# Analysis of Algorithms

## Greedy Methods

Andres Mendez-Vazquez

February 19, 2018

# Outline

# Outline

# Steps of the greedy method

## Proceed as follows

- Determine the optimal substructure.
- Develop a recursive solution.
- Prove that at any stage of recursion, one of the optimal choices is the greedy choice.
- Show that all but one of the sub-problems resulting from the greedy choice are empty.
- Develop a recursive greedy algorithm.
- Convert it to an iterative algorithm.

# Steps of the greedy method

## Proceed as follows

- Determine the optimal substructure.
- Develop a recursive solution.
- Prove that at any stage of recursion, one of the optimal choices is the greedy choice.
- Show that all but one of the sub-problems resulting from the greedy choice are empty.
- Develop a recursive greedy algorithm.
- Convert it to an iterative algorithm.

# Steps of the greedy method

## Proceed as follows

- Determine the optimal substructure.
- Develop a recursive solution.
- Prove that at any stage of recursion, one of the optimal choices is the greedy choice.
- Show that all but one of the sub-problems resulting from the greedy choice are empty.
- Develop a recursive greedy algorithm.
- Convert it to an iterative algorithm.

# Steps of the greedy method

## Proceed as follows

- Determine the optimal substructure.
- Develop a recursive solution.
- Prove that at any stage of recursion, one of the optimal choices is the greedy choice.
- Show that all but one of the sub-problems resulting from the greedy choice are empty.
- Develop a recursive greedy algorithm.
- Convert it to an iterative algorithm.

# Steps of the greedy method

## Proceed as follows

- Determine the optimal substructure.
- Develop a recursive solution.
- Prove that at any stage of recursion, one of the optimal choices is the greedy choice.
- Show that all but one of the sub-problems resulting from the greedy choice are empty.
- Develop a recursive greedy algorithm.
- Convert it to an iterative algorithm.

# Steps of the greedy method

## Proceed as follows

- Determine the optimal substructure.
- Develop a recursive solution.
- Prove that at any stage of recursion, one of the optimal choices is the greedy choice.
- Show that all but one of the sub-problems resulting from the greedy choice are empty.
- Develop a recursive greedy algorithm.
- Convert it to an iterative algorithm.

# Outline

# Dynamic Programming vs Greedy Method

## Dynamic Programming

- Make a choice at each step
- Choice depends on knowing optimal solutions to sub-problems. Solve sub-problems first.
- Solve bottom-up.

# Dynamic Programming vs Greedy Method

## Dynamic Programming

- Make a choice at each step
- Choice depends on knowing optimal solutions to sub-problems. Solve sub-problems first.
- Solve bottom-up.

## Greedy Method

- Make a choice at each step.
- Make the choice before solving the sub-problems.
- Solve top-down.

# Dynamic Programming vs Greedy Method

## Dynamic Programming

- Make a choice at each step
- Choice depends on knowing optimal solutions to sub-problems. Solve sub-problems first.
- Solve bottom-up.

## Greedy Method

- Make a choice at each step.
- Make the choice before solving the sub-problems.
- Solve top-down.

# Dynamic Programming vs Greedy Method

## Dynamic Programming

- Make a choice at each step
- Choice depends on knowing optimal solutions to sub-problems. Solve sub-problems first.
- Solve bottom-up.

## Greedy Method

- Make a choice at each step.
- Make the choice before solving the sub-problems.
- Solve top-down.

# Dynamic Programming vs Greedy Method

## Dynamic Programming

- Make a choice at each step
- Choice depends on knowing optimal solutions to sub-problems. Solve sub-problems first.
- Solve bottom-up.

## Greedy Method

- Make a choice at each step.
- Make the choice before solving the sub-problems.
- Solve top-down.

# Dynamic Programming vs Greedy Method

## Dynamic Programming

- Make a choice at each step
- Choice depends on knowing optimal solutions to sub-problems. Solve sub-problems first.
- Solve bottom-up.

## Greedy Method

- Make a choice at each step.
- Make the choice before solving the sub-problems.
- Solve top-down.

# Outline

# You are a thief

**You get into a store**

With a knapsack/bag

**The bag has capacity $W$**

You want to select items to fill the bag...

**Question**

How do you do it?

# You are a thief

**You get into a store**

With a knapsack/bag

**The bag has capacity $W$**

You want to select items to fill the bag...

**Question**

How do you do it?

# You are a thief

**You get into a store**

With a knapsack/bag

**The bag has capacity $W$**

You want to select items to fill the bag...

**Question**

How do you do it?

# Formalization

## First

- You have $n$ items.
- Each item is worth $v_i$ and it weights $w_i$ pounds.
- The knapsack can stand a weight of $W$.

# Formalization

## First

- You have $n$ items.
- Each item is worth $v_i$ and it weights $w_i$ pounds.
- The knapsack can stand a weight of $W$.

## Second

- You need to find a subset of items with total weight $\leq W$ such that you have the best profit!!!
- **After all you want to be a successful THIEF!!!**

# Formalization

## First

- You have $n$ items.
- Each item is worth $v_i$ and it weights $w_i$ pounds.
- The knapsack can stand a weight of $W$.

## Second

- You need to find a subset of items with total weight $\leq W$ such that you have the best profit!!!
- **After all you want to be a successful THIEF!!!**

## Decisions?

You can actually use a vector to represent your decisions

$$\langle x_1, x_2, x_3, \ldots, x_n \rangle \quad (1)$$

# Formalization

## First

- You have $n$ items.
- Each item is worth $v_i$ and it weights $w_i$ pounds.
- The knapsack can stand a weight of $W$.

## Second

- You need to find a subset of items with total weight $\leq W$ such that you have the best profit!!!
- After all you want to be a successful THIEF!!!

## Decisions?

You can actually use a vector to represent your decisions

$$\langle x_1, x_2, x_3, ..., x_n \rangle \quad (1)$$

# Formalization

## First

- You have $n$ items.
- Each item is worth $v_i$ and it weights $w_i$ pounds.
- The knapsack can stand a weight of $W$.

## Second

- You need to find a subset of items with total weight $\leq W$ such that you have the best profit!!!
- **After all you want to be a successful THIEF!!!**

## Decisions?

You can actually use a vector to represent your decisions

$$\langle x_1, x_2, x_3, \ldots, x_n \rangle \tag{1}$$

# Formalization

## First

- You have $n$ items.
- Each item is worth $v_i$ and it weights $w_i$ pounds.
- The knapsack can stand a weight of $W$.

## Second

- You need to find a subset of items with total weight $\leq W$ such that you have the best profit!!!
- **After all you want to be a successful THIEF!!!**

## Decisions?

You can actually use a vector to represent your decisions

$$\langle x_1, x_2, x_3, ..., x_n \rangle \tag{1}$$

# You have two versions

## 0-1 knapsack problem

- You have to either take an item or not take it, you cannot take a fraction of it.
- Thus, elements in the vector are $x_i \in \{0, 1\}$ with $i = 1, ..., n$.

# You have two versions

## 0-1 knapsack problem

- You have to either take an item or not take it, you cannot take a fraction of it.
- Thus, elements in the vector are $x_i \in \{0, 1\}$ with $i = 1, ..., n$.

## Fractional knapsack problem

- Like the 0-1 knapsack problem, but you can take a fraction of an item.
- Thus, elements in the vector are $x_i \in [0, 1]$ with $i = 1, ..., n$.

# You have two versions

## 0-1 knapsack problem

- You have to either take an item or not take it, you cannot take a fraction of it.
- Thus, elements in the vector are $x_i \in \{0, 1\}$ with $i = 1, ..., n$.

## Fractional knapsack problem

- Like the 0-1 knapsack problem, but you can take a fraction of an item.
- Thus, elements in the vector are $x_i \in [0, 1]$ with $i = 1, ..., n$.

# You have two versions

## 0-1 knapsack problem

- You have to either take an item or not take it, you cannot take a fraction of it.
- Thus, elements in the vector are $x_i \in \{0, 1\}$ with $i = 1, ..., n$.

