Greedy Algorithms

NYIT CSCI-651

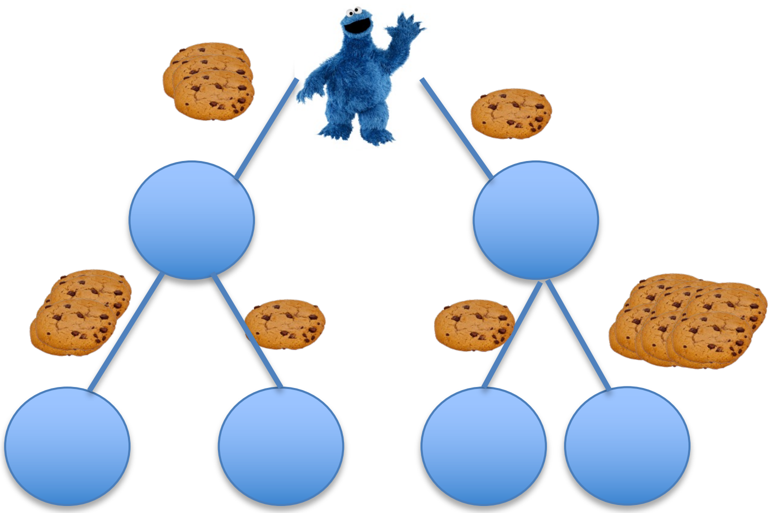
Greedy Algorithms

- A ***greedy algorithm*** always makes the choice that looks best at the moment.

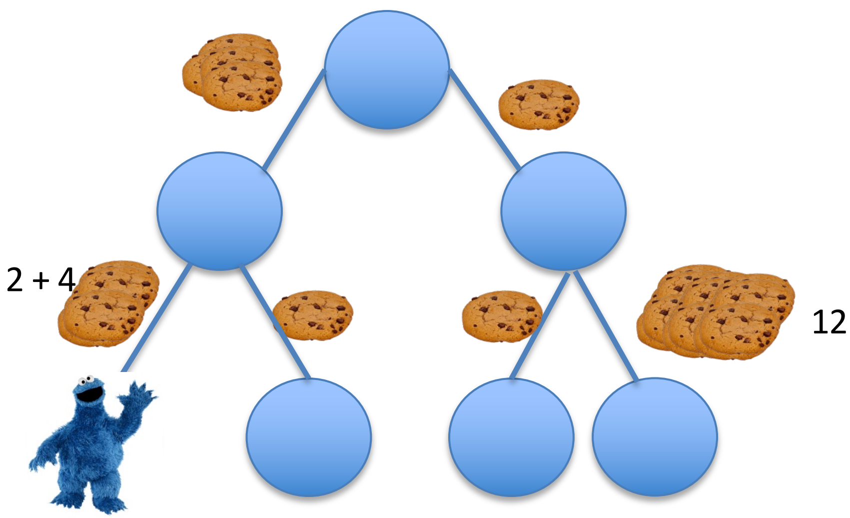
- **Local optimal** choices with the **hope** that the global solution would be optimal too.

- They **do not always** provide the **optimal** answer but for **some** problems, **they do**.

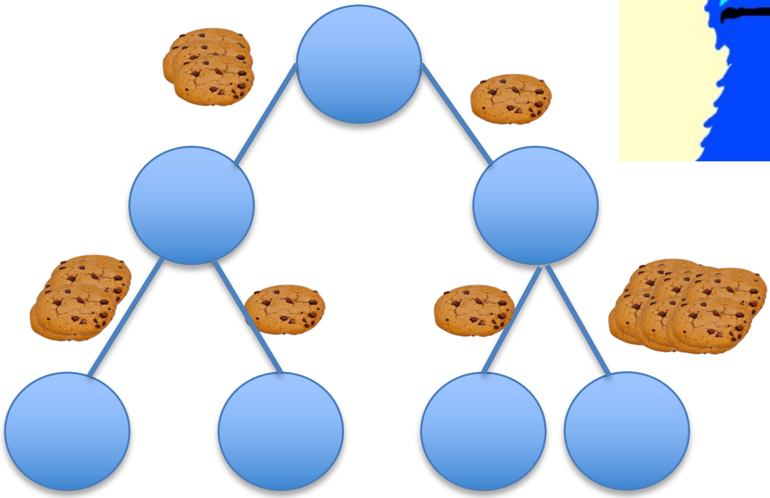
Greedy Algorithms



Greedy Algorithms



Greedy Algorithms



Optimal Substructure

- A problem has **optimal substructure** if an optimal solution can be constructed from optimal solutions of its subproblems. (i.e., optimal solution to the problem contains optimal solutions to subproblems)

- This property is a key ingredient of assessing the applicability of dynamic programming as well as greedy algorithms.

