Greedy Algorithms

Textbook reading

Chapter 4
Chapter 5
Overview

**Design principle:**
- Make progress towards a solution based on local criteria

**Proof techniques:**
- Induction
- “Stay ahead” arguments
- Exchange arguments

**Problems:**
- Interval scheduling
- Minimum spanning trees
- Shortest paths
- Minimum-length codes

**Basic data structures:**
- Binary heaps
- Union-find structure
Problem 1: Interval Scheduling

**Given:** Set of activities competing for time intervals on a given resource

**Goal:** Schedule as many non-conflicting activities as possible
Problem 1: Interval Scheduling

Given: Set of activities competing for time intervals on a given resource

Goal: Schedule as many non-conflicting activities as possible
Problem 1: Interval Scheduling

**Given:** Set of activities competing for time intervals on a given resource

**Goal:** Schedule as many non-conflicting activities as possible
Problem 2: Minimum Spanning Tree (1)

Given \( n \) computers, we want to connect them so that every pair of them can communicate with each other.

- We don’t care whether the connection between every pair of computers is short.
- We don’t care about fault tolerance.
- Every foot of cable costs us $1.

We want the cheapest possible network.
Problem 2: Minimum Spanning Tree (2)

Given a graph $G = (V, E)$ and an assignment of weights (costs) to the edges of $G$, a **minimum spanning tree (MST)** $T$ of $G$ is a spanning tree with minimum total weight

$$w(T) = \sum_{e \in T} w(e).$$
Problem 3: Single-Source Shortest Paths

**Given:**
- A graph $G$,
- An assignment of weights to the edges of $G$, and
- A *source vertex* $s$.

**Goal:**
Compute the distance $\text{dist}(s, v)$ from $s$ to $v$, for all $v \in G$.

**Assumption:**
All edge weights are non-negative.
Problem 4: Minimum-Length Codes

The problem:

Given a text $T$ containing characters $c_1, c_2, \ldots, c_n$, each with frequency $f(c_i)$, find a binary code that encodes $T$ in the minimum number of bits.

Motivation:

- We want to transmit the text over a slow network connection.
- We want to store the text on disk, while minimizing the disk space usage.
- ...
The problem:

Given a text $\mathcal{T}$ containing characters $c_1, c_2, \ldots, c_n$, each with frequency $f(c_i)$, find a binary code that encodes $\mathcal{T}$ in the minimum number of bits.
In an optimization problem, there are many possible solutions, each having an associated value (cost, benefit, etc.). Our goal is to find a solution that minimizes or maximizes this value.

**Examples:**
- **Interval scheduling:** Maximize number of scheduled activities
- **Minimum spanning tree:** Find a spanning tree of minimum cost
- **Shortest paths:** Find a spanning tree that minimizes the cost of all root-vertex paths
- **Minimum-length codes:** Find a code that encodes the given text in the minimum number of bits
A Greedy Framework for Interval Scheduling

**Find-Schedule** ($S'$)

1. $S' \leftarrow \emptyset$
2. while $S$ is not empty
3. do pick an interval $I$ from $S$
4. add $I$ to $S'$
5. remove all intervals from $S$ that conflict with $I$
6. return $S'$
A Greedy Framework for Interval Scheduling

**Find-Schedule**($S$)
1. $S' \leftarrow \emptyset$
2. while $S$ is not empty
3. do pick an interval $I$ from $S$
4. add $I$ to $S'$
5. remove all intervals from $S$ that conflict with $I$
6. return $S'$

Main questions:


**A Greedy Framework for Interval Scheduling**

**Algorithm:**

```
FIND-SCHEDULE(S)
1  S' ← ∅
2  while S is not empty
3      do pick an interval I from S
4          add I to S'
5      remove all intervals from S that conflict with I
6  return S'
```

**Main questions:**

- Can we choose an arbitrary interval I in each iteration?
A Greedy Framework for Interval Scheduling

\[
\text{FIND-SCHEDULE}(S) \\
1 \quad S' \leftarrow \emptyset \\
2 \quad \textbf{while} \ S \text{ is not empty} \\
3 \quad \textbf{do} \ \text{pick an interval} \ I \ \text{from} \ S \\
4 \quad \text{add} \ I \ \text{to} \ S' \\
5 \quad \text{remove all intervals from} \ S \ \text{that conflict with} \ I \\
6 \quad \textbf{return} \ S'
\]

Main questions:
- Can we choose an arbitrary interval \( I \) in each iteration?
- How do we choose interval \( I \) in each iteration?
Greedy Strategies for Interval Scheduling
Choose the interval that begins first
Greedy Strategies for Interval Scheduling

- Choose the interval that begins first

![Diagram of intervals]

- 

CSci 3110  Greedy Algorithms  10/63
Greedy Strategies for Interval Scheduling

- Choose the interval that begins first

- Choose the shortest interval

![Diagram showing intervals]

CSci 3110 • Greedy Algorithms • 10/63
Greedy Strategies for Interval Scheduling

- Choose the interval that begins first

- Choose the shortest interval
Greedy Strategies for Interval Scheduling

- Choose the interval that begins first

- Choose the shortest interval

- Choose the interval that has the fewest conflicts
Greedy Strategies for Interval Scheduling

- Choose the interval that begins first

- Choose the shortest interval

- Choose the interval that has the fewest conflicts
The Strategy That Works

**FIND-SCHEDULE**($S$)

1. $S' \leftarrow \emptyset$
2. **while** $S$ is not empty
3. **do** let $I$ be the interval in $S$ that ends first
4. **add** $I$ to $S'$
5. **remove** all intervals from $S$ that conflict with $I$
6. **return** $S'$
The Strategy That Works

**Find-Schedule(S)**

1. $S' \leftarrow \emptyset$
2. while $S$ is not empty
3. do let $I$ be the interval in $S$ that ends first
4. add $I$ to $S'$
5. remove all intervals from $S$ that conflict with $I$
6. return $S'$
The Strategy That Works

**Find-Schedule** \((S)\)

1. \(S' \leftarrow \emptyset\)
2. while \(S\) is not empty
3. do let \(I\) be the interval in \(S\) that ends first
4. add \(I\) to \(S'\)
5. remove all intervals from \(S\) that conflict with \(I\)
6. return \(S'\)
Lemma: Procedure FIND-SCHEDULE finds a maximal-cardinality set of conflict-free intervals.
Lemma: Procedure \textsc{Find-Schedule} finds a maximal-cardinality set of conflict-free intervals.