## Fractional knapsack problem

- Like the 0-1 knapsack problem, but you can take a fraction of an item.
- Thus, elements in the vector are $x_i \in [0, 1]$ with $i = 1, ..., n$.

# Outline

# Greedy Process for 0-1 Knapsack

**First, You choose an ordering**

What about per price of each item?

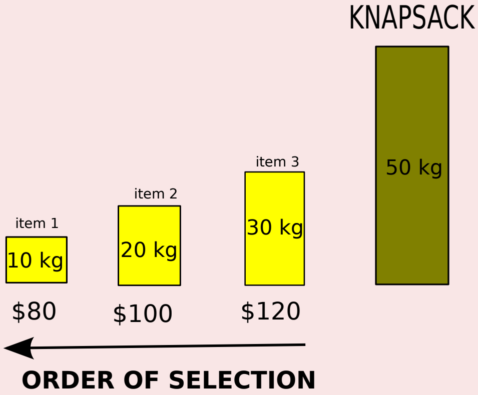
If we have the following situation

# Greedy Process for 0-1 Knapsack

## First, You choose an ordering

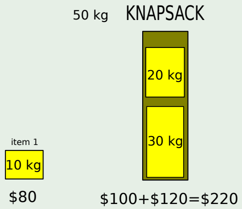
What about per price of each item?

## If we have the following situation



# Thus

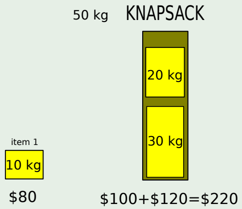
## It works fine



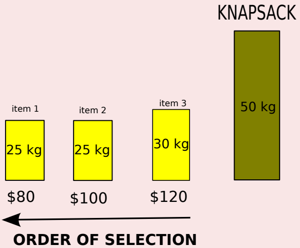

## What about this?

# Thus

## It works fine




## What about this?



# Thus, we need a better way to select elements!!!

**Actually**

Why not to use the price of kg?

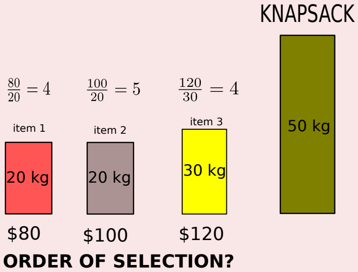
Thus, we have this!!!

# Thus, we need a better way to select elements!!!

## Actually

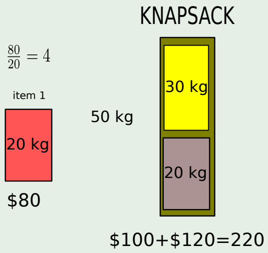
Why not to use the price of kg?

## Thus, we have this!!!



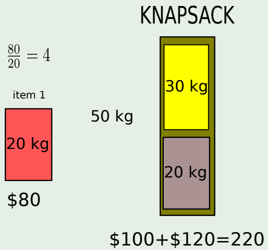
# Did you notice this?

## First

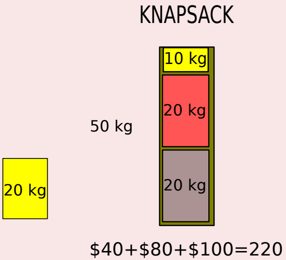


# Did you notice this?

## First

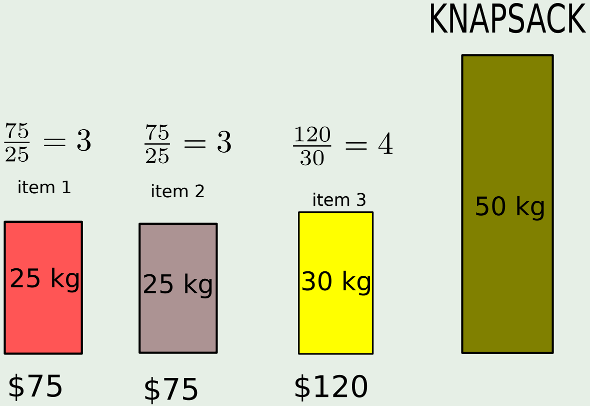



## Second



# However!!!

## Even with an order based in $\frac{v}{w}$ 0-1 Knapsack fails!!!




**ORDER OF SELECTION?**

# Outline

# Definition of Fractional Process

**First**

Push object indexes into a max heap using the key $\frac{v_i}{w_i}$ for $i = 1, ..., n$.

# Definition of Fractional Process

**First**

Push object indexes into a max heap using the key $\frac{v_i}{w_i}$ for $i = 1, ..., n$.

**Then**

- Extract index at the top of the max heap.
- Take the element represented by the index at the top of the max heap and push it into the knapsack.
- Reduce the remaining carry weight by the weight of the element

# Definition of Fractional Process

**First**

Push object indexes into a max heap using the key $\frac{v_i}{w_i}$ for $i = 1, ..., n$.

**Then**

- Extract index at the top of the max heap.
- Take the element represented by the index at the top of the max heap and push it into the knapsack.
- Reduce the remaining carry weight by the weight of the element

**Finally**

If a fraction of space exist, push the next element fraction sorted by key into the knapsack.

# Definition of Fractional Process

## First

Push object indexes into a max heap using the key $\frac{v_i}{w_i}$ for $i = 1, ..., n$.

## Then

- Extract index at the top of the max heap.
- Take the element represented by the index at the top of the max heap and push it into the knapsack.
- Reduce the remaining carry weight by the weight of the element

## Finally

If a fraction of space exist, push the next element fraction sorted by key into the knapsack.

# Definition of Fractional Process

## First

Push object indexes into a max heap using the key $\frac{v_i}{w_i}$ for $i = 1, ..., n$.

## Then

- Extract index at the top of the max heap.
- Take the element represented by the index at the top of the max heap and push it into the knapsack.
- Reduce the remaining carry weight by the weight of the element

## Finally

If a fraction of space exist, push the next element fraction sorted by key into the knapsack.

# Theorem about Greedy Choice

**Theorem**

The greedy choice, which always selects the object with better ratio value/weight, always finds an optimal solution to the Fractional Knapsack problem.

**Proof**

Constraints:

- $x_i \in [0, 1]$

# Theorem about Greedy Choice

**Theorem**

The greedy choice, which always selects the object with better ratio value/weight, always finds an optimal solution to the Fractional Knapsack problem.

**Proof**

Constraints:

- $x_i \in [0, 1]$

# Fractional Greedy

**FRACTIONAL-KNAPSACK$(W, w, v)$**

1. for $i = 1$ to n do $x[i] = 0$
2. $weight = 0$
3. // Use a Max-Heap
4. T = Build-Max-Heap$(v/w)$
5. while $weight < W$ do
6. i = T.Heap-Extract-Max()
7. if $(weight + w[i] \leq W)$ do
8. $x[i] = 1$
9. $weight = weight + w[i]$
10. else
11. $x[i] = \frac{W - weight}{w[i]}$
12. $weight = W$
13. return $x$

# Fractional Greedy

**FRACTIONAL-KNAPSACK$(W, w, v)$**

1. for $i = 1$ to n do $x[i] = 0$
2. $weight = 0$
3. // Use a Max-Heap
4. T = Build-Max-Heap$(v/w)$
5. while $weight < W$ do
6. i = T.Heap-Extract-Max()
7. if $(weight + w[i] \leq W)$ do
8. $x[i] = 1$
9. $weight = weight + w[i]$
10. else
11. $x[i] = \frac{W - weight}{w[i]}$
12. $weight = W$
13. return $x$

# Fractional Greedy

**FRACTIONAL-KNAPSACK$(W, w, v)$**

```
for i = 1 to n do x[i] = 0
weight = 0
// Use a Max-Heap
T = Build-Max-Heap(v/w)
while weight < W do
        i = T.Heap-Extract-Max()
        if (weight + w[i] ≤ W) do
                x[i] = 1
                weight = weight + w[i]
        else
                x[i] = (W − weight) / w[i]
                weight = W
return x
```

