right;
} nodetype;


node_pointer create_node(char symbol, int frequency) {
    node_pointer node = (node_pointer)malloc(sizeof(nodetype));
    node->symbol = symbol;
    node->frequency = frequency;
    node->left = node->right = NULL;
    return node;
}
```

4.4 Huffman Code

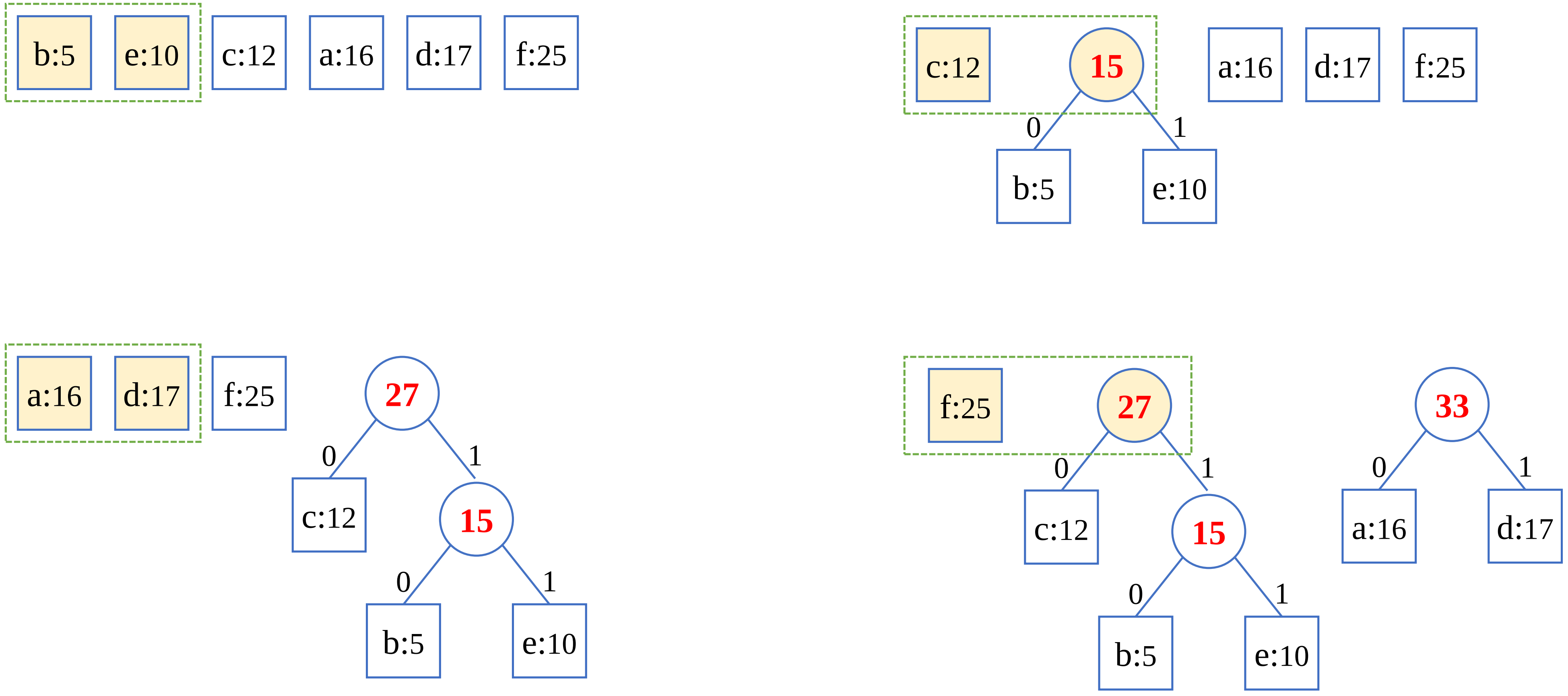

```
struct compare {
    bool operator()(node_pointer p, node_pointer q) {
        if (p->frequency > q->frequency)
            return true;
        return false;
    }
};
```