Proof by induction:

Base case(s): Verify that the claim holds for a set of initial instances.

Inductive step: Prove that, if the claim holds for the first $k$ instances, it holds for the $(k + 1)$-st instance.
The Greedy Algorithm Stays Ahead

Lemma: Procedure `FIND-SCHEDULE` finds a maximal-cardinality set of conflict-free intervals.

Proof idea ("stay ahead" argument)

- Let $I_1 \prec I_2 \prec \cdots \prec I_k$ be the schedule we compute.
- Let $O_1 \prec O_2 \prec \cdots \prec O_m$ be an optimal schedule.

Prove by induction on $j$ that $I_j$ ends no later than $O_j$.

Base case: $I_1$ is the interval that ends first, so it ends no later than $O_1$.

Inductive step: $I_1 \ldots I_k$ by our inductive hypothesis these end no later than $O_1 \ldots O_k$. $I_{k+1}$ ends no later than $I_{k+1}$. 
base case

inductive hypothesis

inductive step
The Greedy Algorithm Stays Ahead

Lemma: Procedure FIND-SCHEDULE finds a maximal-cardinality set of conflict-free intervals.

Proof idea (“stay ahead” argument)

- Let $I_1 \prec I_2 \prec \cdots \prec I_k$ be the schedule we compute
- Let $O_1 \prec O_2 \prec \cdots \prec O_m$ be an optimal schedule

Prove by induction on $j$ that $I_j$ ends no later than $O_j$.

$\therefore$ If $k < m$, the algorithm could have scheduled $O_{k+1}$ after $I_k$. 
Implementing the Algorithm

\textbf{Find-Schedule}(S)

1. \( S' \leftarrow \emptyset \quad \triangleright \text{Represent } S' \text{ as a linked list } O(n) \)
2. sort the intervals by increasing finish time \( O(n \log n) \)
3. \( \triangleright \text{Denote the intervals as } I_1, I_2, \ldots, I_n, \text{ sorted by finish time} \)
4. \( \triangleright \text{The start and finish times of } I_j \text{ are } s_j \text{ and } f_j \)
5. \( f \leftarrow -\infty \quad O(1) \)
6. \text{for } j \leftarrow 1 \text{ to } n \quad O(n) \\
7. \text{do if } s_j \geq f \quad O(1) \\
8. \text{then append } I_j \text{ to } S' \quad O(n) \\
9. \quad f \leftarrow f_j \quad O(1) \\
10. \text{return } S' \quad O(1)
Implementing the Algorithm

**FIND-SCHEDULE**($S$)

1. $S' \leftarrow \emptyset$  \hspace{1em} $\triangleright$ Represent $S'$ as a linked list
2. sort the intervals by increasing finish time
3. $\triangleright$ Denote the intervals as $I_1, I_2, \ldots, I_n$, sorted by finish time
4. $\triangleright$ The start and finish times of $I_j$ are $s_j$ and $f_j$
5. $f \leftarrow -\infty$
6. for $j \leftarrow 1$ to $n$
7. \hspace{1em} do if $s_j \geq f$
8. \hspace{2em} then append $I_j$ to $S'$
9. \hspace{1.5em} $f \leftarrow f_j$
10. return $S'$

**Lemma:** A maximum-cardinality set of conflict-free intervals can be found in $O(n \log n)$ time.
Building a Minimum Spanning Tree Greedily
**Greedy choice:** Pick the shortest edge
**Greedy choice:** Pick the shortest edge that connects two previously disconnected vertices.
Greedy choice: Pick the shortest edge that connects two previously disconnected vertices.

**Kruskal**($G$)

1. $T \leftarrow \emptyset$
2. while there are two vertices that are disconnected
3. do let $e$ be the cheapest edge connecting two previously disconnected vertices
4. $T \leftarrow T \cup \{e\}$
A Cut Theorem

Assumption: No two edges have the same weight.

Theorem: Let $S$ be a proper subset of $V$, and let $e$ be the cheapest edge with exactly one endpoint in $S$. Then every minimum spanning tree of $G$ contains edge $e$.

Proof: An exchange argument
**A Cut Theorem**

**Assumption:** No two edges have the same weight.

**Theorem:** Let $S$ be a proper subset of $V$, and let $e$ be the cheapest edge with exactly one endpoint in $S$. Then every minimum spanning tree of $G$ contains edge $e$.

**Proof:** An exchange argument
Lemma: Kruskal’s algorithm computes a minimum spanning tree.
Lemma: Kruskal’s algorithm computes a minimum spanning tree.
base case: everything disconnected
pick the minimum weight edge \((u,v)\)
cut theorem: \(S = \{u,v\}\)
\(V/S \subseteq \{u,v\}\)
the edge is in any MST of \(G\)

inductive step: assume we have picked \(k\) edges from an MST
pick \(k+1\) to be the minimum weight edge
connecting two components \(S, A\)
cut theorem: \(S\) \(\cap A\) \(\neq \emptyset\), the edge is in any MST
Enforcing Distinct Edge Weights

Redefinition of edge weights:

- Let the edges be $e_1, e_2, \ldots, e_m$.
- Define $w'(e_i) = (w(e_i), i)$.
- Define addition of pairs as $(a, b) + (c, d) = (a + c, b + d)$.
- Define comparison as $(a, b) < (c, d)$ if $a < c$ or $a = c$ and $b < d$.

Lemma: A minimum spanning tree w.r.t. weight function $w'$ is also a minimum spanning tree w.r.t. weight function $w$.

Proof:

If $w(T_1) < w(T_2)$, then $w'(T_1) < w'(T_2)$.
Implementing Kruskal’s Algorithm

**Kruskal(G)**

1. $T \leftarrow \emptyset$
2. while there are two vertices that are disconnected
3. do let $e$ be the cheapest edge connecting two previously disconnected vertices
4. $T \leftarrow T \cup \{e\}$

**Kruskal(G)**

1. $T \leftarrow (V, \emptyset)$
2. sort the edges in $G$ by increasing weight
3. for every edge $(v, w)$ of $G$, in sorted order
4. do if $v$ and $w$ belong to different connected components of $T$
5. then add edge $(v, w)$ to $T$
Union-Find Data Structure

Given a set $S$ of $n$ elements, maintain a partition of $S$ into subsets $S_1, S_2, \ldots, S_k$.