- Example: If the shortest route from Seattle to Los Angeles passes through Portland and then Sacramento, then the shortest route from Portland to Los Angeles must pass through Sacramento too.

Some points about greedy strategy

- In a greedy algorithm, **we make whatever choice seems best at the moment and then solve the subproblem that remains**.

- The choice made by a greedy algorithm may depend on choices so far, but it cannot depend on any future choices

- a greedy strategy usually progresses in a top-down fashion, making one greedy choice after another, reducing each given problem instance to a smaller one.

Elements of the greedy strategy

- Determine the optimal structure of the problem
- Develop a recursive structure
- Show that with the greedy choice, only one sub-problem is remained.
- Prove that it is always safe to use the greedy choice.
- Develop a recursive algorithm that implements the greedy choice.
- Convert the recursive algorithm to an iterative algorithm.

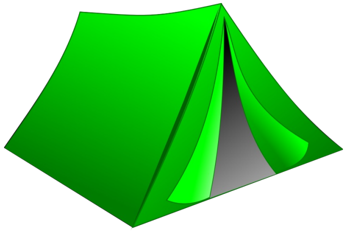
Example












...

Example

M kg

...

Example


200 gr
200 gr
400 gr
M kg
50 gr
2000 gr
300 gr
20 gr
1000 gr
250 gr
3000 gr
40 gr
800 gr
...

Example


200 gr -11
200 gr -10
400 gr - 9
M kg
50 gr - 5
2000 gr - 6
300 gr - 8
20 gr - 2
1000 gr - 7
250 gr -4
3000 gr - 1
40 gr - 12
800 gr - 3
...

Example

- Knapsack Problem

- You have a list of items
  - with value v_i and weight w_i .
- How do you select items so you can collect the highest value without going over the weight limit

0/1 Knapsack Problem

- a.k.a discrete knapsack problem
  - Take whole or nothing
  - Versus fractional (or continuous) knapsack

100 kg

Fractional Knapsack problem

- First Approach
  - Select the highest value items

Item	W	V
1	10	20
2	20	30
3	30	66
4	40	40
5	50	60

100 kg

Items 3, 5, and half of 4

66 + 60 + 20 = 146

Fractional Knapsack problem

- Second Approach
  - Select the lightest items

Item	W	V
1	10	20
2	20	30
3	30	66
4	40	40
5	50	60

100 kg

Items 1, 2, 3, and 4

20 + 30 + 66 + 40 = 156

Fractional Knapsack problem

- Third Approach
  - Select the item with highest V_i/W_i

Item	W	V
1	10	20
2	20	30
3	30	66
4	40	40
5	50	60

100 kg

Fractional Knapsack problem

- Third Approach
  - Select the item with highest V_i/W_i

Item	W	V	V/W
1	10	20	2.0
2	20	30	1.5
3	30	66	2.2
4	40	40	1.0
5	50	60	1.2

100 kg

Items 3, 1, 2, and 5

66 + 20 + 30 + 0.8(60) = 164

30 + 10 + 20 + 0.8 (50)