```
int n, frequency[MAX];
char symbol[MAX];

priority_queue<node_pointer, vector<node_pointer>, compare> PQ;
for (int i = 0; i < n; i++)
    PQ.push(create_node(symbol[i], frequency[i]));
```

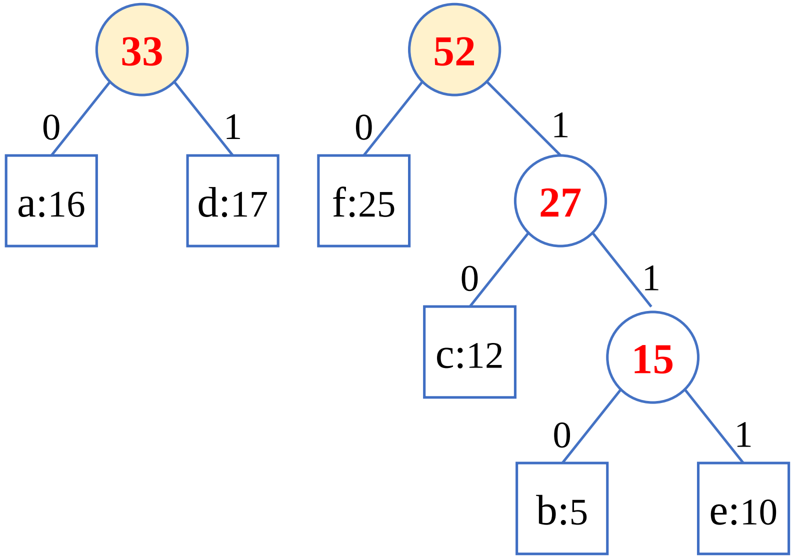
- High-level pseudo-code for the Huffman's algorithm:

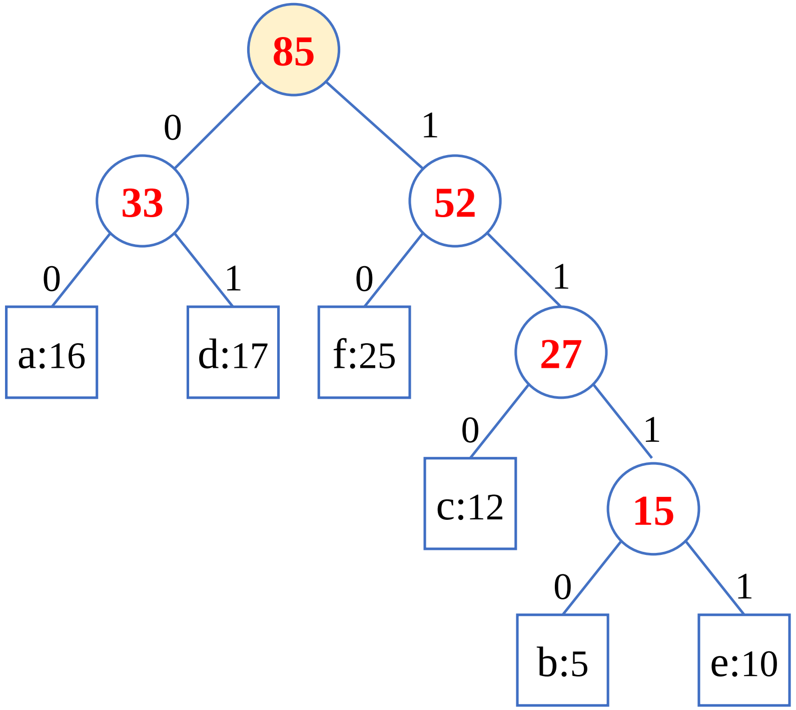
```
n = number of characters in the file;
Arrange n pointers to nodetype records in a PQ;