Support the following operations:

- **Union**($x, y$): Replace sets $S_i$ and $S_j$ such that $x \in S_i$ and $y \in S_j$ with $S_i \cup S_j$ in the current partition.

- **Find**($x$): Returns a **representative** $r(S_j) \in S_j$ of the set $S_j$ that contains $x$.

In particular, Find($x$) = Find($y$) if and only if $x$ and $y$ belong to the same set.
Union-Find Data Structure

Given a set $S$ of $n$ elements, maintain a partition of $S$ into subsets $S_1, S_2, \ldots, S_k$.

Support the following operations:

- **Union($x, y$):** Replace sets $S_i$ and $S_j$ such that $x \in S_i$ and $y \in S_j$ with $S_i \cup S_j$ in the current partition.

- **Find($x$):** Returns a representative $r(S_j) \in S_j$ of the set $S_j$ that contains $x$.

In particular, $\text{Find}(x) = \text{Find}(y)$ if and only if $x$ and $y$ belong to the same set.
Kruskal’s Algorithm Using Union-Find

Kruskal(\(G\))
1. \(T \leftarrow (V, \emptyset)\)
2. sort the edges in \(G\) by increasing weight
3. for every edge \((v, w)\) of \(G\), in sorted order
4.    do if \(\text{Find}(v) \neq \text{Find}(w)\)
5.      then add edge \((v, w)\) to \(T\)
6.      Union\((v, w)\)
Kruskal’s Algorithm Using Union-Find

**Kruskal**$(G)$

1. $T \leftarrow (V, \emptyset)$
2. sort the edges in $G$ by increasing weight
3. for every edge $(v, w)$ of $G$, in sorted order
4. do if $\text{Find}(v) \neq \text{Find}(w)$
5. then add edge $(v, w)$ to $T$
6. Union$(v, w)$

**Lemma:** Kruskal’s algorithm takes $O(m \log m)$ time, plus the cost of $2m$ Find and $n - 1$ Union operations.
A Simple Union-Find Structure

List node:
- Pointers to predecessor and successor
- Pointer to list head
- Pointer to list tail (only valid for head node)
- List size (only valid for head node)
<table>
<thead>
<tr>
<th></th>
<th>6</th>
<th></th>
<th></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>6</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ n \text{ vertices} \]

\[
\text{smart}^+ \quad \begin{array}{cccccc}
1 & 1 & 1 & 1 & 1 & 1 \\
0(n) & 6(n^2) & & & & \\
\end{array}
\]
**Operations**

**Find** ($x$)
1. **return** $\text{key(head}(x))$

**Union** ($x$, $y$)
1. **if** $\text{listSize(head}(x)) < \text{listSize(head}(y))$
2. **then** swap $x \leftrightarrow y$
3. $\text{pred(head}(y)) \leftarrow \text{tail(head}(x))$
4. $\text{succ(tail(head}(x))} \leftarrow \text{head}(y)$
5. $\text{listSize(head}(x)) \leftarrow \text{listSize(head}(x)) + \text{listSize(head}(y))$
6. $\text{tail(head}(x)) \leftarrow \text{tail(head}(y))$
7. $z \leftarrow \text{head}(y)$
8. **while** $z \neq \text{nil}$
9. **do** $\text{head}(z) \leftarrow \text{head}(x)$
10. $z \leftarrow \text{succ}(z)$

*Note: $y$ is always smaller.*
Lemma: A Find operation takes constant time.
Lemma: A Find operation takes constant time.

Total cost of all Union operations:

\[
C(n) = O \left( \sum_{x \in S} c(x) \right),
\]

where \( c(x) = \) total number of times element \( x \) is touched (\( x \)'s head pointer changes).
**Lemma:** A Find operation takes constant time.

**Total cost of all Union operations:**

\[ C(n) = \mathcal{O} \left( \sum_{x \in S} c(x) \right), \]

where \( c(x) = \) total number of times element \( x \) is touched (\( x \)'s head pointer changes).

Let \( s(x, i) = \) size of the list containing \( x \) after \( x \) is touched \( i \) times.

**Lemma:** \( s(x, i) \geq 2^i \), for all \( x \in S \) and \( i \geq 0 \).
Corollary: \( c(x) \leq \lg n \).
Corollary: \( c(x) \leq \lg n \).

Corollary: \( C(n) = O(n \lg n) \).
Corollary: \( c(x) \leq \lg n \).

Corollary: \( C(n) = \mathcal{O}(n \lg n) \).

Corollary: Kruskal's algorithm takes \( \mathcal{O}((n + m) \lg (n + m)) \) time.
**Greedy choice:** From among all edges with exactly one endpoint in $S$, pick the cheapest.
The Abstract Data Type Priority Queue

Operations:

**INSERT**\( (Q, x, p) \): Insert element \( x \) into priority queue \( Q \) and give it priority \( p \)

**DELETE**\( (Q, x) \): Delete element \( x \) from priority queue \( Q \)

**FIND-MIN**\( (Q) \): Find and return the element with minimum priority in \( Q \)

**DELETE-MIN**\( (Q) \): Delete the element with minimum priority from \( Q \) and return it

**DECREASE-KEY**\( (Q, x, p) \): Decrease the priority of element \( x \) to \( p \)
The Binary Heap: A Priority Queue

Perfect binary tree:

- All levels full, except possibly last one.
- All nodes on last level are as far left as possible.
The Binary Heap: A Priority Queue

Perfect binary tree:
- All levels full, except possibly last one.
- All nodes on last level are as far left as possible.

Binary min-heap property:
For the keys $k_v$ and $k_u$ of any node $v$ and its parent $u$, $k_u \leq k_v$. 
The Binary Heap: A Priority Queue

Perfect binary tree:
- All levels full, except possibly last one.
- All nodes on last level are as far left as possible.

Binary min-heap property:
For the keys $k_v$ and $k_u$ of any node $v$ and its parent $u$, $k_u \leq k_v$.