An activity-selection problem

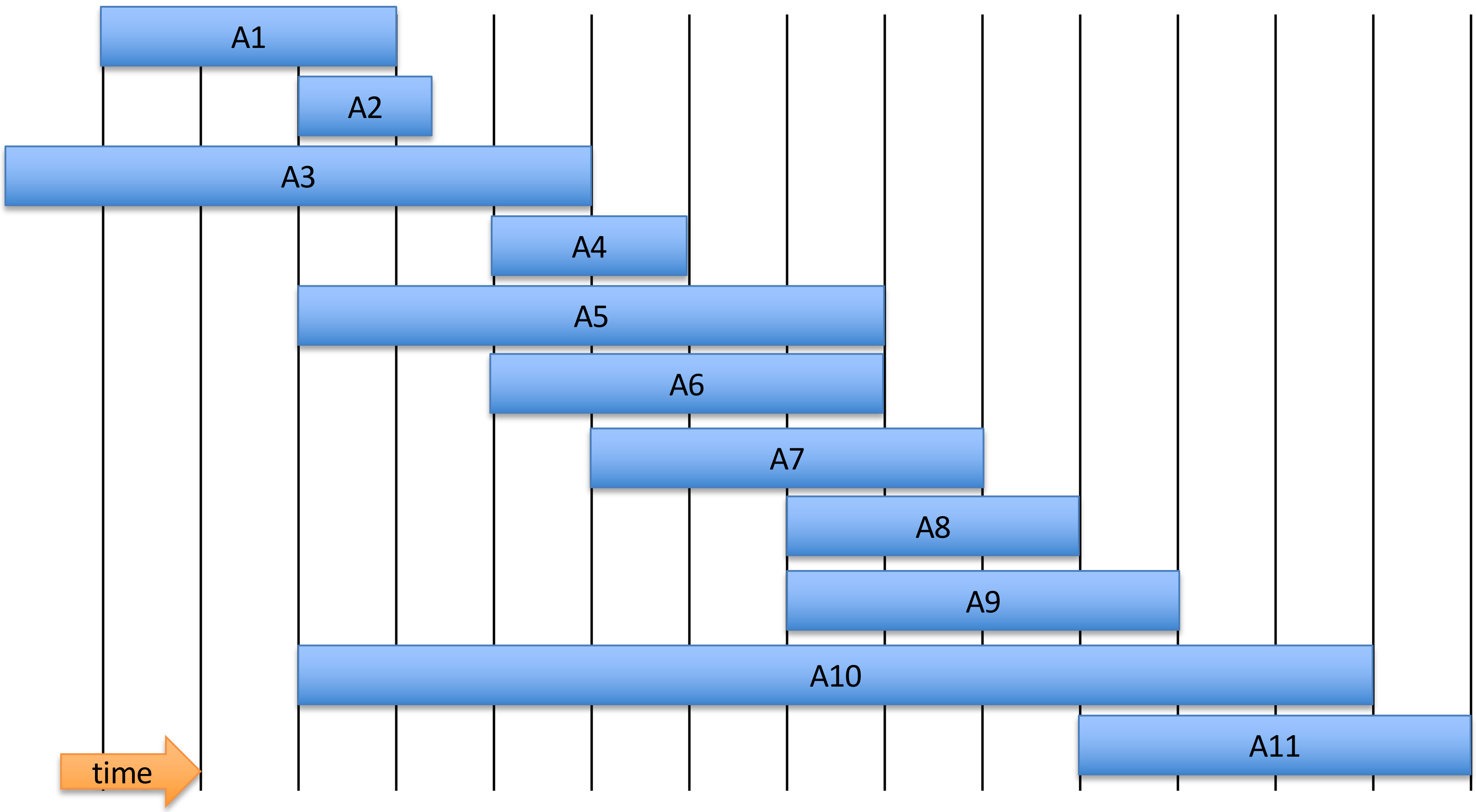
- We have a set of competing activities that require exclusive use of a common resource. Each activity has a start time s and finish time f.

- In the ***activity-selection problem***, we wish to select a maximum-size subset of mutually compatible activities.

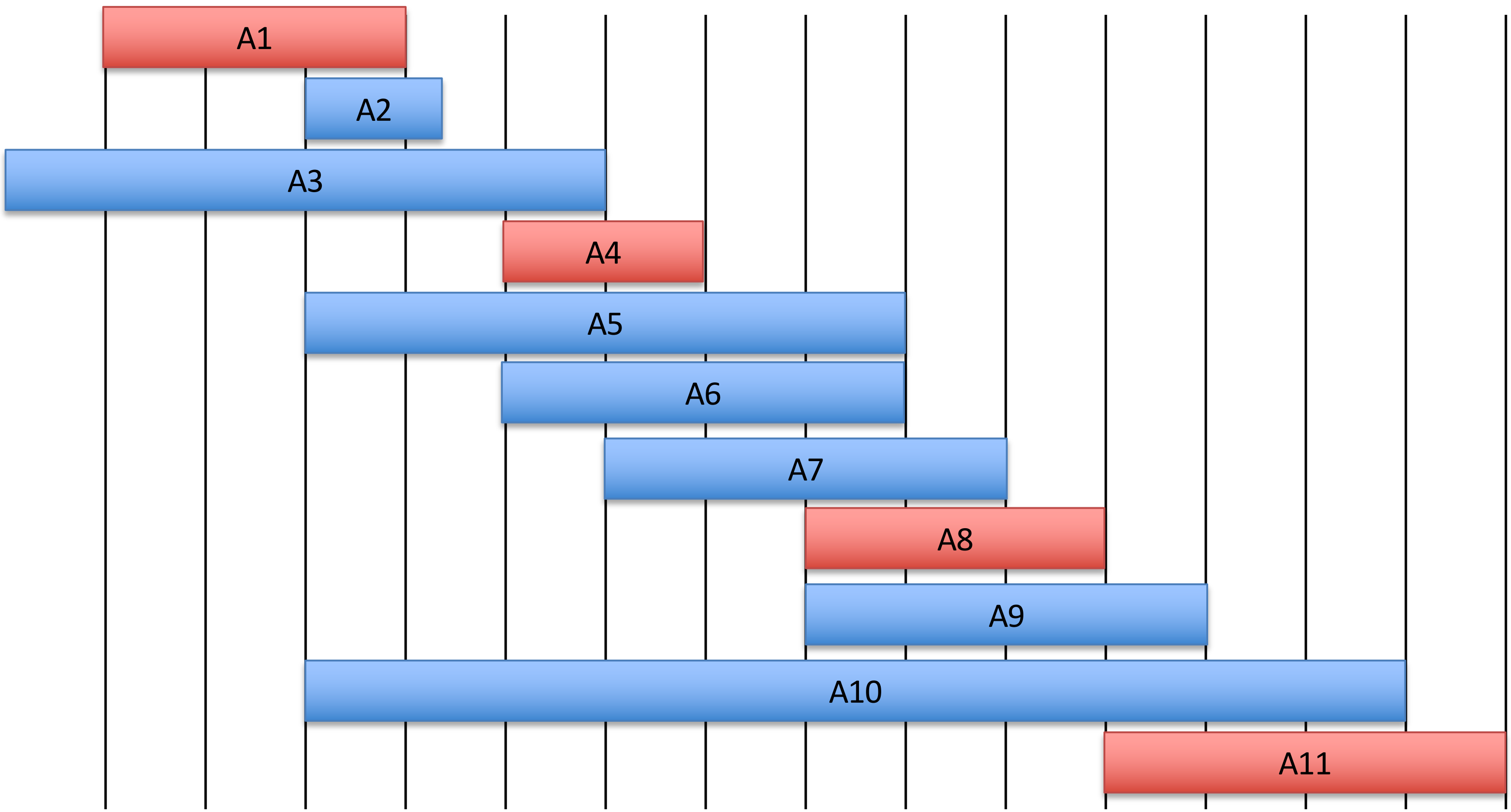
- Two activities a_i and b_j are compatible if two activities have no overlaps (a comes after b or otherwise)

