Chapter 4

Greedy Approach

روش حریصانه
The idea

- Grab data items in sequence, each time taking the “best” one without regard for the choices made before or in the future.
- Often used to solve optimization problems.
- Must determine that the globally optimal solution can be obtained by a sequence of locally optimal solution.
An example:
Making changes

U.S. coins (penny(1), nickel(5), dime(10), quarter(25), half dollar(50))
Making changes

- If the coins consist of U.S. Coins (25 10 5 1) and if there is at least one type of each coin available, the greedy algorithm always returns an optimal solution when a solution exists.
The algorithm

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)
        // solution check
            the instance is solved;

}
When the greedy approach does not work if we include a 12-cent coin with the U.S. coins, the greedy algorithm does not always give an optimal solution.

Greedy solution: $16 = 12 + 1 + 1 + 1 + 1$

Optimal solution: $16 = 10 + 5 + 1$
Basic components in Greedy approach

- A *selection procedure* chooses the next item to add to the set. The selection is performed according to a *greedy criterion* that satisfies some locally optimal consideration at the time.

- A *feasibility check* determines if the new set is feasible by checking whether it is possible to complete this set in such a way as to give a solution to the instance.

- A *solution check* determines whether the new set constitutes a solution to the instance.
Minimum spanning trees

- Some terms:
  - undirected graph
  - path in an undirected graph
  - connected graph
  - simple cycle
  - acyclic graph
  - tree (free tree)
    - acyclic, connected, undirected graph
  - rooted tree
Examples

(a) A connected, weighted, undirected graph $G$.

(b) If $(v_4, v_5)$ were removed from this subgraph, the graph would remain connected.

(c) A spanning tree for $G$.

(d) A minimum spanning tree for $G$.

Figure 4.3 • A weighted graph and three subgraphs.
Minimum spanning tree

- Spanning tree
- Minimum spanning tree
- Formal definition of an undirected graph

Definition
An undirected graph $G$ consists of a finite set $V$ whose members are called the vertices of $G$, together with a set $E$ of pairs of vertices in $V$. These pairs are called the edges of $G$. We denote $G$ by

$$G = (V, E).$$
Finding minimum spanning tree

To find a minimum $T = (V, F)$ for $G = (V, E)$

$F = \emptyset$

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;

    if ($T = (V, F)$ is a spanning tree)
        the instance is solved;
}

// Initialize set of edges to empty.

// selection procedure

// feasibility check

// solution check
Prim’s algorithm

\[ F = \emptyset \]  // Initialize set of edges to empty.

\[ Y = \{v_1\} \]  // Initialize set of vertices to contain only the first one.

while (the instance is not solved)
  select a vertex in \( V - Y \) that is nearest to \( Y \);
  // selection procedure and feasibility check
  add the vertex to \( Y \);
  add the edge to \( F \);

if (\( Y == V \))  // solution check the instance is solved;
}
An example

Determine a minimum spanning tree.

1. Vertex $v_1$ is selected first.
An example

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

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

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

5. Vertex $v_4$ is selected.
Adjacency matrix

\[
W[i][j] = \begin{cases} 
\text{weight on edge} & \text{if there is an edge between } v_i \text{ and } v_j \\
\infty & \text{if there is no edge between } v_i \text{ and } v_j \\
0 & \text{if } i = j.
\end{cases}
\]

\[
\begin{array}{ccccc}
 & 1 & 2 & 3 & 4 & 5 \\
1 & 0 & 1 & 3 & \infty & \infty \\
2 & 1 & 0 & 3 & 6 & \infty \\
3 & 3 & 3 & 0 & 4 & 2 \\
4 & \infty & 6 & 4 & 0 & 5 \\
5 & \infty & \infty & 2 & 5 & 0 \\
\end{array}
\]

Figure 4.5  ● The array representation \( W \) of the graph in Figure 4.3(a).
Prim's algorithm

\[ nearest[i] = \text{index of the vertex in } Y \text{ nearest to } v_i \]

\[ distance[i] = \text{weight on edge between } v_i \text{ and the vertex indexed by } nearest[i] \]