(For the keys $k_v$ and $k_u$ of any node $v$ and an ancestor $u$ of $v$, $k_u \leq k_v$.)
Implicit Representation of Perfect Binary Trees

```
1 2 3
4 5 7 6
8 9 10 11 12
```
Implicit Representation of Perfect Binary Trees

\[
\text{left}(i) = 2i \\
\text{right}(i) = 2i + 1 \\
\text{parent}(i) = \lfloor i/2 \rfloor
\]
Heap Operations
Heap Operations

**FIND-MIN**(Q)
1. `return Q[1]`
**Heap Operations**

**FIND-MIN**($Q$)
1. return $Q[1]$

**INSERT**($Q$, $x$, $p$)
1. $size(Q) \leftarrow size(Q) + 1$
2. $Q[size(Q)] \leftarrow (x, p)$
3. Heapify-Up($Q$, size($Q$))
Heap Operations

**FIND-MIN(Q)**
1. return \( Q[1] \)

**INSERT(Q, x, p)**
1. \( \text{size}(Q) \leftarrow \text{size}(Q) + 1 \)
2. \( Q[\text{size}(Q)] \leftarrow (x, p) \)
3. Heapify-Up(\( Q, \text{size}(Q) \))

**DELETE(Q, i)**
1. \( Q[i] \leftarrow Q[\text{size}(Q)] \)
2. \( \text{size}(Q) \leftarrow \text{size}(Q) - 1 \)
3. if \( Q[i] < Q[\text{parent}(i)] \)
   4. then Heapify-Up(\( Q, i \))
   5. else Heapify-Down(\( Q, i \))
delete can't replace with the left child

problem
no binary tree property
**Restoring the Heap Property (1)**

**HEAPIFY-UP**($Q$, $i$)

1. **while** $i \neq 1$ and $Q[i] < Q[\text{parent}(i)]$
2. **do** swap $Q[i] \leftrightarrow Q[\text{parent}(i)]$
3. $i \leftarrow \text{parent}(i)$
**Restoring the Heap Property (1)**

**Procedure: Heapify-Up**

1. **while** $i \neq 1$ and $Q[i] < Q[\text{parent}(i)]$
2. **do** swap $Q[i] \leftrightarrow Q[\text{parent}(i)]$
3. $i \leftarrow \text{parent}(i)$

**Lemma:** If the only violation of the heap property is between a node $Q[i]$ and its ancestors, the procedure Heapify-Up$(Q, i)$ restores the heap property of $Q$. 
Restoring the Heap Property (2)

**Heapify-Down** \((Q, i)\)

1. smallest \(\leftarrow i\)
2. \(i \leftarrow 0\)
3. **while** \(i \neq\) smallest
4. **do** \(i \leftarrow\) smallest
5. **if** \(\text{left}(i) \leq \text{heap-size}(Q)\) and \(Q[\text{left}(i)] < Q[\text{smallest}]\)
6. **then** smallest \(\leftarrow\) \(\text{left}(i)\)
7. **if** \(\text{right}(i) \leq \text{heap-size}(Q)\) and \(Q[\text{right}(i)] < Q[\text{smallest}]\)
8. **then** smallest \(\leftarrow\) \(\text{right}(i)\)
9. swap \(Q[i] \leftrightarrow Q[\text{smallest}]\)
Restoring the Heap Property (2)

**HEAPIFY-DOWN** *(Q, i)*

1. smallest ← i
2. i ← 0
3. while *i* ≠ smallest
4. do i ← smallest
5. if left(*i*) ≤ heap-size(*Q*) and
   *Q*[left(*i*)] < *Q*[smallest]
6. then smallest ← left(*i*)
7. if right(*i*) ≤ heap-size(*Q*) and
   *Q*[right(*i*)] < *Q*[smallest]
8. then smallest ← right(*i*)
9. swap *Q*[i] ↔ *Q*[smallest]

**Lemma:** If the only violation of the heap property is between a node *Q*[i] and its descendants, the procedure Heapify-Down(*Q*, i) restores the heap property of *Q*. 
Complexity of Heap Operations

Find  $O(1)$

insert  $O(\log n)$
Observation: Operations Find-Min, Insert, Delete, Delete-Min, and Decrease-Key take $O(1)$ time, excluding the time spent in Heapify-Up and Heapify-Down operations.
Observation: Operations Find-Min, Insert, Delete, Delete-Min, and Decrease-Key take $O(1)$ time, excluding the time spent in Heapify-Up and Heapify-Down operations.

Observation: Operations Find-Min, Insert, Delete, Delete-Min, and Decrease-Key perform one Heapify-Up or Heapify-Down operation each.
Observation: Operations Find-Min, Insert, Delete, Delete-Min, and Decrease-Key take $\mathcal{O}(1)$ time, excluding the time spent in Heapify-Up and Heapify-Down operations.

Observation: Operations Find-Min, Insert, Delete, Delete-Min, and Decrease-Key perform one Heapify-Up or Heapify-Down operation each.

Observation: Operations Heapify-Up and Heapify-Down take time proportional to the height of the heap, that is, $\mathcal{O}(\lg n)$.
**Observation:** Operations Find-Min, Insert, Delete, Delete-Min, and Decrease-Key take $O(1)$ time, excluding the time spent in Heapify-Up and Heapify-Down operations.

**Observation:** Operations Find-Min, Insert, Delete, Delete-Min, and Decrease-Key perform one Heapify-Up or Heapify-Down operation each.

**Observation:** Operations Heapify-Up and Heapify-Down take time proportional to the height of the heap, that is, $O(\log n)$.