for (i = 1; i <= n - 1; i++) {
    remove(PQ, p);
    remove(PQ, q);
    r = new nodetype;
    r->left = p;
    r->right = q;
    r->frequency = p->frequency + q->frequency;
    insert(PQ, r);
}
remove(PQ, r);
return r;
```

4.4 Huffman Code



4.4 Huffman Code





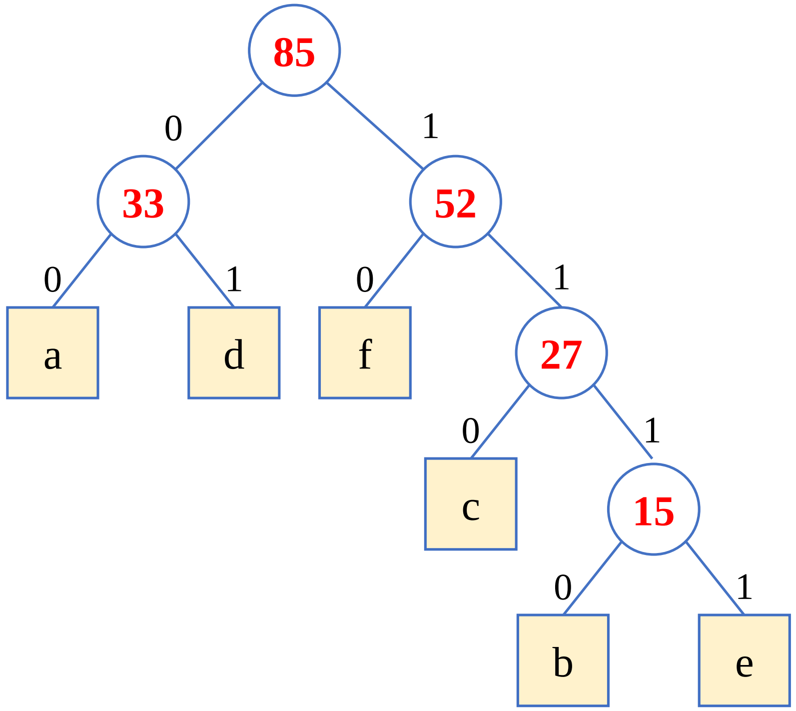
4.4 Huffman Code

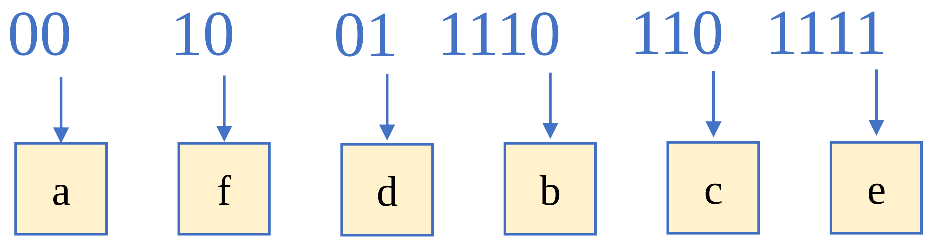
- Time Complexity of Huffman's Algorithm
  - If a priority queue is implemented *as a min-heap*,
    - each heap operation (insert & remove) requires $\Theta(\lg n)$ time.
  - Since there are $n-1$ passes through the `for`-i loop,
    - the algorithm runs in $\Theta(n \lg n)$ time.

- It is *provable* that
  - Huffman's algorithm always produces an optimal binary code.
  - based on Lemma:
    - The binary tree corresponding to an *optimal binary prefix code* is *full*.
    - That is, *every nonleaf node has two children*.

4.4 Huffman Code

- How to *decode* an encoded binary string?
  - Given with an encoded binary string, 0010011111011011111,
    - you can *traverse* the binary tree to decode it into $afdbce$.





4.5 The Knapsack Problem: Greedy .vs. D.P.

- The *greedy approach* and *dynamic programming*
  - are *two ways* to solve *optimization problems*.
  - For example, the Single-Source-Shortest-Paths problem
    - is solved using *dynamic programming* in Algorithm 3.3 (Floyd's)
    - and is solved using the *greedy approach* in Algorithm 4.3 (Dijkstra's).
  - However, the D.P. algorithm is *overkill* in that
    - it produces the shortest paths from all sources.
    - Floyd's (*D.P.*): $\Theta(n^3)$, Dijkstra's (*Greedy*): $\Theta(n^2)$.
  - *Often* when the *greedy* approach solves a problem,