Algorithm 4.1: Prim's Algorithm

```c
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 = Ø;
    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
    }
...```

Prim’s algorithm (cont’d)

**repeat** *(n - 1 times)*{  // Add all n - 1 vertices to Y.
    \( min = \infty; \)
    **for** *(i = 2; i <= n; i++)* // Check each vertex for being nearest to Y.
    \[\text{if} \ (0 \leq distance[i] < min) \{\]
    \( \quad \text{min} = distance[i]; \)
    \( \quad \text{vnear} = i; \)
    \[\} \]
    \( e = \text{edge connecting vertices indexed by vnear and nearest[vnear]}; \)
    add e to F;
    \( distance[vnear] = -1; \) // Add vertex indexed by
    **for** *(i = 2; i <= n; i++)* // vnear to Y.
    \[\text{if} \ (W[i][vnear] < distance[i]) \{ \]
    \( \quad \text{distance[i]} = W[i] [vnear]; \) // Y, update its distance
    \( \quad \text{nearest [i]} = \text{vnear}; \) // from Y.
    \[\} \]
}
An example

\[ F = \emptyset; \]
\[
\text{for } (i = 2; i <= n; i++)\{
\]
\[
\text{nearest } [i] = 1; \quad \text{// For all vertices, initialize } v_1
\]
\[
\text{distance } [i] = W[1][i]; \quad \text{// to be the nearest vertex in}
\]
\[
\}
\]
repeat \( \text{(n - 1 times)} \) {
    \( \text{min} = \infty; \)
    \( \text{for} \ (i = 2; \ i \leq n; \ i+++) \)
    \( \text{if} \ (0 \leq \text{distance}[i] < \text{min}) \{ \)
    \( \text{min} = \text{distance}[i]; \)
    \( \text{vnear} = i; \}\)

    \( e = \text{edge connecting vertices indexed by vnear and nearest[vmear];} \)
    \( \text{add } e \text{ to } F; \)
    \( \text{distance[vmear]} = -1; \)
    \( \text{for} \ (i = 2; \ i \leq n; \ i+++) \)
    \( \text{if} \ (W[i][vnear] < \text{distance}[i]) \{ \)
    \( \text{distance}[i] = W[i][vnear]; \)
    \( \text{nearest } [i] = vnear; \}\)
}
An example

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;
        }
    }
}
repeat \((n - 1\) times)\{
    \text{min} = \infty;
    \text{for} (i = 2; i \leq n; i++)
    \text{if} (0 \leq \text{distance}[i] < \text{min})\
    \text{\{ min = distance}[i];
    \text{vnear} = i; \text{\}} \\
    \text{e = edge connecting vertices indexed by vnear and nearest[vnear];}
    \text{add e to F;}
    \text{distance[vnear] = -1;}
    \text{for} (i = 2; i \leq n; i++)
    \text{if} (W[i][vnear] < \text{distance}[i])\
    \text{\{ distance}[i] = W[i][vnear];
    \text{nearest}[i] = vnear; \text{\}}
\}
repeat \( (n - 1 \text{ times}) \) {
\[ \text{min} = \infty; \]
\[ \text{for} \ (i = 2; \ i \leq n; \ i++) \]
\[ \text{if} \ (0 \leq \text{distance}[i] < \text{min}) \{ \]
\[ \text{min} = \text{distance}[i]; \]
\[ \text{vnear} = i; \} \]
\[ e = \text{edge connecting vertices indexed by vnear and nearest[vnear];} \]
\[ \text{add } e \text{ to } F; \]
\[ \text{distance}[\text{vnear}] = -1; \]
\[ \text{for} \ (i = 2; \ i \leq n; \ i++) \]
\[ \text{if} \ (W[i][\text{vnear}] < \text{distance}[i]) \{ \]
\[ \text{distance}[i] = W[i][\text{vnear}]; \]
\[ \text{nearest}[i] = \text{vnear}; \} \]
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
- Then:

\[
T(n) = 2(n-1)(n-1) \in \Theta(n^2)
\]
Proof

Lemma 4.1 Let $G = (V, E)$ be a connected, weighted, undirected graph; let $F$ be a promising subset of $E$; and let $Y$ be the set of vertices connected by the edges in $F$. If $e$ is an edge of minimum weight that connects a vertex in $Y$ to a vertex in $V - Y$, then $F \cup \{e\}$ is promising.

Theorem 4.1: Prim’s algorithm always produces a minimum spanning tree
Kruskal's algorithm

\( F = \emptyset; \) \hspace{1cm} // Initialize set of
\hspace{1cm} // 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; \hspace{1cm} // selection procedure

    if (the edge connects two vertices in
        disjoint subsets) {
        merge the subsets;
        add the edge to \( F \);
    }

    if (all the subsets are merged) \hspace{1cm} // solution check
        the instance is solved;
}


An illustration (1)

Determine a minimum spanning tree.

1. Edges are sorted by 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
An illustration (2)

2. Disjoint set are created.

3. Edge $(v_1, v_2)$ is selected.
An illustration (3)

4. Edge \((v_3, v_5)\) is selected.

5. Edge \((v_1, v_3)\) is selected.
An illustration (4)

6. Edge $(v_2, v_3)$ is selected.

7. Edge $(v_3, v_4)$ is selected.
Data types and operations

- Data types
  - index \( i \);
  - set_pointer \( p, q \);

- Operations
  - \textit{initial}(n) initializes \( n \) disjoint subsets, each of which contains exactly one of the indices between 1 and \( n \).
  - \( p = \text{find}(i) \) makes \( p \) point to the set containing index \( i \).
  - \textit{merge}(p, q) merges the two sets, to which \( p \) and \( q \) point, into the set.
  - \textit{equal}(p, q) returns true if \( p \) and \( q \) both point to the same set.
void kruskal (int n, int m, set_of_edges E, set_of_edges& F) {
    index i, j;
    set_pointer p, q;
    edge e;
    Sort the \( m \) edges in \( E \) by weight in nondecreasing order;
    \( F = \emptyset; \)
    initial (n); // Initialize \( n \) disjoint subsets.
    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 \( F \);
        }
    }
}
Worst-case time complexity

- Basic operation: a comparison instruction
- Input size: \( n \), the number of vertices, and \( m \), the number of edges

Analysis:
- The time to sort the edges: \( W(m) \in \Theta(mlgm) \)
- The time in the while loop: \( W(m) \in \Theta(mlgm) \)
- The time to initialize \( n \) disjoint sets: \( T(n) \in \Theta(n) \)
- Overall: \( W(m, n) \in \Theta(mlgm) \)
- In the worst case every vertex can be connected to every other vertex \( m=n(n-1)/2 \), Therefore \( W(m,n) = \Theta(n^2\lg n) \)
Proof

Lemma 4.2  Let $G = (V, E)$ be a connected, weighted, undirected graph; let $F$ be a promising subset of $E$; and let $e$ be an edge of minimum weight in $E - F$ such that $F \cup \{e\}$ has no simple cycles. Then $F \cup \{e\}$ is promising.

Theorem 4.2: Kruskal’s algorithm always produces a minimum spanning tree
Comparing Prim’s algorithm with Kruskal’s algorithm

- Prim’s algorithm: \( T(n) \in \Theta(n^2) \)

- Kruskal’s algorithm:
  - \( W(m, n) \in \Theta(mlgm) \)
  - \( n - 1 \leq m \leq n(n-1)/2 \)

- Sparse graph
  - *Kruskal's algorithm should be faster.*

- Highly connected
  - *Prim's algorithm should be faster.*
Dijkstra’s algorithm
shortest paths from one source to all the others

\[ Y = \{v1\}; \]
\[ F = \emptyset; \]
while (the instance is not solved) {
  select a vertex \( v \) from \( V - Y \), that has a shortest path from \( v1 \), using only vertices in \( Y \) as intermediates;
  //selection procedure and 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
}
An example (1)

Compute shortest paths from $v_1$. 

Diagram: A graph with nodes $v_1$, $v_2$, $v_3$, $v_4$, and $v_5$. Edges and their weights are labeled: $v_1$ to $v_5$ with weight 6, $v_1$ to $v_4$ with weight 1, $v_1$ to $v_2$ with weight 7, $v_2$ to $v_3$ with weight 2, $v_4$ to $v_3$ with weight 5.
An example (2)

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.
An example (3)

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

- $Touch[i] = \text{index of vertex } v \text{ in } Y \text{ such that the edge } <v, v_i> \text{ is the last edge on the current shortest path from } v_1 \text{ to } v_i \text{ using only vertices in } Y \text{ as intermediates}$

- $length[i] = \text{length of the current shortest path from } v_1 \text{ to } v_i \text{ using only vertices in } Y \text{ as intermediates}$
The algorithm (1)

- Algorithm 4.3: Dijkstra's Algorithm