# Fractional Greedy

**FRACTIONAL-KNAPSACK$(W, w, v)$**

for $i = 1$ to n do $x[i] = 0$
$weight = 0$
// Use a Max-Heap
T = Build-Max-Heap$(\boldsymbol{v}/\boldsymbol{w})$
while $weight < W$ do
    i = T.Heap-Extract-Max()
    if $(weight + w[i] \leq W)$ do
        $x[i] = 1$
        $weight = weight + w[i]$
    else
        $x[i] = \frac{W - weight}{w[i]}$
        $weight = W$
return $x$

# Fractional Greedy

**FRACTIONAL-KNAPSACK$(W, w, v)$**

for $i = 1$ to n do $x[i] = 0$
$weight = 0$
// Use a Max-Heap
T $=$ Build-Max-Heap$(\boldsymbol{v}/\boldsymbol{w})$
while $weight < W$ do
    i $=$ T.Heap-Extract-Max()
    if $(weight + w[i] \leq W)$ do
        $x[i] = 1$
        $weight = weight + w[i]$
    else
        $x[i] = \frac{W - weight}{w[i]}$
        $weight = W$
return $x$

# Fractional Greedy

**FRACTIONAL-KNAPSACK$(W, w, v)$**

for $i = 1$ to n do $x[i] = 0$

$weight = 0$

// Use a Max-Heap

T = Build-Max-Heap$(\boldsymbol{v}/\boldsymbol{w})$

while $weight < W$ do

    i = T.Heap-Extract-Max()

    if $(weight + w[i] \leq W)$ do

        $x[i] = 1$

        $weight = weight + w[i]$

    else

        $x[i] = \frac{W - weight}{w[i]}$

        $weight = W$

return $x$

# Fractional Greedy

**FRACTIONAL-KNAPSACK$(W, w, v)$**

```
for i = 1 to n do x[i] = 0
weight = 0
// Use a Max-Heap
T = Build-Max-Heap(v/w)
while weight < W do
      i = T.Heap-Extract-Max()
      if (weight + w[i] ≤ W) do
            x[i] = 1
            weight = weight + w[i]
      else
            x[i] = (W − weight)/w[i]
            weight = W
return x
```

# Fractional Greedy

**FRACTIONAL-KNAPSACK$(W, w, v)$**

```
for i = 1 to n do x[i] = 0
weight = 0
// Use a Max-Heap
T = Build-Max-Heap(v/w)
while weight < W do
        i = T.Heap-Extract-Max()
        if (weight + w[i] ≤ W) do
                x[i] = 1
                weight = weight + w[i]
        else
                x[i] = (W − weight) / w[i]
                weight = W
return x
```

# Fractional Greedy

**FRACTIONAL-KNAPSACK$(W, w, v)$**

```
for i = 1 to n do x[i] = 0
weight = 0
// Use a Max-Heap
T = Build-Max-Heap(v/w)
while weight < W do
        i = T.Heap-Extract-Max()
        if (weight + w[i] ≤ W) do
                x[i] = 1
                weight = weight + w[i]
        else
                x[i] = (W − weight) / w[i]
                weight = W
return x
```

# Fractional Greedy

**FRACTIONAL-KNAPSACK$(W, w, v)$**

for $i = 1$ to n do $x[i] = 0$

$weight = 0$

// Use a Max-Heap

T $=$ Build-Max-Heap$(\boldsymbol{v}/\boldsymbol{w})$

while $weight < W$ do

    i $=$ T.Heap-Extract-Max()

    if $(weight + w[i] \leq W)$ do

        $x[i] = 1$

        $weight = weight + w[i]$

    else

        $x[i] = \frac{W - weight}{w[i]}$

        $weight = W$

return $x$

# Fractional Greedy

## Complexity

- Under the fact that this algorithm is using a heap we can get the complexity $O(n \log n)$.
- If we assume already an initial sorting or use a linear sorting we get complexity $O(n)$.

# Fractional Greedy

## Complexity

- Under the fact that this algorithm is using a heap we can get the complexity $O(n \log n)$.
- If we assume already an initial sorting or use a linear sorting we get complexity $O(n)$.

# Outline

# Activity selection

## Problem

Set of activities $S = a_1, ..., a_n$. The $a_i$ activity needs a resource (Class Room, Machine, Assembly Line, etc) during period $[s_i, f_i)$, which is a half-open interval, where $s_i$ is the start time of activity $a_i$ and $f_i$ is the finish time of activity $a_i$.

## For example

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

## Goal:

Select the largest possible set of non-overlapping activities.

# Activity selection

## Problem

Set of activities $S = a_1, ..., a_n$. The $a_i$ activity needs a resource (Class Room, Machine, Assembly Line, etc) during period $[s_i, f_i)$, which is a half-open interval, where $s_i$ is the start time of activity $a_i$ and $f_i$ is the finish time of activity $a_i$.

## For example

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

## Goal:

Select the largest possible set of non-overlapping activities.

# Activity selection

## Problem

Set of activities $S = a_1, ..., a_n$. The $a_i$ activity needs a resource (Class Room, Machine, Assembly Line, etc) during period $[s_i, f_i)$, which is a half-open interval, where $s_i$ is the start time of activity $a_i$ and $f_i$ is the finish time of activity $a_i$.

## For example

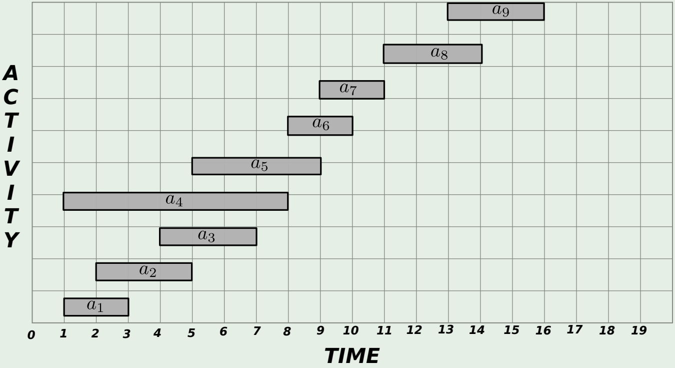
| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| $s_i$ | 1 | 2 | 4 | 1 | 5 | 8 | 9 | 11 | 13 |
| $f_i$ | 3 | 5 | 7 | 8 | 9 | 10 | 11 | 14 | 16 |

## Goal:

Select the largest possible set of non-overlapping activities.

# We have something like this

## Example

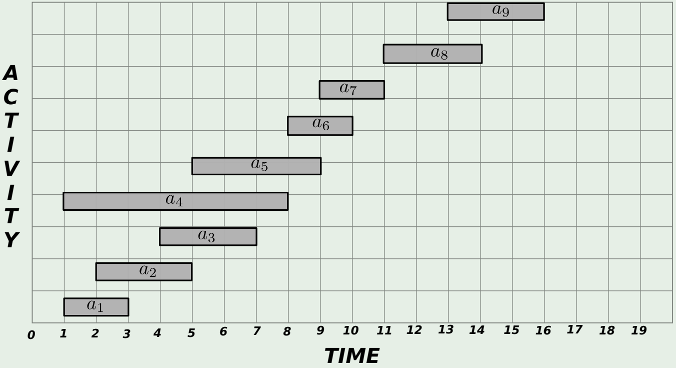



## Goal:

Select the largest possible set of non-overlapping activities.

# We have something like this

## Example

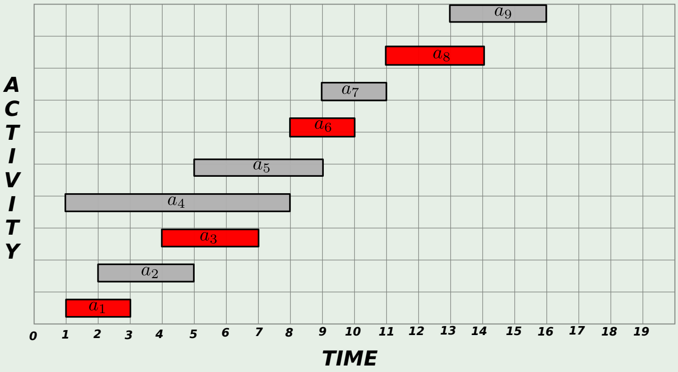



## Goal:

Select the largest possible set of non-overlapping activities.