    - the result is a *simpler*, *more efficient* than *dynamic programming*.

4.5 The Knapsack Problem: Greedy .vs. D.P.

- The *greedy approach* and *dynamic programming*
  - On the other hand, it is usually *more difficult* to determine
    - whether a *greedy* algorithm *always produces* an *optimal* solution.
    - A proof is needed to show that it does.
  - In the case of *dynamic programming*, we need only determine
    - whether the *principle of optimality* applies.

4.5 The Knapsack Problem: Greedy .vs. D.P.

- Two Similar Problems for the ***Knapsack Problem***:
  - The *Fractional Knapsack Problem*
    - concerns a thief breaking into a jewelry store carrying a knapsack.
    - In this case, the thief does not have to steal all of an item,
      - but rather can take *any fraction of the item*.
    - We can think of the items as being *bags of gold or silver dust*.
  - The 0-1 *Knapsack Problem*
    - In this case, the thief *can not take some fraction of the item*.
    - We can think of the items as *being gold or silver ingots*.

4.5 The Knapsack Problem: Greedy .vs. D.P.

- The Knapsack Problem:
  - The knapsack will break
    - if the *total weight of the items* stolen exceeds some *maximum weight*.
  - Each item has a *value* and a *weight*.
  - The goal of the thief is to *maximize the total value of the items*
    - while *not making* the total weight *exceed the maximum weight* W.

4.5 The Knapsack Problem: Greedy .vs. D.P.

- Formal definition of the Knapsack Problem:
  - Suppose there are n items, and let
    - $S = \{item_1, item_2, \cdots, item_n\}$
    - w_i = weight of $item_i$
    - p_i = profit of $item_i$
    - W = maximum weight the knapsack can hold,
    - where w_i, p_i, and W are positive integers.