```c
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 = Ø;
    for (i = 2; i<= n; i++ ){
        touch [i] = 1;
        length [i] = W[1] [i];
    }
```
The algorithm (2)

repeat (n - 1 times){
    min = ∞;
    for (i = 2; i <= n; i++)
        if (0 ≤ 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;
}
Compute shortest paths from $v_1$.

\begin{verbatim}
for (i = 2; i <= n; i++){
    touch[i] = 1;
    length[i] = W[1][i];
}
\end{verbatim}
repeat \( (n - 1) \) times
\[
\begin{align*}
\text{min} &= \infty; \\
\text{for } &\quad (i = 2; i \leq n; i++) \\
\text{if } &\quad (0 \leq \text{length}[i] < \text{min}) \\
&\quad \text{min} = \text{length}[i]; \\
&\quad \text{vnear} = i; \\
\end{align*}
\]

\( e = \text{edge} \) from vertex indexed by \( \text{touch}[\text{vnear}] \) to vertex indexed by \( \text{vnear} \); 
add \( e \) to \( F \); 
\( \text{for } (i = 2; i \leq n; i++) \\
\text{if } &\quad (\text{length}[\text{vnear}] + W[\text{vnear}][i] < \text{length}[i]) \\
&\quad \text{length}[i] = \text{length}[\text{vnear}] + W[\text{vnear}][i]; \\
&\quad \text{touch}[i] = \text{vnear}; \\
\end{align*}
\]

\( \text{length}[\text{vnear}] = -1; \)
repeat \((n - 1 \text{ times})\) \{ 
\[ \text{min} = \infty; \]
for \((i = 2; i <= n; i++)\) 
\[ \text{if} \ (0 \leq \text{length}[i] < \text{min}) \{ \]
\[ \text{min} = \text{length}[i]; \]
\[ \text{vnear} = i; \} \]
\[ \text{e} = \text{edge from vertex indexed by touch[vnear]} \to \text{vertex indexed by vnear}; \]
\[ \text{add e to } F; \]
for \((i = 2; i <= n; i++)\) 
\[ \text{if} \ (\text{length[vnear]} + W[vnear][i] < \text{length}[i])\{ \]
\[ \text{length}[i] = \text{length[vnear]} + W[vnear][i]; \]
\[ \text{touch}[i] = \text{vnear}; \} \]
\[ \text{length[vnear]} = -1; \} \]
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.

**Example**