- Activities a_i and a_j are compatible if $\mathbf{s_i >= f_j}$ **or** $\mathbf{s_j >= f_i}$

An activity-selection problem



An activity-selection problem



An activity-selection problem - Example

- Find a set of compatible activities for the following activities

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

(1, 4, 8, 11)
(1,4, 9, 11)
(3, 8, 11)
(2, 4, 8, 11)

An activity-selection problem - Example

- Select a **maximum-size subset** of mutually compatible activities.

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

An activity-selection problem

- What is the greedy choice here?

- What is our intuition?
  - We should <u>select an activity that leaves the resource available</u> for as many activities as possible.

An activity-selection problem

- What does that mean?
  - The activity that finishes first
  - Since that would leave the resource available for as many activities as possible.

An activity-selection problem

- Let's say that the activities are <u>sorted</u> monotonically <u>based on the finish time</u>.

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

What if it is not sorted? Sort it with O(nlogn).

An activity-selection problem

- Let's say that the activities are sorted monotonically.

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

- What is the first chosen activity?
  - The greedy choice is activity a_1
- We have only one remaining subproblem to solve:
  - finding activities that start after a_1 finishes.

An activity-selection problem

- Let $S_k = \{a_i \in S: s_i \geq f_k\}$ be the set of activities that start after activity a_k finishes

- If we make a greedy choice of a_1 as the first activity, then S_1 remains the only sub problem to solve
- When do we have an optimal solution?
  - When the solution of S_1 is also optimal

An activity-selection problem

- Optimal substructure tells us that if a_1 is in the optimal solution, then an **optimal solution to the original problem** consists of
  - activity a_1 and
  - all the activities in an optimal solution to the sub problem S_1

Question

- Is our intuition correct?

- Is the greedy choice—in which we choose the first activity to finish—always part of some optimal solution?
- The following theorem shows that it is.

Theorem 16.1

Consider any nonempty subproblem S_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.

The activity with the <u>earliest finish time</u> is always included in the solution.

Solution

1. We can repeatedly choose the activity that finishes first

2. Keep only the compatible activities

3. Repeat until no activity remains

A recursive greedy algorithm

An array of start times

RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)

$m = k + 1$
while $m \leq n$ and $s[m] < f[k]$ // find the first activity in S_k to finish
    $m = m + 1$
if $m \leq n$
    return $\{a_m\} \cup$ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
else return $\emptyset$

We first start with (s,f,0,n)
In order to start, we add the fictitious activity a_0 with $f_0 = 0$, so that sub problem S_0 is the entire set of activities S.
The initial call, which solves the entire problem, is RECURSIVE-ACTIVITY-SELECTOR (s,f,0,n)

A recursive greedy algorithm

An array of finish times

RECURSIVE-ACTIVITY-SELECTOR (s, f, k, n)

```
m = k + 1
while m ≤ n and s[m] < f[k]        // find the first activity in S_k to finish
    m = m + 1
if m ≤ n
    return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
else return ∅
```

We first start with (s,f,0,n)
In order to start, we add the fictitious activity a_0 with $f_0 = 0$, so that sub problem S_0 is the entire set of activities S.
The initial call, which solves the entire problem, is RECURSIVE-ACTIVITY-SELECTOR (s,f,0,n)