**Corollary:** Operations Insert, Delete, Delete-Min, and Decrease-Key take $O(\log n)$ time. Operation Find-Min takes $O(1)$ time.
Decrease key

change weight

x \rightarrow y
Prim’s Algorithm

**Prim(G)**
1. Mark every vertex of G as unexplored
2. Mark every edge of G as non-tree edge
3. \( s \leftarrow \) some vertex of G
4. \( Q \leftarrow V(G) \) (priority queue)
5. Give every vertex priority \( \infty \), except \( s \):
   \( p(s) = 0 \)
6. \( e(s) \leftarrow \) nil
7. while \( Q \) is not empty
8. do \( v \leftarrow \) Delete-Min(\( Q \))
9. Mark vertex \( v \) as visited
10. Mark edge \( e(v) \) as tree edge
11. for every edge \( (v, w) \in \text{Adj}(v) \)
12. do if vertex \( w \) is unexplored and
   \( p(w) > w(v, w) \)
   then Decrease-Key(\( Q, w, w(v, w) \))
13. \( e(w) \leftarrow (v, w) \)
Prim’s Algorithm

**Prim**($G$)

1. Mark every vertex of $G$ as unexplored \( O(n) \)
2. Mark every edge of $G$ as non-tree edge \( O(m) \)
3. \( s \leftarrow \) some vertex of $G$ \( O(1) \)
4. \( Q \leftarrow V(G) \) \( O(n) \)
5. Give every vertex priority \( \infty \), except \( s \): \( p(s) = 0 \) \( O(1) \)
6. \( e(s) \leftarrow \text{nil} \) \( O(1) \)
7. \textbf{while} $Q$ is not empty \( - O(n) \) steps
8. \quad \textbf{do} \quad \textbf{v} \leftarrow \text{Delete-Min}(Q) \( O(\log n) \)
9. \quad \textbf{Mark vertex} \( v \) as visited \( O(1) \)
10. \quad \textbf{Mark edge} \( e(v) \) as tree edge \( O(1) \)
11. \quad \textbf{for} \quad \textbf{every edge} \( (v, w) \in \text{Adj}(v) \)
12. \quad \quad \textbf{do} \quad \textbf{if} \quad \textbf{vertex} \( w \) is unexplored and \( p(w) > w(v, w) \) \( O(1) \)
13. \quad \quad \quad \textbf{then} \quad \textbf{Decrease-Key}(Q, w, w(v, w)) \( O(\log n) \)
14. \quad \quad \quad \quad \textbf{e}(w) \leftarrow (v, w) \( O(1) \)

**Running time:**

- **Time complexity:** \( O(E + V \log V) \)
**Prim’s Algorithm**

**Prim(G)**

1. Mark every vertex of $G$ as unexplored
2. Mark every edge of $G$ as non-tree edge
3. $s \leftarrow$ some vertex of $G$
4. $Q \leftarrow V(G)$
5. Give every vertex priority $\infty$, except $s$:
   \[ p(s) = 0 \]
6. $e(s) \leftarrow$ nil
7. while $Q$ is not empty
8.   do $v \leftarrow$ Delete-Min($Q$)
9.   Mark vertex $v$ as visited
10. Mark edge $e(v)$ as tree edge
11. for every edge $(v, w) \in \text{Adj}(v)$
12.   do if vertex $w$ is unexplored and $p(w) > w(v, w)$
13.      then Decrease-Key($Q$, $w$, $w(v, w)$)
14.      \[ e(w) \leftarrow (v, w) \]

**Running time:**

- $n$ Insert operation
Prim’s Algorithm

\textbf{PRIM}(G)

1. Mark every vertex of $G$ as unexplored
2. Mark every edge of $G$ as non-tree edge
3. $s \leftarrow$ some vertex of $G$
4. $Q \leftarrow V(G)$
5. Give every vertex priority $\infty$, except $s$:
   \[ p(s) = 0 \]
6. $e(s) \leftarrow \text{nil}$
7. while $Q$ is not empty
   8. do $v \leftarrow \text{Delete-Min}(Q)$
      9. Mark vertex $v$ as visited
      10. Mark edge $e(v)$ as tree edge
      11. for every edge $(v, w) \in \text{Adj}(v)$
         12. do if vertex $w$ is unexplored and $p(w) > w(v, w)$
            then \text{Decrease-Key}($Q$, $w$, $w(v, w)$)
            \[ e(w) \leftarrow (v, w) \]

\textbf{Running time:}

- $n$ Insert operation
- $n$ Delete-Min operations
**Prim's Algorithm**

**Prim(G)**
1. Mark every vertex of \( G \) as unexplored
2. Mark every edge of \( G \) as non-tree edge
3. \( s \leftarrow \) some vertex of \( G \)
4. \( Q \leftarrow V(G) \)
5. Give every vertex priority \( \infty \), except \( s \):
   \[ p(s) = 0 \]
6. \( e(s) \leftarrow \) nil
7. while \( Q \) is not empty
   8. do \( v \leftarrow \) Delete-Min(\( Q \))
   9. Mark vertex \( v \) as visited
   10. Mark edge \( e(v) \) as tree edge
   11. for every edge \( (v, w) \in \text{Adj}(v) \)
   12. do if vertex \( w \) is unexplored and \( p(w) > w(v, w) \)
   13. then Decrease-Key(\( Q, w, w(v, w) \))
   14. \( e(w) \leftarrow (v, w) \)

**Running time:**
- \( n \) Insert operation
- \( n \) Delete-Min operations
- \( \mathcal{O}(m) \) Decrease-Key operations
Prim’s Algorithm

$\text{PRIM}(G)$
1. Mark every vertex of $G$ as unexplored
2. Mark every edge of $G$ as non-tree edge
3. $s \leftarrow$ some vertex of $G$
4. $Q \leftarrow V(G)$
5. Give every vertex priority $\infty$, except $s$:
   \[ p(s) = 0 \]
6. $e(s) \leftarrow \text{nil}$
7. While $Q$ is not empty
6. do $v \leftarrow \text{Delete-Min}(Q)$
9. Mark vertex $v$ as visited
10. Mark edge $e(v)$ as tree edge
11. For every edge $(v, w) \in \text{Adj}(v)$
12. do if vertex $w$ is unexplored and $p(w) > w(v, w)$
13. \hspace{1em} then $\text{Decrease-Key}(Q, w, w(v, w))$
14. \hspace{1em} $e(w) \leftarrow (v, w)$

Running time:
- $n$ Insert operation
- $n$ Delete-Min operations
- $O(m)$ Decrease-Key operations
- Each costs $O(lg n)$ time
Prim’s Algorithm