  - Then, *determine a subset A of S* such that
    - $\sum_{item_i \in A} p_i$ is *maximized* subject to $\sum_{item_i \in A} w_i \le W$.

4.5 The Knapsack Problem: Greedy .vs. D.P.

- *Greedy Approach* to the 0-1 Knapsack Problem:
  - Our greedy strategy is
    - to choose the items with the *largest profit per unit weight first*.
  - That is, *order the items* in nonincreasing order by the profit/unit weight,
    - and select them in sequence.
  - An item is put in the knapsack
    - if its weight does not bring the *total weight* above W.
  - Note that this strategy can waste some capacities of an item.
    - Therefore, greedy algorithm *does not solve* the 0-1 Knapsack Problem.

4.5 The Knapsack Problem: Greedy .vs. D.P.

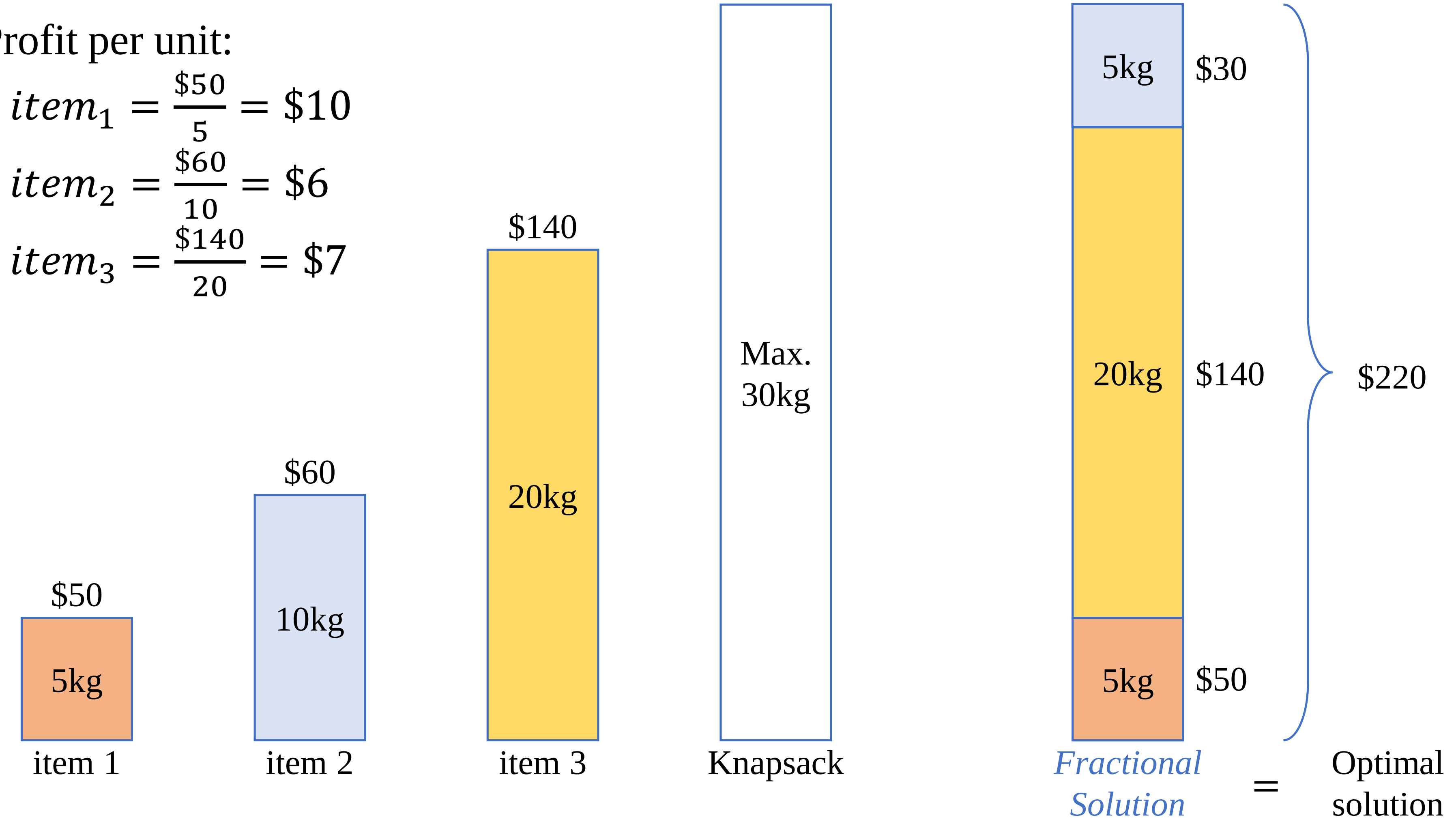
- Profit per unit:
  - $item_1 = \frac{\$50}{5} = \$10$
  - $item_2 = \frac{\$60}{10} = \$6$
  - $item_3 = \frac{\$140}{20} = \$7$

4.5 The Knapsack Problem: Greedy .vs. D.P.

- *Greedy Approach* to the *Fractional* Knapsack Problem:
  - If our greedy strategy is again
    - to *choose* the items with the *largest profit per unit weight first*,
    - all of $item_1$ and $item_3$ will be taken as before.
  - However, we can use
    - the 5 kg of remaining capacity to take $5/10$ of $item_2$.
  - Our total profit is
    - $\$50 + \$140 + \frac{5}{10} \times (\$60) = \$220$.
  - Our greedy algorithm never wastes any capacity.
    - It is *provable* that it *always* yields an *optimal* solution.

4.5 The Knapsack Problem: Greedy .vs. D.P.

- Profit per unit:
  - $item_1 = \frac{\$50}{5} = \$10$
  - $item_2 = \frac{\$60}{10} = \$6$
  - $item_3 = \frac{\$140}{20} = \$7$



4.5 The Knapsack Problem: Greedy .vs. D.P.