# For Example

## Example



# Thus!!!

**First**

The Optimal Substructure!!!

**Second**

We need the recursive solution!!!

**Third**

The Greedy Choice

**Fourth**

Prove it is the only one!!!

**Fifth**

We need the iterative solution!!!

# Thus!!!

**First**

The Optimal Substructure!!!

**Second**

We need the recursive solution!!!

**Third**

The Greedy Choice

**Fourth**

Prove it is the only one!!!

**Fifth**

We need the iterative solution!!!

# Thus!!!

**First**

The Optimal Substructure!!!

**Second**

We need the recursive solution!!!

**Third**

The Greedy Choice

**Fourth**

Prove it is the only one!!!

**Fifth**

We need the iterative solution!!!

# Thus!!!

## First

The Optimal Substructure!!!

## Second

We need the recursive solution!!!

## Third

The Greedy Choice

## Fourth

Prove it is the only one!!!

## Fifth

We need the iterative solution!!!

# Thus!!!

**First**

The Optimal Substructure!!!

**Second**

We need the recursive solution!!!

**Third**

The Greedy Choice

**Fourth**

Prove it is the only one!!!

**Fifth**

We need the iterative solution!!!

# How do we discover the Optimal Structure

**We know we have the following**

$S = a_1, ..., a_n$, a set of activities.

**We can then refine the set of activities in the following way**

We define:

- $S_{ij}$ = the set of activities that start after activity $a_i$ finishes and that finishes before activity $a_j$ start.

# How do we discover the Optimal Structure

**We know we have the following**

$S = a_1, ..., a_n$, a set of activities.

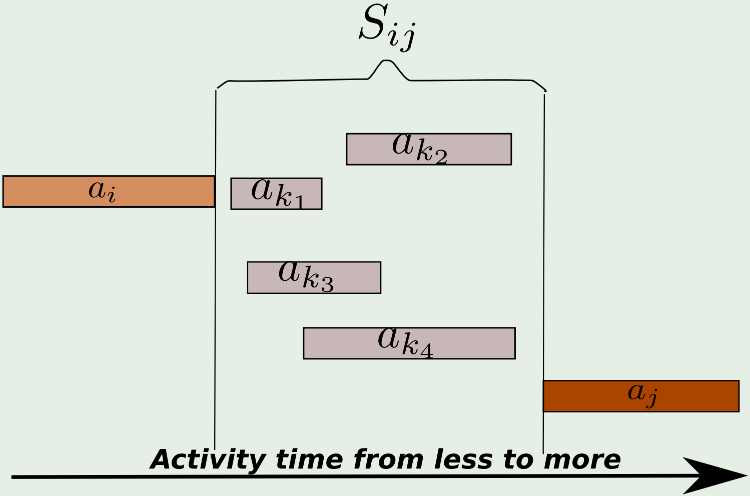
**We can then refine the set of activities in the following way**

We define:

- $S_{ij}$ = the set of activities that start after activity $a_i$ finishes and that finishes before activity $a_j$ start.

# The Optimal Substructure

Thus, we have



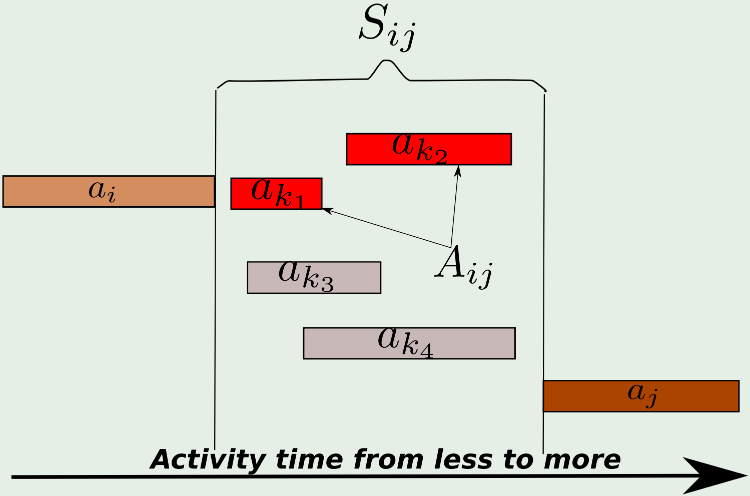
# The Optimal Substructure

## Second

- Suppose that the set $A_{ij}$ denotes the maximum set of compatible activities for $S_{ij}$

# For Example

## Thus, we have



# Then

## Third

- In addition assume that $A_{ij}$ includes some activity $a_k$.
- Then, imagine that $a_k$ belongs to some optimal solution.

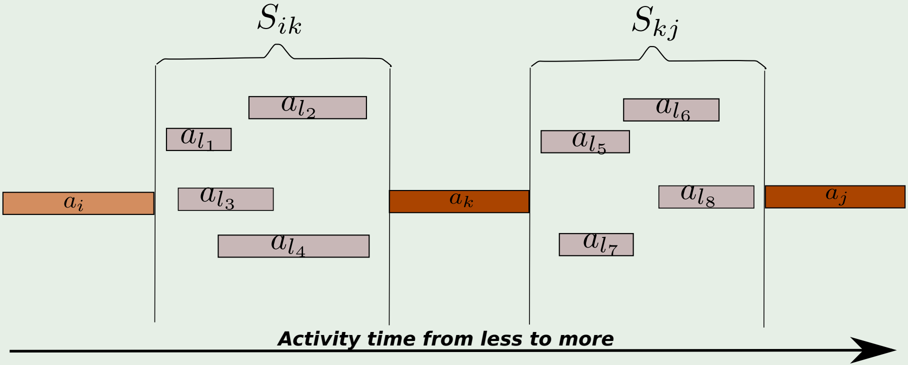
# Then

## Third

- In addition assume that $A_{ij}$ includes some activity $a_k$.
- Then, imagine that $a_k$ belongs to some optimal solution.

# What do we need?

We need to find the optimal solutions for $S_{ik}$ and $S_{kj}$.



# The Optimal Substructure

## Then

We need to prove the optimal substructure:

- For this we will use
  - The Cut-and-Paste Argument
  - Contradiction

# The Optimal Substructure

## Then

We need to prove the optimal substructure:

- For this we will use
  - The Cut-and-Paste Argument
  - Contradiction

## First

We have that $A_{ik} = S_{ik} \cap A_{ij}$ and similarly $A_{kj} = S_{kj} \cap A_{ij}$.

# The Optimal Substructure

## Then

We need to prove the optimal substructure:

- For this we will use
  - The Cut-and-Paste Argument
  - Contradiction

## First

We have that $A_{ik} = S_{ik} \cap A_{ij}$ and similarly $A_{kj} = S_{kj} \cap A_{ij}$.

## Then

$A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$ or $|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$

# The Optimal Substructure

## Then

We need to prove the optimal substructure:

- For this we will use
  - The Cut-and-Paste Argument
  - Contradiction

## First

We have that $A_{ik} = S_{ik} \cap A_{ij}$ and similarly $A_{kj} = S_{kj} \cap A_{ij}$.

## Then

$A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$ or $|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$

# The Optimal Substructure

## Then

We need to prove the optimal substructure:

- For this we will use
  - The Cut-and-Paste Argument
  - Contradiction

## First

We have that $A_{ik} = S_{ik} \cap A_{ij}$ and similarly $A_{kj} = S_{kj} \cap A_{ij}$.