\text{\textbf{Prim}(G)}

1. Mark every vertex of \( G \) as unexplored
2. Mark every edge of \( G \) as non-tree edge
3. \( s \leftarrow \) some vertex of \( G \)
4. \( Q \leftarrow V(G) \)
5. Give every vertex priority \( \infty \), except \( s \):
   \( p(s) = 0 \)
6. \( e(s) \leftarrow \text{nil} \)
7. \textbf{while} \( Q \) is not empty
8. \textbf{do} \( v \leftarrow \text{Delete-Min}(Q) \)
9. \quad Mark vertex \( v \) as visited
10. \quad Mark edge \( e(v) \) as tree edge
11. \quad \textbf{for} every edge \((v, w) \in \text{Adj}(v)\)
12. \quad \textbf{do if} vertex \( w \) is unexplored and \( p(w) > w(v, w) \)
13. \quad \textbf{then} \text{Decrease-Key}(Q, w, w(v, w))
14. \quad \quad \quad e(w) \leftarrow (v, w)

\textbf{Running time:}

\begin{itemize}
  \item \( n \) Insert operation
  \item \( n \) Delete-Min operations
  \item \( \mathcal{O}(m) \) Decrease-Key operations
  \item Each costs \( \mathcal{O}(\lg n) \) time
\end{itemize}

\textbf{Total:} \( \mathcal{O}(m \lg n) \)

\[ \lfloor n^{\frac{3}{2}} \rfloor = 2 \log_{5} n \]
**Prim’s Algorithm**

**Prim(G)**

1. Mark every vertex of \( G \) as unexplored
2. Mark every edge of \( G \) as non-tree edge
3. \( s \) ← some vertex of \( G \)
4. \( Q \) ← \( V(G) \)
5. Give every vertex priority \( \infty \), except \( s \):
   \[ p(s) = 0 \]
6. \( e(s) \) ← nil
7. **while** \( Q \) is not empty
8. **do** \( v \) ← Delete-Min(\( Q \))
9. Mark vertex \( v \) as visited
10. Mark edge \( e(v) \) as tree edge
11. **for** every edge \( (v, w) \) ∈ \( \text{Adj}(v) \)
12. **do if** vertex \( w \) is unexplored and
   \[ p(w) > w(v, w) \]
13. **then** Decrease-Key(\( Q, w, w(v, w) \))
14. \( e(w) \) ← \( (v, w) \)

**Running time:**

- \( n \) Insert operation
- \( n \) Delete-Min operations
- \( \mathcal{O}(m) \) Decrease-Key operations
- Each costs \( \mathcal{O}(\lg n) \) time

**Total:** \( \mathcal{O}(m \lg n) \)

**Fibonacci heaps:**

- Delete-Min: \( \mathcal{O}(\lg n) \)
- Insert: \( \mathcal{O}(1) \)
- Decrease-Key: \( \mathcal{O}(1) \)

**Total:** \( \mathcal{O}(n \lg n + m) \)
A Greedy Choice for Shortest Paths

BFS does this, if there are no weights
A Greedy Choice for Shortest Paths

Greedy choice: Pick the vertex $v$ with the shortest path that contains only explored vertices besides $v$. 
**Dijkstra’s Algorithm**

**Dijkstra**\( (G, s) \)

1. Mark every vertex of \( G \) as unexplored
2. Mark every edge of \( G \) as non-tree edge
3. \( Q \leftarrow V(G) \)
4. Give every vertex priority \( \infty \), except \( s \): \( p(s) = 0 \)
5. \( e(s) \leftarrow \) nil
6. **while** \( Q \) is not empty
7. **do** \( (v, p) \leftarrow \) Delete-Min\( (Q) \)
8. Mark vertex \( v \) as visited
9. \( d(v) \leftarrow p \)
10. Mark edge \( e(v) \) as tree edge
11. **for** every edge \( (v, w) \in \text{Adj}(v) \)
12. **do if** vertex \( w \) is unexplored and \( p(w) > d(v) + w(v, w) \)
13. **then** Decrease-Key\( (Q, w, d(v) + w(v, w)) \)
14. \( e(w) \leftarrow (v, w) \)
Note: we don't directly store the paths, just $d(v) =$ the distance to get to $v$ and $e(v) =$ the edge we took to get to $v$.

- by backtracking following $e$ pointers we can recover the path.
**Dijkstra’s Algorithm**

**DIJKSTRA**\((G, s)\)

1. Mark every vertex of \(G\) as unexplored
2. Mark every edge of \(G\) as non-tree edge
3. \(Q \leftarrow V(G)\)
4. Give every vertex priority \(\infty\), except \(s\): \(p(s) = 0\)
5. \(e(s) \leftarrow \text{nil}\)
6. **while** \(Q\) is not empty
7. **do** \((v, p) \leftarrow \text{Delete-Min}(Q)\)
8. Mark vertex \(v\) as visited
9. \(d(v) \leftarrow p\)
10. Mark edge \(e(v)\) as tree edge
11. **for** every edge \((v, w) \in \text{Adj}(v)\)
12. **do if** vertex \(w\) is unexplored and \(p(w) > d(v) + w(v, w)\)
13. **then** Decrease-Key\((Q, w, d(v) + w(v, w))\)
14. \(e(w) \leftarrow (v, w)\)

**Lemma:** The running time of Dijkstra’s algorithm, when implemented using Fibonacci heaps, is \(O(n \lg n + m)\). Using binary heaps, it takes \(O((n + m) \lg n)\).
Lemma: If all edge weights are non-negative, Dijkstra's algorithm correctly computes the distances of all vertices from \( s \).

Proof:

If \( \ell(P_1) > \ell(P_2) \), then \( p_v > p_w + \text{dist}(w, v) \).

\[ \therefore p_v > p_w, \text{ a contradiction.} \]
(Non-)Decodable Codes

Which kinds of codes can be decoded?

```
hello
010011

h = 01
0 = 00
1 = 10
11

x = 010
```
Which kinds of codes can be decoded?

Consider the code:

- $a = 01$
- $m = 10$
- $n = 111$
- $o = 0$
- $r = 11$
- $s = 1$
- $t = 0011$
- $\square = 0111$
Which kinds of codes can be decoded?

■ Consider the code:
  
  \[
  \begin{align*}
  a &= 01 & m &= 10 & n &= 111 & o &= 0 \\
  r &= 11 & s &= 1 & t &= 0011 & u &= 0111
  \end{align*}
  \]

■ Now you send a fan-letter to your favourite movie star. One of the sentences is
  
  "You are a star."

■ You encode "star" as \( \langle 1|0011|01|11 \rangle \).
(Non-)Decodable Codes

Which kinds of codes can be decoded?