```
/* Greedy Algorithm for the Fractional Knapsack Problem */
void knapsack1(int n, int W, int w[], int p[]) {
    priority_queue<item_pointer, vector<item_pointer>, compare> PQ;
    initialize(PQ, n, w, p);
    int total_weight = 0;
    while (!PQ.empty()) {
        item_pointer i = PQ.top();
        PQ.pop();
        total_weight += i->weight;
        if (total_weight > W) {
            int diff = total_weight - W;
            int profit = diff * i->unit_profit;
            cout << "(" << i->id << "," << diff << "," << profit << ")" << endl;
            break;
        }
        cout << "(" << i->id << "," << i->weight << "," << i->profit << ")" << endl;
    }
}
```

4.5 The Knapsack Problem: Greedy .vs. D.P.

```
typedef struct item *item_pointer;
typedef struct item {
    int id;
    int weight;
    int profit;
    float unit_profit;
} itemtype;

void initialize(priority_queue<item_pointer, vector<item_pointer>, compare> &PQ,
                int n, int w[], int p[])
{
    for (int i = 1; i <= n; i++) {
        item_pointer item = (item_pointer)malloc(sizeof(itemtype));
        item->id = i;
        item->weight = w[i];
        item->profit = p[i];
        item->unit_profit = (float)p[i] / (float)w[i];
        PQ.push(item);
    }
}
```

4.5 The Knapsack Problem: Greedy .vs. D.P.

- Solving the 0-1 *Knapsack Problem* with *Dynamic Programming*:
  - To show that the *principle of optimality* applies,
    - let A be an optimal subset of the n items.
  - There are two cases: either A *contains* $item_n$ *or not*.
    - If A *does not*, A is equal to an *optimal subset* of the first $n-1$ items.
    - If A *does*, the *total profit* of the items in A is equal to
      - p_n + the *optimal profit* obtained when the items can be chosen from the first $n-1$ items,
      - under the *restriction* that the *total weight cannot exceed* $W - w_n$.
  - Therefore, the *principle of optimality* applies.

4.5 The Knapsack Problem: Greedy .vs. D.P.

- The design of an algorithm using dynamic programming:
  - Let $P[i][w]$ be the *optimal profit* obtained
    - when choosing items only from the first i items
    - under the restriction that the total weight cannot exceed w.
  - $P[i][w] = \begin{cases} \text{maximum}(P[i-1][w],\ p_i + P[i-1][w-w_i]), & \text{if } w_i \le w \\ P[i-1][w], & \text{if } w_i > w \end{cases}$
  - Then, the *maximum profit* is equal to $P[n][W]$.
  - We compute the values in the *rows* of the array P in sequence
    - using the previous expression for $P[i][w]$.
    - The values of $P[0][w]$ and $P[i][0]$ are set to 0.

```
/* Simple dynamic programming for the 0-1 Knapsack Problem */
void knapsack2(int n, int W, int w[], int p[], int P[][MAX]) {

    for (int i = 1; i <= n; i++)
        P[i][0] = 0;
    for (int j = 1; j <= W; j++)
        P[0][j] = 0;