## Then

$A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$ or $|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$

# The Optimal Substructure

## Then

We need to prove the optimal substructure:

- For this we will use
  - The Cut-and-Paste Argument
  - Contradiction

## First

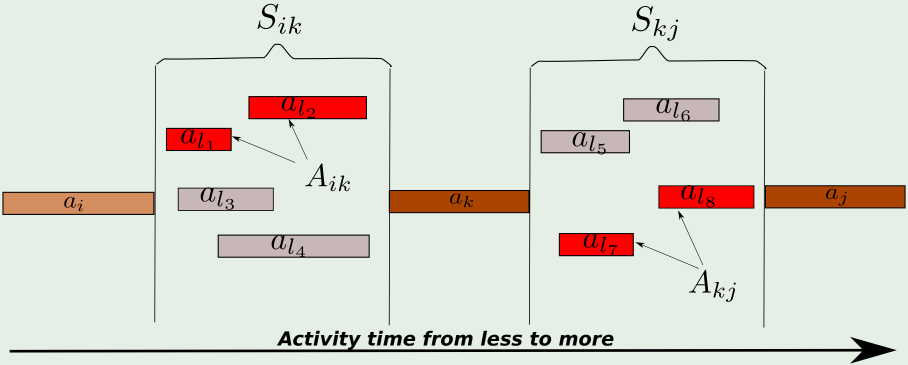
We have that $A_{ik} = S_{ik} \cap A_{ij}$ and similarly $A_{kj} = S_{kj} \cap A_{ij}$.

## Then

$A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$ or $|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$

# For Example

We need to find the optimal solutions for $S_{ik}$ and $S_{kj}$.



# The Optimal Substructure: Using Contradictions

## Use Cut-and-Paste Argument

How? Assume that exist $A_{ik}^{'}$ such that $\left|A_{ik}^{'}\right| > |A_{ik}|$.

# The Optimal Substructure: Using Contradictions

## Use Cut-and-Paste Argument

How? Assume that exist $A'_{ik}$ such that $\left|A'_{ik}\right| > |A_{ik}|$.

## Meaning

- We assume that we cannot construct the large answer using the small answers!!!
- The Basis of Contradiction $(S \cup \{\neg P\} \vdash F) \implies (S \vdash P)$

# The Optimal Substructure: Using Contradictions

## Use Cut-and-Paste Argument

How? Assume that exist $A_{ik}^{'}$ such that $\left|A_{ik}^{'}\right| > |A_{ik}|$.

## Meaning

- We assume that we cannot construct the large answer using the small answers!!!
- The Basis of Contradiction $(S \cup \{\neg P\} \vdash \boldsymbol{F}) \Longrightarrow (S \vdash P)$

# Important

## Here

$\neg P =$exist $A^{'}_{ik}$ such that $\left|A^{'}_{ik}\right| > |A_{ik}|$.

# Then

**This means that $A'_{ij}$ has more activities than $A_{ij}$**

Then $\left|A'_{ij}\right| = \left|A'_{ik}\right| + |A_{kj}| + 1 > |A_{ik}| + |A_{kj}| + 1 = |A_{ij}|$, which is a **contradiction**.

**Finally**

We have the optimal-substructure.

# Then

**This means that $A'_{ij}$ has more activities than $A_{ij}$**

Then $\left|A'_{ij}\right| = \left|A'_{ik}\right| + |A_{kj}| + 1 > |A_{ik}| + |A_{kj}| + 1 = |A_{ij}|$, which is a **contradiction**.

**Finally**

**We have the optimal-substructure.**

# Then

**Given**

$A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$ or $|A_{ij}| = |A_{ik}| + |A_{kj}| + 1.$

**Question**

Can anybody give me the recursion?

# Then

**Given**

$A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$ or $|A_{ij}| = |A_{ik}| + |A_{kj}| + 1.$

**Question**

Can anybody give me the recursion?

# Recursive Formulation

**Recursive Formulation**

$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \varnothing \\ \max\limits_{\substack{i<k<j \\ a_k \in S_{ij}}} c[i,k] + c[k,j] + 1 & \text{if } S_{ij} \neq \varnothing \end{cases}$$

Did you notice that you can use Dynamic Programming?

# Recursive Formulation

## Recursive Formulation

$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \varnothing \\ \max\limits_{\substack{i<k<j \\ a_k \in S_{ij}}} c[i,k] + c[k,j] + 1 & \text{if } S_{ij} \neq \varnothing \end{cases}$$

Did you notice that you can use Dynamic Programming?

# We have our Goal?

**Thus**

Any Ideas to obtain the largest possible set using Greedy Choice?

# We have our Goal?

## Thus

Any Ideas to obtain the largest possible set using Greedy Choice?

## What about?

- Early Starting Time?
- Early Starting + Less Activity Length?

# We have our Goal?

## Thus

Any Ideas to obtain the largest possible set using Greedy Choice?

## What about?

- Early Starting Time?
- Early Starting + Less Activity Length?

## Me!!!

- Do we have something that combines both things?

# We have our Goal?

## Thus

Any Ideas to obtain the largest possible set using Greedy Choice?

## What about?

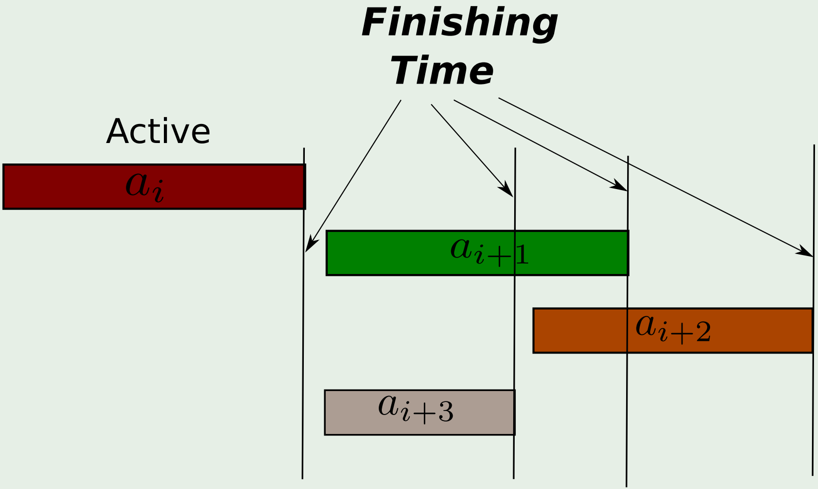
- Early Starting Time?
- Early Starting + Less Activity Length?

## Me!!!

- Do we have something that combines both things?

# Which Greedy Choice?

## Did you notice the following



# The Early Finishing Time

**Yes!!!**

The Greedy Choice = **Select by earliest finishing time**.

# Greedy solution

**Theorem 16.1**

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

# Thus

## We can do the following

- We can repeatedly choose the activity that finishes first.
- Remove all activities incompatible with this activity
- Then repeat

# Thus

## We can do the following

- We can repeatedly choose the activity that finishes first.
- Remove all activities incompatible with this activity
- Then repeat

## Not only that

- Because we always choose the activity with the earliest finish time.
  - Then, finish times of the activities we choose must strictly increase.

# Thus

## We can do the following

- We can repeatedly choose the activity that finishes first.
- Remove all activities incompatible with this activity
- Then repeat

## Not only that

- Because we always choose the activity with the earliest finish time.
  - Then, finish times of the activities we choose must strictly increase.

# Thus

## We can do the following

- We can repeatedly choose the activity that finishes first.
- Remove all activities incompatible with this activity
- Then repeat

## Not only that

- Because we always choose the activity with the earliest finish time.
  - Then, finish times of the activities we choose must strictly increase.

# Thus

## We can do the following

- We can repeatedly choose the activity that finishes first.
- Remove all activities incompatible with this activity
- Then repeat

## Not only that

- Because we always choose the activity with the earliest finish time.
  - Then, finish times of the activities we choose must strictly increase.

# Top-Down Approach

**Something Notable**

An algorithm to solve the activity-selection problem does not need to work bottom-up!!!

**Instead**

It can work top-down, choosing an activity to put into the optimal solution and then solving the subproblem of choosing activities from those that are compatible with those already chosen.