```
repeat (n - 1 times) {
    min = \infty;
    for (i = 2; i <= n; i++)
        if (0 \leq \text{length}[i] < min) {
            min = \text{length}[i];
            vnear = i;
        }
    e = \text{edge} from vertex indexed by \text{touch}[vnear] to vertex indexed by vnear;
    add e to F;
    for (i = 2; i <= n; i++)
        if (\text{length}[vnear] + W[vnear][i] < \text{length}[i]) {
            \text{length}[i] = \text{length}[vnear] + W[vnear][i];
            \text{touch}[i] = vnear;
        }
    \text{length}[vnear] = -1;
}
```
repeat (n - 1 times) {
  min = \infty;
  for (i = 2; i <= n; i++)
    if (0 \leq \text{length}[i] < min) {
      min = \text{length}[i];
      vnear = i;
    }
  e = \text{edge} from vertex indexed by \text{touch}[vnear] to vertex indexed by vnear;
  add e to F;
  for (i = 2; i <= n; i++)
    if (\text{length}[vnear] + W[vnear][i] < \text{length}[i]){
      \text{length}[i] = \text{length}[vnear] + W[vnear][i];
      \text{touch}[i] = vnear;
    }
  \text{length}[vnear] = -1;
}
Every-case time complexity

- $T(n) = 2(n-1)^2 \in (n^2)$
Dijkstra vs Prim

**Dijkstra's Algorithm**