    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= W; j++)
            P[i][j] = (w[i] > j) ? P[i - 1][j] :
                max(P[i-1][j], p[i] + P[i-1][j-w[i]]);
}
```

4.5 The Knapsack Problem: Greedy .vs. D.P.

- Time Complexity of Simple Implementation
  - It is straightforward that
    - the number of array entries computed is $nW \in \Theta(nW)$.
  - Note that there is no relationship between n and W.
  - Therefore, for a given n, it can take arbitrarily large running times
    - by taking arbitrarily large number of W.
    - ex) If $W = n!$, then $nW \in \Theta(n!)$.
  - This algorithm should be improved so that
    - the worst-case number of entries computed is in $\Theta(2^n)$.
  - With this improvement, it never performs
    - worse than the *brute-force algorithm* and often performs much better.

- Enhancing the Simple Algorithm:
  - The improvement is based on the fact that it is not necessary
    - to determine the entries in the ith row for every w between 1 and W.
  - Rather, in the nth row we need only determine $P[n][W]$.
    - The only entries needed in the $(n-1)$st row are the ones needed
      - to compute $P[n][W]$: $P[n-1][W]$ and $P[n-1][W - w_n]$.
  - We continue to backward from n to determine which entries are needed.
  - That is, determine entries needed in the ith row,
    - determine entries needed in the $(i-1)$st row using the fact that
    - $P[i][w]$ is computed from $P[i-1][w]$ and $P[i-1][w - w_n]$.
  - We *stop* when $n = 1$ or $w \leq 0$.

4.5 The Knapsack Problem: Greedy .vs. D.P.

- $n = 3,\ W = 30$
- $w = [\,5,\ 10,\ 20\,]$
- $p = [\,50,\ 60,\ 140\,]$

$P[3][W] = P[3][30]$

$P[3-1][30]$
$= P[2][30]$

$P[3-1][30-w_3]$
$= P[2][10]$

$P[2-1][30]$
$= P[1][30]$

$P[2-1][30-w_2]$
$= P[1][20]$

$P[2-1][10]$
$= P[1][10]$

$P[2-1][10-w_2]$
$= P[1][0]$

\$50 \$50 \$50 \$0

$$P[1][w] = \begin{cases} \max(P[0][w], \quad \$50 + P[0][w-5]), & \text{if } w_1 = 5 \le w \\ P[0][w], & \text{if } w_1 = 5 > w \end{cases}$$

4.5 The Knapsack Problem: Greedy .vs. D.P.

$$P[3][W] = P[3][30]$$

$$P[3-1][30] = P[2][30] \quad \$110$$

$$P[3-1][30-w_3] = P[2][10] \quad \$60$$

$$P[2][30] = \begin{cases} \max(P[1][30], \quad \$60 + P[1][20]), & \text{if } w_2 = 10 \leq 30 \\ P[1][30], & \text{if } w_2 = 10 > 30 \end{cases}$$

$$P[2][10] = \begin{cases} \max(P[1][10], \quad \$60 + P[1][0]), & \text{if } w_2 = 10 \leq 10 \\ P[1][10], & \text{if } w_2 = 10 > 10 \end{cases}$$

4.5 The Knapsack Problem: Greedy .vs. D.P.

$$P[3][W] = P[3][30] \quad \color{red}{\$200}$$

$$P[3][30] = \begin{cases} \max(P[2][30], \quad \$140 + P[2][10]), & \text{if } w_3 = 20 \leq 30 \\ P[2][30], & \text{if } w_1 = 20 > 30 \end{cases}$$

4.5 The Knapsack Problem: Greedy .vs. D.P.

- The Efficiency of the Improved Algorithm:
  - Notice that we compute at most 2^i entries in the $(n-1)$th row.
  - Therefore, *at most* the *total number of entries computed* is
    - $1 + 2 + 2^i + \cdots + 2^{n-1} = 2^n - 1 \in \Theta(2^n)$.
  - Consider a bound *in terms of n and W combined*.
  - It is provable that if $n = W + 1$ and $w_i = 1$ for all i,
    - then the total number of entries computed is about
    - $1 + 2 + \cdots + n = \frac{n(n+1)}{2} = \frac{(W+1)(n+1)}{2} \in \Theta(nW)$.
  - Combining these two results,
    - the worst-case number of entries computed is $O(minimum(2^n, nW))$

4.5 The Knapsack Problem: Greedy .vs. D.P.

- Space Complexity of the Improved Algorithm:
  - We do not need to create the *entire array* to implement the algorithm.
    - Instead, we can *store* just *the entries* that are *needed*.
  - Then, the worst-case memory usage has the same bounds
    - $O(minimum(2^n, nW))$.

4.5 The Knapsack Problem: Greedy .vs. D.P.

```
/* Enhanced dynamic programming for the 0-1 Knapsack Problem */
int knapsack3(int n, int W, int w[], int p[], map<pair<int, int>, int> &P) {
    if (n == 0 || W <= 0)
        return 0;

    int lvalue = (P.find(make_pair(n-1, W)) != P.end()) ?
        P[make_pair(n-1, W)] : knapsack3(n-1, W, w, p, P);
    int rvalue = (P.find(make_pair(n-1, W-w[n])) != P.end()) ?
        P[make_pair(n-1, W)] : knapsack3(n-1, W-w[n], w, p, P);
    P[make_pair(n, W)] = (w[n] > W) ? lvalue : max(lvalue, p[n] + rvalue);
    return P[make_pair(n, W)];
}
```

Any Questions?