Consider the code:

- \(a = 01\)
- \(m = 10\)
- \(n = 111\)
- \(o = 0\)
- \(r = 11\)
- \(s = 1\)
- \(t = 0011\)
- \(\square = 0111\)

Now you send a fan-letter to your favourite movie star. One of the sentences is

“You are a star.”

You encode “star” as \(\langle 1|0011|01|11\rangle\).

Your idol receives the letter and decodes the text using your coding table:

\[
\langle 100110111 \rangle = \langle 10|0|11|0|111 \rangle = “moron”
\]

Oops, you have just insulted your idol.
Which kinds of codes can be decoded?

Consider the code:

\[
\begin{align*}
a &= 01 & m &= 10 & n &= 111 & o &= 0 \\
r &= 11 & s &= 1 & t &= 0011 & \Box &= 0111
\end{align*}
\]

Now you send a fan-letter to your favorite movie star. One of the sentences is

"You are a star."

You encode "star" as \langle 1|0011|01|11 \rangle.

Your idol receives the letter and decodes the text using your coding table:

\[
\langle 100110111 \rangle = \langle 10|0|11|0|111 \rangle = "\text{moron}" 
\]

Oops, you have just insulted your idol.

Using ASCII code, this would not have happened. Why?
Prefix Codes

A prefix code $C$ has the property that there are no two characters $x_1 
eq x_2$ such that $C(x_1)$ is a prefix of $C(x_2)$.

Examples:

- Fixed-length codes:
  - ASCII
  - 16-bit integers
- Huffman codes
  \[
  \begin{align*}
  c &= 010 & e &= 000 & i &= 0010 & k &= 00110 \\
  m &= 00111 & n &= 01100 & o &= 0111 & p &= 01101 \\
  r &= 110 & s &= 111 & t &= 1000 & u &= 1001 \\
  \square &= 101
  \end{align*}
  \]
- Number encoding: $C(i) = 0^i 1$

\[
\begin{align*}
\text{\# bits} &\quad 0 \quad 1 \\
\end{align*}
\]
Prefix Codes Can Be Decoded

It suffices to show that the first character can be decoded unambiguously. (Subsequent characters are decoded iteratively.)

Assume that there are two characters $c$ and $c'$ that could be the first character of the text, and assume that $|C(c)| \leq |C(c')|$. Then $C(c)$ is a prefix of $C(c')$, a contradiction.
Prefix Codes Can Be Decoded

It suffices to show that the first character can be decoded unambiguously. (Subsequent characters are decoded iteratively.)

Assume that there are two characters $c$ and $c'$ that could be the first character of the text, and assume that $|C(c)| \leq |C(c')|$. 

Then $C(c)$ is a prefix of $C(c')$, a contradiction.

$\because$ We want a minimum-length prefix code.
The Cost of Encoding Text

Let $\text{Cost}(\mathcal{T}, C)$ be the cost of encoding a text $\mathcal{T}$ using code $C$

Let $|C(x)|$ denote the number of bits used to encode character $x$

Let $f(x)$ be the frequency of character $x$ (number of times $x$ occurs in $\mathcal{T}$)

Then

$$\text{Cost}(\mathcal{T}, C) = \sum_x f(x) \cdot |C(x)|.$$
Every prefix code can be represented as a binary tree as follows:

- Edges are labelled
  - parent–left child = 0
  - parent–right child = 1
- Leaves correspond to characters
- Code of a character = labelling of edges from root to corresponding leaf

**Lemma:** Every internal node in a binary tree corresponding to an optimal prefix code has two children.
Huffman’s Algorithm
Huffman’s Algorithm
The Final Code

![Binary Tree Representation]

- **□ = 000**  
- **i = 0010**  
- **k = 0011**  
- **m = 0100**  
- **n = 0101**  
- **p = 0110**  
- **t = 0111**  
- **e = 100**  
- **u = 1010**  
- **o = 1011**  
- **c = 110**  
- **r = 1110**  
- **s = 1111**
Implementing Huffman’s Algorithm

**Huffman(T)**

1. Compute the set \( C \) of characters in \( T \) and determine their frequencies \( f(c) \).

2. Create one node \( v_c \) for each character in \( c \) and insert it into priority queue \( Q \).

3. **while** \( |Q| > 1 \) **do**

4. \( v \leftarrow \text{DELETEMIN}(Q) \) \( O(\log n) \)

5. \( w \leftarrow \text{DELETEMIN}(Q) \) \( O(\log n) \)

6. Create a new node \( u \) \( O(1) \)

7. \( f(u) \leftarrow f(v) + f(w) \) \( O(1) \)

8. Make \( v \) the left child of \( u \) \( O(1) \)

9. Make \( w \) the right child of \( u \) \( O(1) \)

10. **END**

\( n = \# \text{ characters} \quad n = \# \text{ distinct} \quad \Theta(n) \)
Implementing Huffman’s Algorithm

Huffman($T$)
1. Compute the set $C$ of characters in $T$ and determine their frequencies $f(c)$
2. Create one node $v_c$ for each character in $c$ and insert it into priority queue $Q$ \[ O(n lg n) \]
3. while $|Q| > 1$
4. do $v \leftarrow \text{DELETEMIN}(Q)$
5. $w \leftarrow \text{DELETEMIN}(Q)$
6. Create a new node $u$
7. $f(u) \leftarrow f(v) + f(w)$
8. Make $v$ the left child of $u$
9. Make $w$ the right child of $u$
10. INSERT($Q, u$)

Lemma: Huffman’s algorithm takes $O((n + m) lg n)$ time, where $m$ is the number of characters in $T$ and $n$ is the number of distinct characters in $T$. 
Huffman’s Algorithm is Greedy
Huffman’s Algorithm is Greedy

- By merging two trees $T_1$ and $T_2$, we add one bit to the code of every character in $T_1$ and $T_2$. 
Huffman’s Algorithm is Greedy

By merging two trees $T_1$ and $T_2$, we add one bit to the code of every character in $T_1$ and $T_2$.

By merging the trees with minimum frequency, we grow the encoding of the fewest characters and thereby add the fewest bits to the encoding of $T$. 
Lemma: There is an optimal prefix code for $T$ in which the two least frequent characters are sibling leaves.