A recursive greedy algorithm

Number of activities

```
RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)
m = k + 1
while m ≤ n and s[m] < f[k]        // find the first activity in S_k to finish
    m = m + 1
if m ≤ n
    return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
else return Ø
```

We first start with (s,f,0,n)
In order to start, we add the fictitious activity a_0 with $f_0 = 0$, so that sub problem S_0 is the entire set of activities S.
The initial call, which solves the entire problem, is RECURSIVE-ACTIVITY-SELECTOR (s,f,0,n)

A recursive greedy algorithm

Index of the sub-problem

RECURSIVE-ACTIVITY-SELECTOR (s, f, k, n)

```
m = k + 1
while m ≤ n and s[m] < f[k]        // find the first activity in S_k to finish
    m = m + 1
if m ≤ n
    return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
else return Ø
```

We first start with (s,f,0,n)
In order to start, we add the fictitious activity a_0 with $f_0 = 0$, so that sub problem S_0 is the entire set of activities S.
The initial call, which solves the entire problem, is RECURSIVE-ACTIVITY-SELECTOR (s,f,0,n)

RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)

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

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

Assuming that the activities have already been sorted by finish times, the running time of the call RECURSIVE-ACTIVITY-SELECTOR. O(n)

Over all recursive calls, each activity is examined exactly once in the **while** loop test of line 2.

Analysis of Running Time

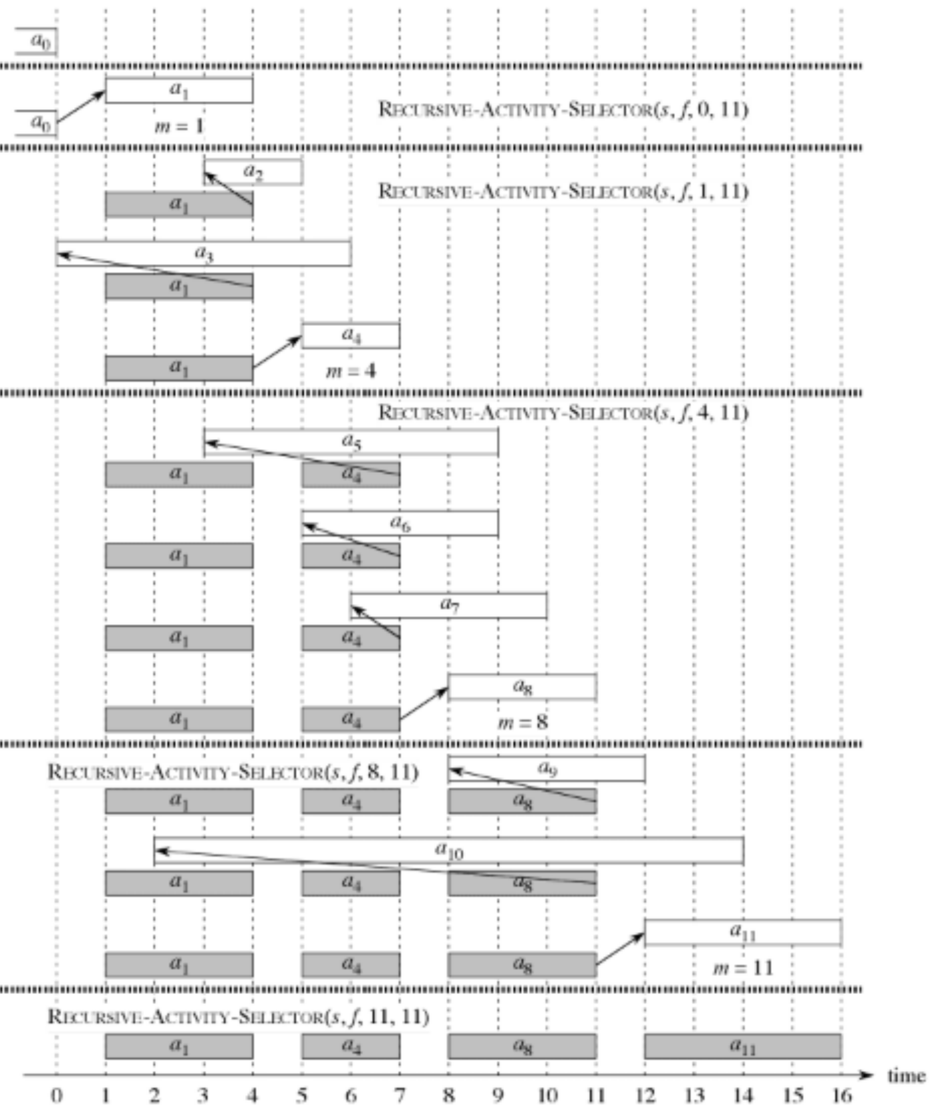
- Assuming that the activities have already been sorted by finish times, the running time of the call