```plaintext
repeat (n - 1 times){
    min = ∞;
    for (i = 2; i <= n; i++)
        if (0 ≤ 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;
}
```

**Prim's Algorithm**

```plaintext
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;
        }
}
```
Scheduling

- Two types of scheduling:
  - Minimize the total time in \textit{waiting} and \textit{being served} (time in the system)
  - Scheduling with deadlines

- Compute the optimal scheduling:
  \[ t_1 = 5, \ t_2 = 10, \ \text{and} \ t_3 = 4 \]
  three jobs and their service times
The algorithm

sort the jobs by service time in nondecreasing order;

while (the instance is not solved) {
    schedule the next job; // selection procedure
    // feasibility check

    if (there are no more jobs) // solution check
        the instance is solved;
}

Proof

- Time complexity: $W(n) \in \Theta(n \lg n)$
- Theorem 4.3
  The only schedule that minimizes the total time in the system is one that schedules jobs in nondecreasing order by service time
Multiple-server scheduling problem

- Server 1 serves jobs 1, (1+m), (1+2m), (1+3m), ...
- Server 2 serves jobs 2, (2+m), (2+2m), (3+3m), ...
- ...
- Server $i$ serves jobs $i$, (i+m), (i+2m), (i+3m), ...
- ...
- Server $m$ serves jobs $m$, (m+m), (m+2m), (m+3m), ...
Scheduling with deadlines

- In this scheduling problem, 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.

<table>
<thead>
<tr>
<th>Job</th>
<th>Deadline</th>
<th>Profit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>30</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>35</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>25</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>40</td>
</tr>
</tbody>
</table>

- Goal = Maximize the total profit
Some terms

- **Feasible sequence**
  - all the jobs in the sequence start by their deadlines

- **Feasible set**
  - there exists at least one feasible sequence for the jobs in this set.

- **Optimal sequence**
  - a feasible sequence with maximum total profit.
Check for feasibility

- Lemma-4.3 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.

- Example: Determine whether $\{1, 2, 4, 7\}$ is feasible.
The algorithm

sort the jobs in nonincreasing order by profit;

\[ S = \emptyset \]

while (the instance is not solved){
    select next job; // selection procedure
    if (\( S \) is feasible with this job added)
        add this job to \( S \);
    if (there are no more jobs)
        the instance is solved;
}
Algorithm 4.4: Scheduling with Deadlines