**Important**

Greedy algorithms typically have this top-down design:

- They make a choice and then solve **one** subproblem.

# Top-Down Approach

**Something Notable**

An algorithm to solve the activity-selection problem does not need to work bottom-up!!!

**Instead**

It can work top-down, choosing an activity to put into the optimal solution and then solving the subproblem of choosing activities from those that are compatible with those already chosen.

**Important**

Greedy algorithms typically have this top-down design:

- They make a choice and then solve **one** subproblem.

# Top-Down Approach

## Something Notable

An algorithm to solve the activity-selection problem does not need to work bottom-up!!!

## Instead

It can work top-down, choosing an activity to put into the optimal solution and then solving the subproblem of choosing activities from those that are compatible with those already chosen.

## Important

Greedy algorithms typically have this top-down design:

- They make a choice and then solve **one** subproblem.

# Then

**First**

The activities are already sorted by finishing time

**In addition**

There are dummy activities:

- $a_0$= fictitious activity, $f_0 = 0$.

# Then

**First**

The activities are already sorted by finishing time

**In addition**

There are dummy activities:

- $a_0$= fictitious activity, $f_0 = 0$.

# Recursive Activity Selector

**REC-ACTIVITY-SELECTOR$(s, f, k, n)$ - Entry Point $(s, f, 0, n)$**

1. $m = k + 1$
2. // find the first activity in $S_k$ to finish
3. while $m \leq n$ and $s[m] < f[k]$
4. $\quad m = m + 1$
5. if $m \leq n$
6. $\quad$ return $\{a_m\} \cup$REC-ACTIVITY-SELECTOR$(s, f, m, n)$
7. else return $\emptyset$

# Recursive Activity Selector

**REC-ACTIVITY-SELECTOR$(s, f, k, n)$ - Entry Point $(s, f, 0, n)$**

1. $m = k + 1$
2. // find the first activity in $S_k$ to finish
3. while $m \leq n$ and $s[m] < f[k]$
4. $\quad m = m + 1$
5. if $m \leq n$
6. $\quad$ return $\{a_m\} \cup$REC-ACTIVITY-SELECTOR$(s, f, m, n)$
7. else return $\emptyset$

# Recursive Activity Selector

**REC-ACTIVITY-SELECTOR$(s, f, k, n)$ - Entry Point $(s, f, 0, n)$**

1. $m = k + 1$
2. // find the first activity in $S_k$ to finish
3. while $m \leq n$ and $s[m] < f[k]$
4. $\quad\quad m = m + 1$
5. if $m \leq n$
6. $\quad\quad$ return $\{a_m\} \cup$REC-ACTIVITY-SELECTOR$(s, f, m, n)$
7. else return $\emptyset$

# Recursive Activity Selector

**REC-ACTIVITY-SELECTOR$(s, f, k, n)$ - Entry Point $(s, f, 0, n)$**

1. $m = k + 1$
2. // find the first activity in $S_k$ to finish
3. while $m \leq n$ and $s[m] < f[k]$
4. $\quad\quad m = m + 1$
5. if $m \leq n$
6. $\quad\quad$ return $\{a_m\} \cup$REC-ACTIVITY-SELECTOR$(s, f, m, n)$
7. else return $\emptyset$

# Recursive Activity Selector

**REC-ACTIVITY-SELECTOR$(s, f, k, n)$ - Entry Point $(s, f, 0, n)$**

1. $m = k + 1$
2. // find the first activity in $S_k$ to finish
3. while $m \leq n$ and $s[m] < f[k]$
4. $\qquad m = m + 1$
5. if $m \leq n$
6. $\qquad$ return $\{a_m\} \cup$REC-ACTIVITY-SELECTOR$(s, f, m, n)$
7. else return $\emptyset$

# Do we need the recursion?

**Did you notice?**

We remove all the activities before $f_k$!!!

# We can do more...

## GREEDY-ACTIVITY-SELECTOR$(s, f, n)$

$n = s.length$

$A = \{a_1\}$

$k = 1$

for $m = 2$ to $n$

  if $s[m] \geq f[k]$

    $A = A \cup \{a_m\}$

    $k = m$

return $A$

# We can do more...

**GREEDY-ACTIVITY-SELECTOR$(s, f, n)$**

1. $n = s.length$
2. $A = \{a_1\}$
3. $k = 1$
4. for $m = 2$ to $n$
5. if $s[m] \geq f[k]$
6. $A = A \cup \{a_m\}$
7. $k = m$
8. return $A$

**Complexity of the algorithm**

$$\Theta(n)$$

Note: Clearly, we are not taking into account the sorting of the activities.

# We can do more...

## GREEDY-ACTIVITY-SELECTOR$(s, f, n)$

1. $n = s.length$
2. $A = \{a_1\}$
3. $k = 1$
4. for $m = 2$ to $n$
5. if $s[m] \geq f[k]$
6. $A = A \cup \{a_m\}$
7. $k = m$
8. return $A$

## Complexity of the algorithm

$$\Theta(n)$$

Note: Clearly, we are not taking into account the sorting of the activities.

# We can do more...

**GREEDY-ACTIVITY-SELECTOR$(s, f, n)$**

$n = s.length$

$A = \{a_1\}$

$k = 1$

for $m = 2$ to $n$

    if $s[m] \geq f[k]$

        $A = A \cup \{a_m\}$

        $k = m$

return $A$

**Complexity of the algorithm**

$$\Theta(n)$$

Note: Clearly, we are not taking into account the sorting of the activities.

# We can do more...

## GREEDY-ACTIVITY-SELECTOR$(s, f, n)$

$n = s.length$

$A = \{a_1\}$

$k = 1$

for $m = 2$ to $n$

    if $s[m] \geq f[k]$

        $A = A \cup \{a_m\}$

        $k = m$

return $A$

## Complexity of the algorithm

$$\Theta(n)$$

Note: Clearly, we are not taking into account the sorting of the activities.

# Outline

# Huffman codes

## Use

Huffman codes are widely used and very effective technique for compressing.

# Huffman codes

## Use

Huffman codes are widely used and very effective technique for compressing.

## Advantages

Savings of 20% to 90% are typical, depending on the characteristics of the data being compressed.

- Codes in which no codeword is also a prefix of some other codeword
- It is possible to show that the optimal data compression achievable by a character code can always be achieved with a prefix code.

# Huffman codes

## Use

Huffman codes are widely used and very effective technique for compressing.

## Advantages

Savings of 20% to 90% are typical, depending on the characteristics of the data being compressed.

## Idea

Huffman codes are based on the idea of prefix codes:

- Codes in which no codeword is also a prefix of some other codeword.
- It is possible to show that the optimal data compression achievable by a character code can always be achieved with a prefix code.

# Huffman codes

## Use

Huffman codes are widely used and very effective technique for compressing.

## Advantages

Savings of 20% to 90% are typical, depending on the characteristics of the data being compressed.

## Idea

Huffman codes are based on the idea of prefix codes:

- Codes in which no codeword is also a prefix of some other codeword.
- It is possible to show that the optimal data compression achievable by a character code can always be achieved with a prefix code.

# Huffman codes

## Use

Huffman codes are widely used and very effective technique for compressing.

## Advantages

Savings of 20% to 90% are typical, depending on the characteristics of the data being compressed.

## Idea

Huffman codes are based on the idea of prefix codes:

- Codes in which no codeword is also a prefix of some other codeword.
- It is possible to show that the optimal data compression achievable by a character code can always be achieved with a prefix code.

# Imagine the following

## We have the following

- Imagine having 100,000-character data file, and we want to store it compactly.
- In addition, we have the following distribution of character in the hundred of thousands.

# Imagine the following

## We have the following

- Imagine having 100,000-character data file, and we want to store it compactly.
- In addition, we have the following distribution of character in the hundred of thousands.