  RECURSIVE-ACTIVITY-SELECTOR (s,f,0,n) is **Θ(n),** which we can see as follows:

  – Over all recursive calls, each activity is examined exactly once in the while loop test of line 2

A recursive greedy algorithm

k	s_k	f_k
0	–	0
1	1	4
2	3	5
3	0	6
4	5	7
5	3	9
6	5	9
7	6	10
8	8	11
9	8	12
10	2	14
11	12	16



Non recursive algorithm

- Find a non recursive algorithm

In **traditional recursion**, the typical model is that you perform your recursive calls first, and then you take the return value of the recursive call and calculate the result. In this manner, you don't get the result of your calculation until you have returned from every recursive call.

In **tail recursion**, you perform your calculations first, and then you execute the recursive call, passing the results of your current step to the next recursive step. This results in the last statement being in the form of (return (recursive-function params)). **Basically, the return value of any given recursive step is the same as the return value of the next recursive call**.

Non recursive algorithm

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

Like the recursive version, GREEDY-ACTIVITY-SELECTOR schedules a set of n activities in O(n) time, assuming that the activities were already sorted initially by their finish times.

Data Compression Problem

- Given a text that uses 32 symbols (26 different letters, space, and some punctuation characters), how can we encode this text in bits?
- We need to assign a binary code to each letter.

A:00
B:01
C:10
Y:11

BABY

01000111

How many bits? Can we do better?

Data Compression

- Given a text that uses 32 symbols (26 different letters, space, and some punctuation characters), how can we encode this text in bits?
- We need to assign a binary code to each letter.

A:0
B:01
C:10
Y:1

BABY: 010011

010011:AYAAYY

is this unique?

Data Compression

- How to differentiate between letters?
  - One possibility is to use a separation symbol (not very efficient, you need extra bits for each letter)

BABY: 01,0,01,1

  - Use codes in such a way that there is no ambiguity:

by ensuring that no code is a prefix of another one.

Prefix codes are desirable because they simplify decoding. Since no codeword is a prefix of any other, the codeword that begins an encoded file is unambiguous.

Data Compression

- A prefix code for a set <u>S</u> is a function <u>c</u> that maps each <u>x∈ S</u> to <u>1</u>s and <u>0</u>s in such a way that <u>for x,y∈S</u>, <u>x≠ y</u>, <u>c(x)</u> is not a prefix of <u>c(y)</u>.

c(A) = 11
c(O) = 01
c(E) = 001
c(C) = 10
c(D) = 000

What is 1001000001?

There is no ambiguity!

Data Compression

- It is desired that letters with higher frequency have smaller length.

Data Compression

- The average bits per letter of a prefix code c is the sum over all symbols of its frequency times the number of bits of its encoding

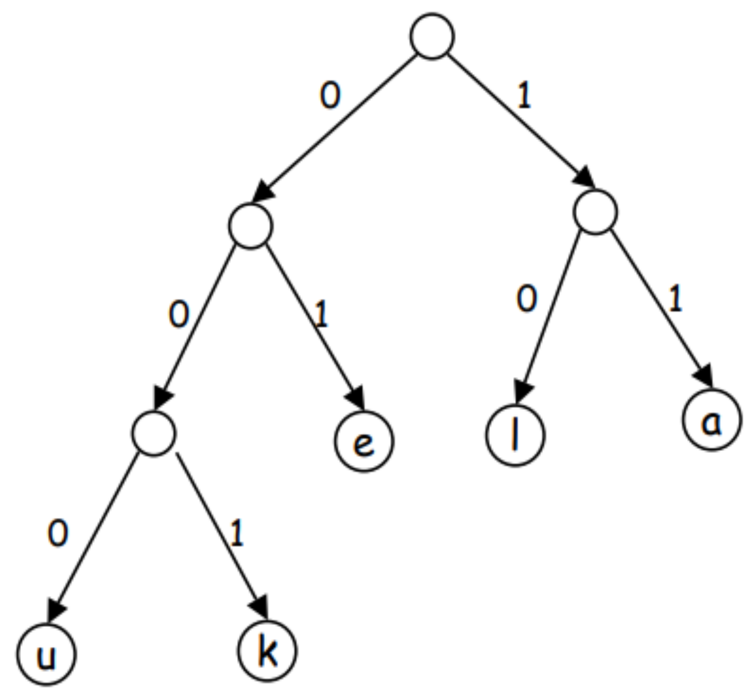
$$ABL(c) = \sum_{x \in S} f_x \cdot |c(x)|$$

- We would like to find a prefix code that has the lowest possible average bits per letter.

Binary Tree

The decoding process needs a convenient representation for the prefix code so that we can easily pick off the initial codeword. A binary tree whose **leaves are the given characters** provides one such representation.