```c
void schedule (int n, const int deadline [],
    sequence_of_integer& j)
{
    //sorted jobs in nonincreasing order by profit
    index 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;
    }
}
```
**Example**

<table>
<thead>
<tr>
<th>Job</th>
<th>Deadline</th>
<th>Profit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>40</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>35</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>30</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>25</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>15</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>10</td>
</tr>
</tbody>
</table>
Example


2. $K$ is set to $[2, 1]$ and is determined to be feasible.

   $J$ is set to $[2, 1]$ because $K$ is feasible.

3. $K$ is set to $[2, 3, 1]$ and is rejected because it is not feasible.

4. $K$ is set to $[2, 1, 4]$ and is determined to be feasible.

   $J$ is set to $[2, 1, 4]$ because $K$ is feasible.

5. $K$ is set to $[2, 5, 1, 4]$ and is rejected because it is not feasible.

6. $K$ is set to $[2, 1, 6, 4]$ and is rejected because it is not feasible.

7. $K$ is set to $[2, 7, 1, 4]$ and is rejected because it is not feasible.

The final value of $J$ is $[2, 1, 4]$. 
Worst-case time complexity

- Basic operation: comparison instructions
- Input size: $n$, the number of jobs
- Time complexity:
  - time for sorting: $\Theta(n \lg n)$
  - comparisons in for-i loop:
    $$\sum_{i=2}^{n} [(i - 1) + i] = n^2 - 1 \in \Theta(n^2)$$
  - Overall: $W(n) \in \Theta(n^2)$
Theorem 4.4

- Algorithm 4.4 always produces an optimal set of jobs
- Proof: induction on the number of jobs $n$
Huffman code

- Data compression
- binary code
- codeword
An example

- File: ababcbbbc
- Encoding scheme:
  - a: 00, b: 01, c: 11
  - a: 10, b: 0, c: 11
Prefix codes

Figure 4.9  • Binary tree corresponding to Code 4.2.
An example

<table>
<thead>
<tr>
<th>Character</th>
<th>Frequency</th>
<th>C1 (Fixed-Length)</th>
<th>C2</th>
<th>C3 (Huffman)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>16</td>
<td>000</td>
<td>10</td>
<td>00</td>
</tr>
<tr>
<td>b</td>
<td>5</td>
<td>001</td>
<td>11110</td>
<td>1110</td>
</tr>
<tr>
<td>c</td>
<td>12</td>
<td>010</td>
<td>1110</td>
<td>110</td>
</tr>
<tr>
<td>d</td>
<td>17</td>
<td>011</td>
<td>110</td>
<td>01</td>
</tr>
<tr>
<td>e</td>
<td>10</td>
<td>100</td>
<td>1111</td>
<td>111</td>
</tr>
<tr>
<td>f</td>
<td>25</td>
<td>101</td>
<td>0</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 4.1  Three codes for the same file. C3 is optimal.

Figure 4.10  The binary character code for Code C2 in Example 4.7 appears in (a), while the one for Code C3 (Huffman) appears in (b).
The number of bits taken to encode a file

\[ \text{bits}(T) = \sum_{i=1}^{n} \text{frequency}(v_i) \text{depth}(v_i) \]

- Compute Bits(C1), Bits(C2), and Bits(C3)
Huffman’s algorithm

- Data structure