## Table

| | a | b | c | d | e | f |
|---|---|---|---|---|---|---|
| Frequency | 45,000 | 13,000 | 12,000 | 16,000 | 9,000 | 5,000 |
| Fixed-Leng cw | 000 | 001 | 010 | 011 | 100 | 101 |
| Vari Leng cw | 0 | 101 | 100 | 111 | 1101 | 1100 |

Table: Distribution of characters in the text and their codewords.

# Imagine the following

## We have the following

- Imagine having 100,000-character data file, and we want to store it compactly.
- In addition, we have the following distribution of character in the hundred of thousands.

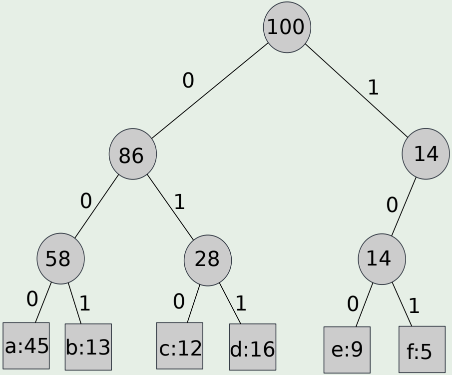
## Table

| | a | b | c | d | e | f |
|---|---|---|---|---|---|---|
| Frequency | 45,000 | 13,000 | 12,000 | 16,000 | 9,000 | 5,000 |
| Fixed-Leng cw | 000 | 001 | 010 | 011 | 100 | 101 |
| Vari Leng cw | 0 | 101 | 100 | 111 | 1101 | 1100 |

Table: Distribution of characters in the text and their codewords.

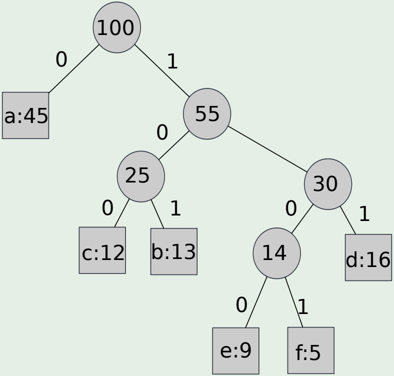
# We have the following representations for the previous codes

## Fix Code: No optimal tree in our problem



# We have the following representations for the previous codes

## Prefix Code: Binary tree for the variable prefix code in table



# Cost in the number of bits to represent the text

**Fix Code**

$$3 \times 100,000 = 300,000 \text{ bits} \tag{2}$$

**Variable Code**

$$[45 \times 1 + 13 \times 3 + 12 \times 3 + 16 \times 3 + ...$$
$$9 \times 4 + 5 \times 4] \times 1000 = 224,000 \text{ bits}$$

# Cost in the number of bits to represent the text

## Fix Code

$$3 \times 100,000 = 300,000 \text{ bits} \quad (2)$$

## Variable Code

$$[45 \times 1 + 13 \times 3 + 12 \times 3 + 16 \times 3 + ...$$
$$9 \times 4 + 5 \times 4] \times 1000 = 224,000 \text{ bits}$$

# Prefix Codes

## Something Notable

It has been show that prefix codes:

- Codes in which no codeword is also a prefix of some other codeword.

# Prefix Codes

## Something Notable

It has been show that prefix codes:

- Codes in which no codeword is also a prefix of some other codeword.

## They have the following nice properties

- Easy to decode.
- They are unambiguous.
  - For example in our example the string 001011101 transform as $0 \circ 0 \circ 101 \circ 1101 = aabe$.

# Prefix Codes

## Something Notable

It has been show that prefix codes:

- Codes in which no codeword is also a prefix of some other codeword.

## They have the following nice properties

- Easy to decode.
- They are unambiguous.
  - For example in our example the string 001011101 transform as $0 \circ 0 \circ 101 \circ 1101 = aabe$.

## Properties

As we can prove an optimal code for a text is always represented by a full binary tree!!!

# Prefix Codes

## Something Notable

It has been show that prefix codes:

- Codes in which no codeword is also a prefix of some other codeword.

## They have the following nice properties

- Easy to decode.
- They are unambiguous.
  - For example in our example the string 001011101 transform as $0 \circ 0 \circ 101 \circ 1101 = aabe$.

## Properties

As we can prove an optimal code for a text is always represented by a full binary tree!!!

# Prefix Codes

## Something Notable

It has been show that prefix codes:

- Codes in which no codeword is also a prefix of some other codeword.

## They have the following nice properties

- Easy to decode.
- They are unambiguous.
  - For example in our example the string 001011101 transform as $0 \circ 0 \circ 101 \circ 1101 = aabe$.

## Properties

As we can prove an optimal code for a text is always represented by a full binary tree!!!

# It is more...

Given that we will concentrate our attention for the prefix codes to full binary tree.

Given that $C$ is the alphabet for the text file.

# It is more...

**Given that we will concentrate our attention for the prefix codes to full binary tree.**

Given that $C$ is the alphabet for the text file.

**We have that**

1. The tree for the optimal prefix code has $|C|$ leaves.
2. The number of internal leaves is $|C| - 1$.
3. Each character $x$ at the leaves has a depth $d_T(x)$ which is the length of the codeword.

# It is more...

**Given that we will concentrate our attention for the prefix codes to full binary tree.**

Given that $C$ is the alphabet for the text file.

**We have that**

1. The tree for the optimal prefix code has $|C|$ leaves.
2. The number of internal leaves is $|C| - 1$.
3. Each character $x$ at the leaves has a depth $d_T(x)$ which is the length of the codeword.

**Thus**

- Knowing the frequency of each character and the tree $T$ representing the optimal prefix encoding.
- We can define the number of bits necessary to encode the text file.

# It is more...

**Given that we will concentrate our attention for the prefix codes to full binary tree.**

Given that $C$ is the alphabet for the text file.

**We have that**

1. The tree for the optimal prefix code has $|C|$ leaves.
2. The number of internal leaves is $|C| - 1$.
3. Each character $x$ at the leaves has a depth $d_T(x)$ which is the length of the codeword.

**Thus**

- Knowing the frequency of each character and the tree $T$ representing the optimal prefix encoding.
- We can define the number of bits necessary to encode the text file.

# It is more...

## Given that we will concentrate our attention for the prefix codes to full binary tree.

Given that $C$ is the alphabet for the text file.

## We have that

1. The tree for the optimal prefix code has $|C|$ leaves.
2. The number of internal leaves is $|C| - 1$.
3. Each character $x$ at the leaves has a depth $d_T(x)$ which is the length of the codeword.

## Thus

- Knowing the frequency of each character and the tree $T$ representing the optimal prefix encoding.
- We can define the number of bits necessary to encode the text file.

# It is more...

**Given that we will concentrate our attention for the prefix codes to full binary tree.**

Given that $C$ is the alphabet for the text file.

**We have that**

1. The tree for the optimal prefix code has $|C|$ leaves.
2. The number of internal leaves is $|C| - 1$.
3. Each character $x$ at the leaves has a depth $d_T(x)$ which is the length of the codeword.

**Thus**

- Knowing the frequency of each character and the tree $T$ representing the optimal prefix encoding.
- We can define the number of bits necessary to encode the text file.

# Cost of Trees for Coding

**Cost in the number of bits for a text**

$$B(T) = \sum_{c \in C} c.freq \times d_T(c).$$

# Now, Which Greedy Choice?

**I leave this to you**

The optimal substructure

**Greedy Choice**

Ideas?

# Now, Which Greedy Choice?

**I leave this to you**

The optimal substructure

**Greedy Choice**

Ideas?

# Now, Which Greedy Choice?

**I leave this to you**

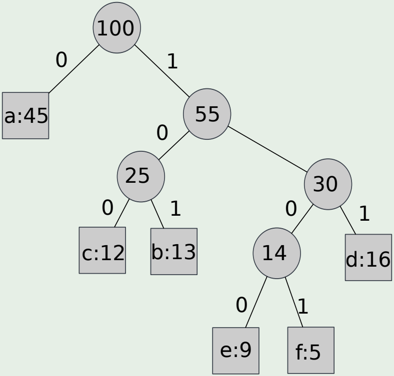
The optimal substructure

**Greedy Choice**

Ideas?