Proof:

Assumption: $f(x) \leq f(y) \leq f(x') \leq f(y')$
Assumption:
\[ f(x) \leq f(y) \leq f(x') \leq f(y') \]

Cost(\(T, C_{T'}\)) − Cost(\(T, C_T\)) =
Assumption:

\[ f(x) \leq f(y) \leq f(x') \leq f(y') \]

\[
\text{Cost}(\mathcal{T}, C_{T'}) - \text{Cost}(\mathcal{T}, C_T) = \frac{f(x)d_{T'}(x) + f(y)d_{T'}(y) + f(x')d_{T'}(x') + f(y')d_{T'}(y') - f(x)d_T(x) - f(y)d_T(y) - f(x')d_T(x') - f(y')d_T(y')}{\text{Cost of encoding with } T'}
\]
Assumption:
\[ f(x) \leq f(y) \leq f(x') \leq f(y') \]

\[
\text{Cost}(\mathcal{T}, C_{T'}) - \text{Cost}(\mathcal{T}, C_T) = f(x)d_{T'}(x) + f(y)d_{T'}(y) + f(x')d_{T'}(x') + f(y')d_{T'}(y') - f(x)d_T(x) - f(y)d_T(y) - f(x')d_T(x') - f(y')d_T(y')
\]
\[
= f(x)d_T(x) + f(y)d_T(y) + f(x')d_T(x) + f(y')d_T(y) - f(x)d_T(x) - f(y)d_T(y) - f(x')d_T(x') - f(y')d_T(y')
\]
Cost($T, C_{T'}$) − Cost($T, C_T$) = $f(x)d_{T'}(x) + f(y)d_{T'}(y) + f(x')d_{T'}(x') + f(y')d_{T'}(y') -$

$f(x)d_T(x) - f(y)d_T(y) - f(x')d_T(x') - f(y')d_T(y')$

$= \underbrace{f(x)d_T(x)} + \underbrace{f(y)d_T(y)} + \underbrace{f(x')d_T(x)} + \underbrace{f(y')d_T(y)} -$

$\underbrace{f(x)d_{T'}(x)} - \underbrace{f(y)d_{T'}(y)} - \underbrace{f(x')d_{T'}(x')} - \underbrace{f(y')d_{T'}(y')}$

$= (f(x) - f(x'))(d_T(x') - d_T(x)) +$

$\underbrace{(f(y) - f(y'))(d_T(y') - d_T(y))}$

Assumption:

$f(x) \leq f(y) \leq f(x') \leq f(y')$
Assumption:
\[ f(x) \leq f(y) \leq f(x') \leq f(y') \]

\[
\text{Cost}(\mathcal{T}, C_{T'}) - \text{Cost}(\mathcal{T}, C_T) = f(x)d_{T'}(x) + f(y)d_{T'}(y) + f(x')d_{T'}(x') + f(y')d_{T'}(y') - f(x)d_T(x) - f(y)d_T(y) - f(x')d_T(x') - f(y')d_T(y')
\]

\[
= f(x)d_T(x') + f(y)d_T(y') + f(x')d_T(x) + f(y')d_T(y) - f(x)d_T(x) - f(y)d_T(y) - f(x')d_T(x') - f(y')d_T(y')
\]

\[
= (f(x) - f(x'))(d_T(x') - d_T(x)) + (f(y) - f(y'))(d_T(y') - d_T(y)) \leq 0.
\]
The previous lemma captures the idea Huffman’s algorithm is based on:
The previous lemma captures the idea Huffman’s algorithm is based on:

- Make the two least frequent characters, \( x \) and \( y \), siblings and replace all their occurrences in \( T \) with a new character \( z \).
Understanding Huffman’s Algorithm

The previous lemma captures the idea Huffman’s algorithm is based on:

- Make the two least frequent characters, $x$ and $y$, siblings and replace all their occurrences in $T$ with a new character $z$

- The resulting text $T'$ contains $n - 1$ distinct characters
Understanding Huffman’s Algorithm

The previous lemma captures the idea Huffman’s algorithm is based on:

- Make the two least frequent characters, \( x \) and \( y \), siblings and replace all their occurrences in \( T \) with a new character \( z \).

- The resulting text \( T' \) contains \( n - 1 \) distinct characters.

- “Recursively” find an optimal prefix code \( C' \) for \( T' \).
The previous lemma captures the idea Huffman’s algorithm is based on:

- Make the two least frequent characters, $x$ and $y$, siblings and replace all their occurrences in $T$ with a new character $z$.
- The resulting text $T'$ contains $n - 1$ distinct characters.
- “Recursively” find an optimal prefix code $C'$ for $T'$.
- Compute $C$ as

$$C'(c) = \begin{cases} 
C''(c) & \text{if } c \notin \{x, y\} \\
C''(z) \circ 0 & \text{if } c = x \\
C''(z) \circ 1 & \text{if } c = y
\end{cases}$$
Lemma: Huffman’s algorithm computes a minimum-length prefix code for text $T$. 
Correctness of Huffman’s Algorithm

**Lemma:** Huffman’s algorithm computes a minimum-length prefix code for text $T$.

**Proof by induction on $n$:**

Base case: ($n = 2$)
Inductive step: \((n > 2)\)

- \(\text{Cost}(S) = \text{Cost}(T) - f(x) - f(y)\)
- \(\text{Cost}(S') = \text{Cost}(T') - f(x) - f(y)\)
Inductive step: \((n > 2)\)

\[\text{Cost}(S) = \text{Cost}(T) - f(x) - f(y)\]

\[\text{Cost}(S') = \text{Cost}(T') - f(x) - f(y)\]

\[\therefore \text{Cost}(T') < \text{Cost}(T) \Rightarrow \text{Cost}(S') < \text{Cost}(S)\]
Inductive step: \((n > 2)\)

- \(\text{Cost}(S) = \text{Cost}(T) - f(x) - f(y)\)
- \(\text{Cost}(S') = \text{Cost}(T') - f(x) - f(y)\)

\[ \therefore \text{Cost}(T') < \text{Cost}(T) \Rightarrow \text{Cost}(S') < \text{Cost}(S) \]

This is a contradiction.
Greedy algorithms use natural local criteria to make progress towards a solution.

This is a vague concept.

Many good heuristics are greedy:
- Simple
- Work well in practice

Proof that a greedy algorithm produces an optimal solution:
- Induction
- “Stay ahead” arguments
- Exchange arguments