```c
struct nodetype {
    char symbol; // The value of a character.
    int frequency; // The number of times the character is in the file.
    nodetype* left;
    nodetype* right;
};
```
Priority queue $PQ$

- Arrange $n$ pointers to nodetype records in $PQ$ so that for each pointer $p$ in $PQ$
  - $p$ -> symbol = a distinct character in the file
  - $p$ -> frequency = the frequency of that character in the file
  - $p$ -> left = $p$ -> right = NULL
  - for ($i = 1; i <= n-1; i++$) { // There is no solution check; rather,
    - remove ($PQ$, $p$); // solution is obtained when $i = n - 1$.
    - remove ($PQ$, $q$); // Selection procedure.
    - $r$ = new nodetype; // There is no feasibility check.
    - $r$->left = $p$;
    - $r$->right = $q$;
    - $r$->frequency = $p$->frequency + $q$->frequency;
    - insert ($PQ$, $r$);
  }
  - remove ($PQ$, $r$);
  - return $r$;
An example of application (1)

b:5  e:10  c:12  a:16  d:17  f:25
An example of application (2)
An example of application (3)
An example of application (4)
An example of application (5)
Proof

- **Lemma 4.4**
  The binary tree corresponding to an optimal binary prefix code is full. That is, every nonleaf has two children.

- **Theorem 4.5**
  Huffman’s algorithm produces an optimal binary code.

![Diagram](image)

*Figure 4.12*  The branches rooted at $v$ and $w$ are swapped.
The greedy approach versus dynamic programming: The knapsack problem

- Efficiency:
  - Greedy approach is often simpler and more efficient

- Proof:
  - Dynamic programming: principle of optimality
  - Greedy approach: A proof is needed to show that a particular greedy algorithm always produces an optimal solution
the knapsack problem

- The knapsack problem
  - Let
    - $S = \{item_1, item_2, ..., item_n\}$
    - $w_i =$ weight of $item_i$
    - $p_i =$ profit of $item_i$
    - $W =$ maximum weight the knapsack can hold

- The fractional knapsack problem
- The 0-1 knapsack problem
A greedy approach to the fractional knapsack problem

- Item 1 2 3
- Profit 50$ 60$ 140$
- Weight 5 10 20
- Knapsack capacity = 30

- Order the items in non-increasing order according to profit per unit weight
- Total profit in the previous example
$50 + 140 + (5/10)(60) = 220$
A greedy approach to the 0-1 knapsack problem

- The 0-1 knapsack problem
  - Let
    - $S = \{item_1, item_2, \ldots, item_n\}$
    - $w_i = \text{weight of } item_i$
    - $p_i = \text{profit of } item_i$
    - $W = \text{maximum weight the knapsack can hold}$
  - Determine a subset $A$ such that
    $$\sum_{item_i \in A} p_i \text{ is maximized subject to } \sum_{item_i \in A} w_i \leq W$$
Greedy approach fails

- Approach 1: steal the items with the largest profit first
- Approach 2: Steal the lightest items first
- Approach 3: Steal the items with the largest profit per unit weight first
Greedy approach fails (cont’d)

Figure 4.13  A greedy solution and an optimal solution to the 0-1 Knapsack problem.
A dynamic programming approach to the 0-1 knapsack problem

- The algorithm

\[ P[i][w] = \begin{cases} \max(P[i-1][w], p_i + P[i-1][w-w_i]) & \text{if } w_i \leq w \\ P[i-1][w] & \text{if } w_i > w. \end{cases} \]

- \( P[i][w] \) be the optimal profit obtained when choosing items only from the first \( i \) items under the restriction that the total weight cannot exceed \( w \),
- The maximum profit = \( P[n][W] \)
- Using array \( P[0\text{-}n][0\text{-}W] \)
  - Set \( P[0][W] \) and \( P[i][0] \) to 0
- Time complexity:
  - The number of entries computed is \( nW \Theta(nW) \)
A refinement of the dynamic programming algorithm for the 0-1 knapsack problem

- Going back to determine what entries are needed. Because

\[
P[n][W] = \begin{cases} 
\text{maximum} (P[n-1][W], p_n + P[n-1][W - w_n]) & \text{if } w_n \leq W \\
P[n-1][W] & \text{if } w_n > W,
\end{cases}
\]

- Entries needed in the (n-1)st row are \( P[n-1][W] \) and \( P[n-1][W - w_n] \)

- In general, we can use the fact that \( P[i][w] \) is computed from \( P[i-1][w] \) and \( P[i-1][w - w_i] \)

- Until \( n = 1 \) or \( w \leq 0 \)
An example

- $W = 30$

Figure 4.13  A greedy solution and an optimal solution to the 0-1 Knapsack problem.
Time complexity

- We compute at most $2^i$ entries in the $(n-i)$th row.
- The total number of entries computed is at most
  \[1 + 2 + 2^2 + \ldots + 2^{n-1} = 2^n - 1 = \Theta(2^n)\]
Exercises

- 3,9
- 11
- 20
- 25,26
- 34
The End of Chapter 4