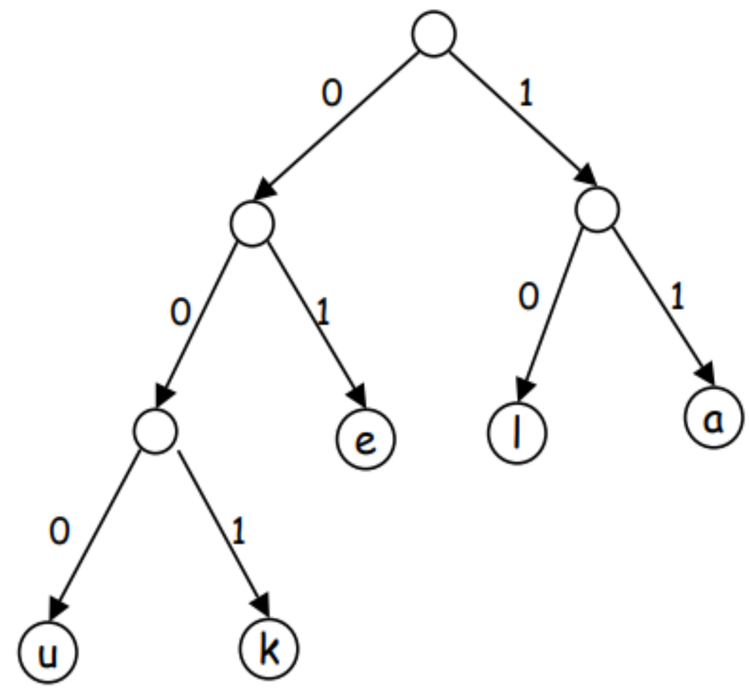
c(a) = 11
c(e) = 01
c(k) = 001
c(l) = 10
c(u) = 000



Binary Tree

We interpret the binary codeword for a character as **the simple path from the root to that character,** where **0** means "go to the **left** child" and **1** means "go to the **right** child."

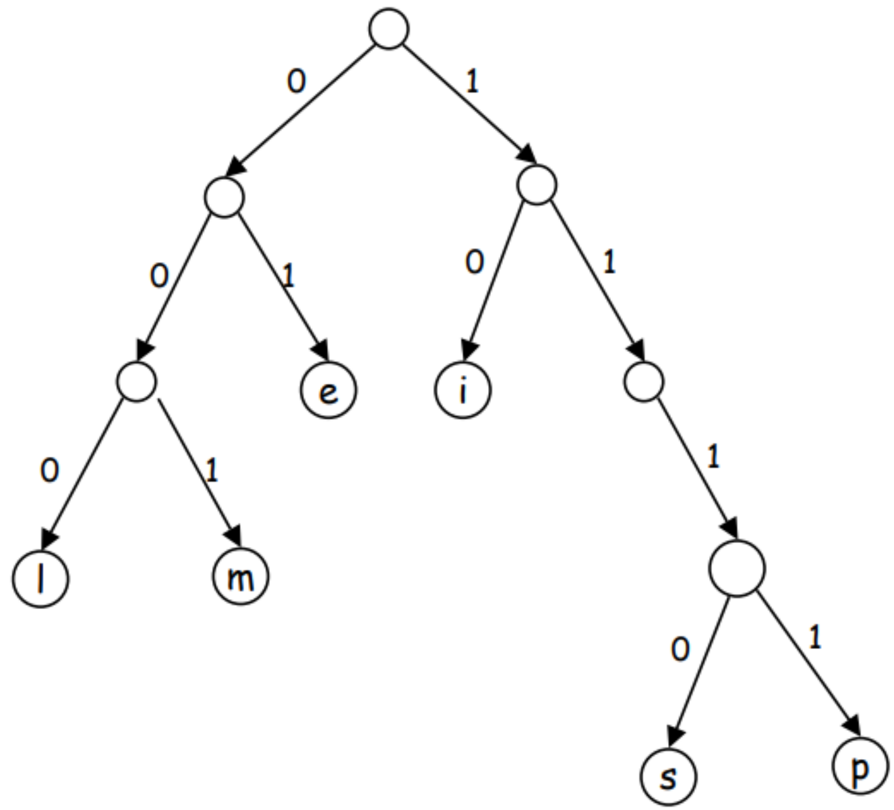
c(a) = 11
c(e) = 01
c(k) = 001
c(l) = 10
c(u) = 000