# What about this?

## Prefix Code: Binary tree for the variable prefix code in table



# Greedy Process for Huffman Codes

## Process

- You start with an alphabet $C$ with an associated frequency for each element in it.
- Use the frequencies to build a min priority queue.
- Subtract the two least frequent elements (Greedy Choice).
- Build a three using as children the two nodes of the subtrees extracted from the min priority queue. The new root holds the sum of frequencies of the two subtrees.
- Put it back into the Priority Queue.

# Greedy Process for Huffman Codes

## Process

- You start with an alphabet $C$ with an associated frequency for each element in it.
- Use the frequencies to build a min priority queue.
- Subtract the two least frequent elements (Greedy Choice).
- Build a three using as children the two nodes of the subtrees extracted from the min priority queue. The new root holds the sum of frequencies of the two subtrees.
- Put it back into the Priority Queue.

# Greedy Process for Huffman Codes

## Process

- You start with an alphabet $C$ with an associated frequency for each element in it.
- Use the frequencies to build a min priority queue.
- Subtract the two least frequent elements (Greedy Choice).
- Build a three using as children the two nodes of the subtrees extracted from the min priority queue. The new root holds the sum of frequencies of the two subtrees.
- Put it back into the Priority Queue.

# Greedy Process for Huffman Codes

## Process

- You start with an alphabet $C$ with an associated frequency for each element in it.
- Use the frequencies to build a min priority queue.
- Subtract the two least frequent elements (Greedy Choice).
- Build a three using as children the two nodes of the subtrees extracted from the min priority queue. The new root holds the sum of frequencies of the two subtrees.
- Put it back into the Priority Queue.

# Greedy Process for Huffman Codes

## Process

- You start with an alphabet $C$ with an associated frequency for each element in it.
- Use the frequencies to build a min priority queue.
- Subtract the two least frequent elements (Greedy Choice).
- Build a three using as children the two nodes of the subtrees extracted from the min priority queue. The new root holds the sum of frequencies of the two subtrees.
- Put it back into the Priority Queue.

# Algorithm

**HUFFMAN($C$)**

$n = |C|$
Q = C
for $i$ = 1 to $n$-1
    allocate new node $z$
    z.left = x = Extract-Min(Q)
    z.right = y = Extract-Min(Q)
    z.freq = x.freq+y.freq
    Insert(Q,z)
return Extract-Min(Q) // return root of the Huffman Tree

**Complexity**

$$\Theta(n \log n)$$

# Algorithm

**HUFFMAN($C$)**

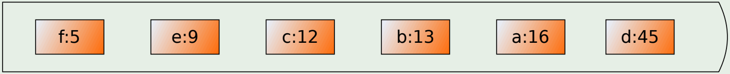
1. $n = |C|$
2. Q = C
3. for $i$ = 1 to $n$-1
4. allocate new node $z$
5. z.left = x = Extract-Min(Q)
6. z.right = y = Extract-Min(Q)
7. z.freq = x.freq+y.freq
8. Insert(Q,z)
9. return Extract-Min(Q) // return root of the Huffman Tree

**Complexity**

$$\Theta(n \log n)$$

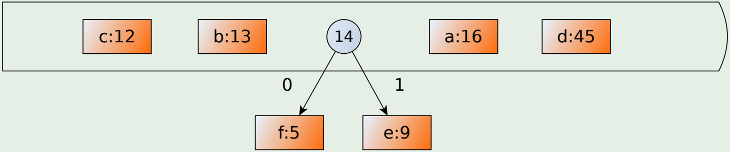
# Example

## The Process!!!



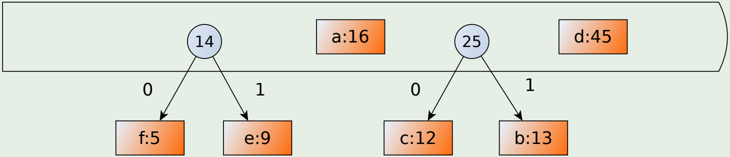
# Example

## The Process!!!



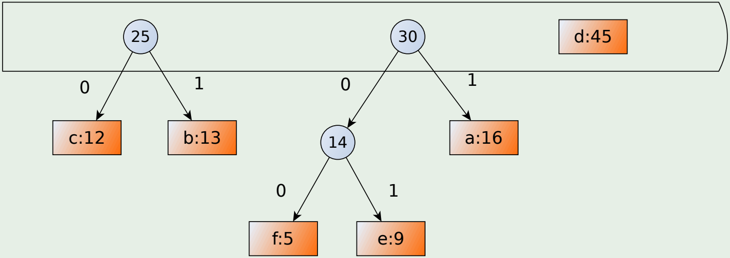
# Example

## The Process!!!



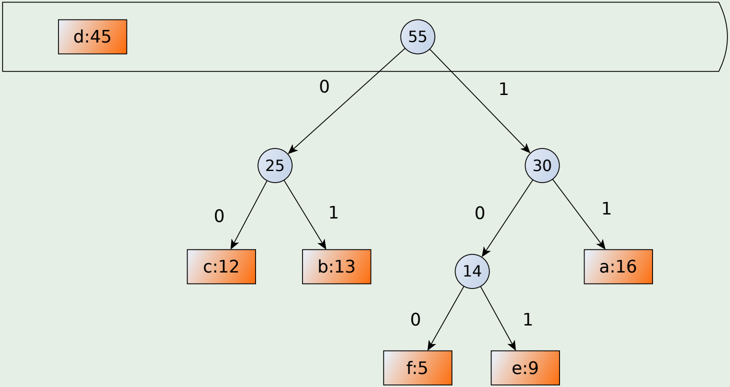
# Example

## The Process!!!



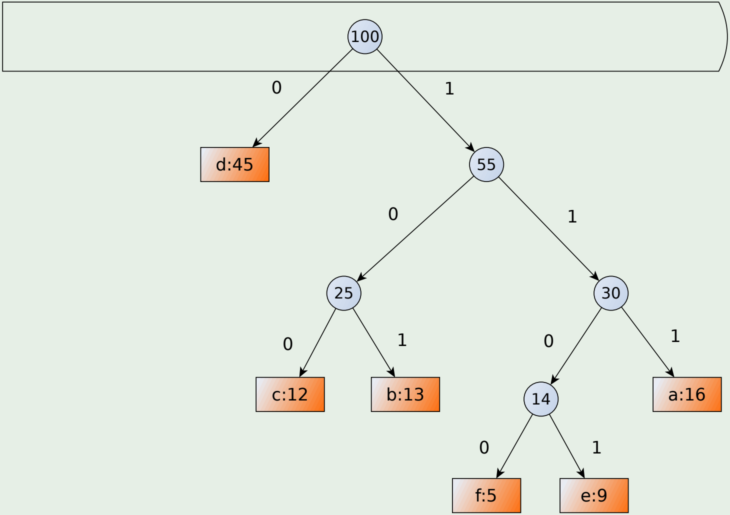
# Example

## The Process!!!



# Example

## The Process!!!



# Lemmas to sustain the claims

**Lemma 16.2**

Let $C$ be an alphabet in which each character $c \in C$ has frequency $c.freq$. Let $x$ and $y$ be two characters in $C$ having the lowest frequencies. Then there exists an optimal prefix code for $C$ in which the codewords for $x$ and $y$ have the same length and differ only in the last bit.

**Lemma 16.3**

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

# Lemmas to sustain the claims

## Lemma 16.2

Let $C$ be an alphabet in which each character $c \in C$ has frequency $c.freq$. Let $x$ and $y$ be two characters in $C$ having the lowest frequencies. Then there exists an optimal prefix code for $C$ in which the codewords for $x$ and $y$ have the same length and differ only in the last bit.

## Lemma 16.3

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

# Exercises

## From Cormen's book solve the following

- 16.2-1
- 16.2-4
- 16.2-5
- 16.2-7
- 16.1-3
- 16.3-3
- 16.3-5
- 16.3-7