Only leaves have labels

What is 1110100011111100001 ?
simple

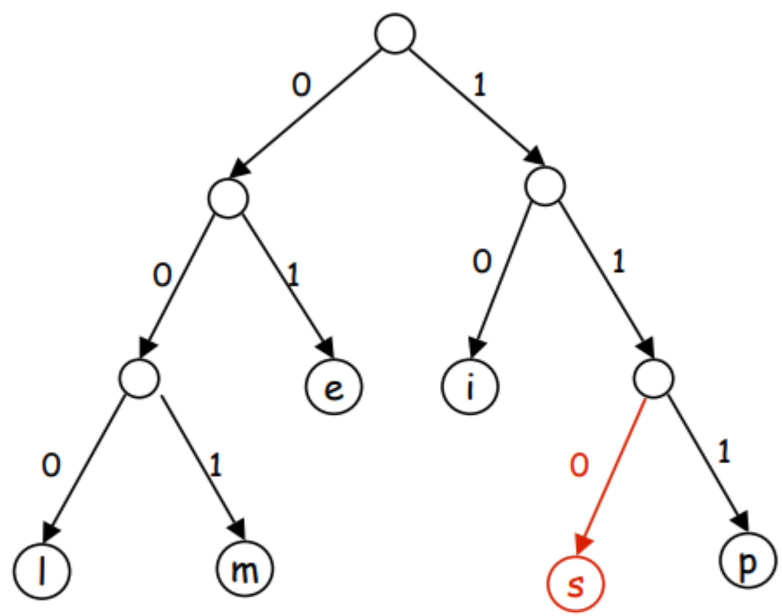
How can this prefix code
be made more efficient?



What is 11101000111110000 1 ?
simple

How can this prefix code be made more efficient?

An optimal code for a file is always represented by a **full binary tree**, in which every non-leaf node has two children



This tree is full: Each node that is not a leaf has two children

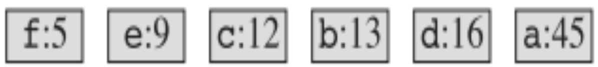
Binary tree

- The binary tree corresponding to the optimal prefix code is full. Otherwise we are wasting bits

- Where in the tree of an optimal prefix code should letters be placed with a high frequency?
  – High frequency letters should be placed at leaves whose the depth is the lowest.

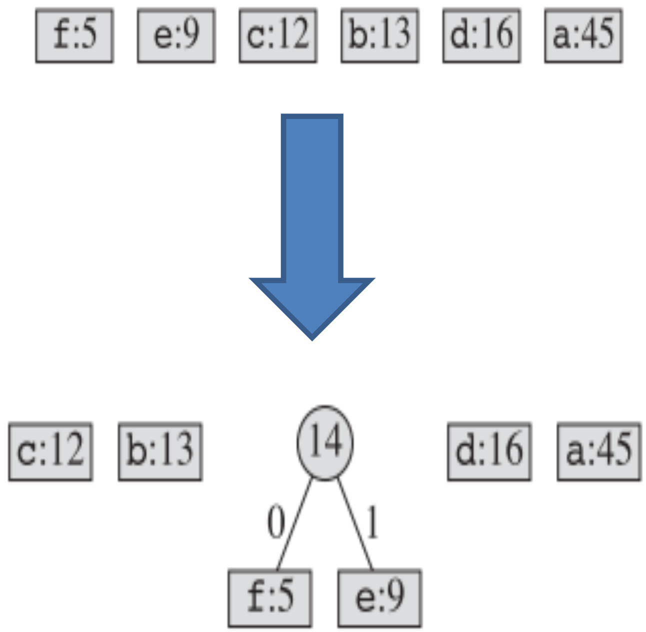
Huffman Coding

- How to make an efficient full tree for encoding the letters?



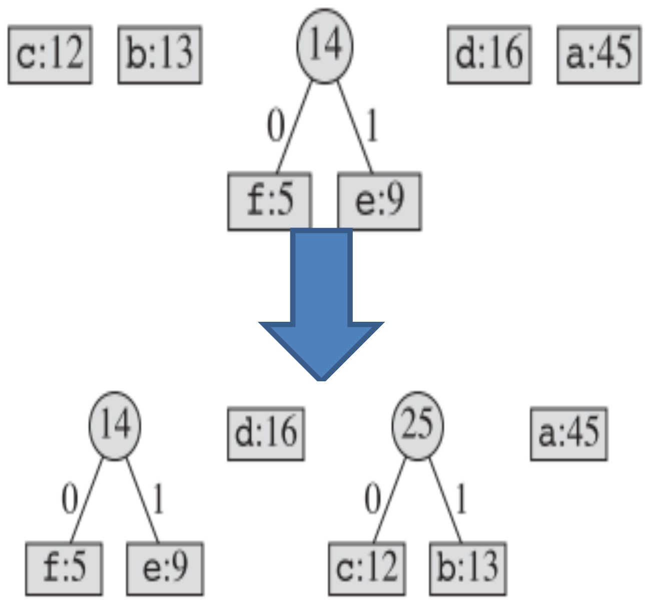
Huffman Coding

- Recursively <u>combine two elements with lowest frequency</u>



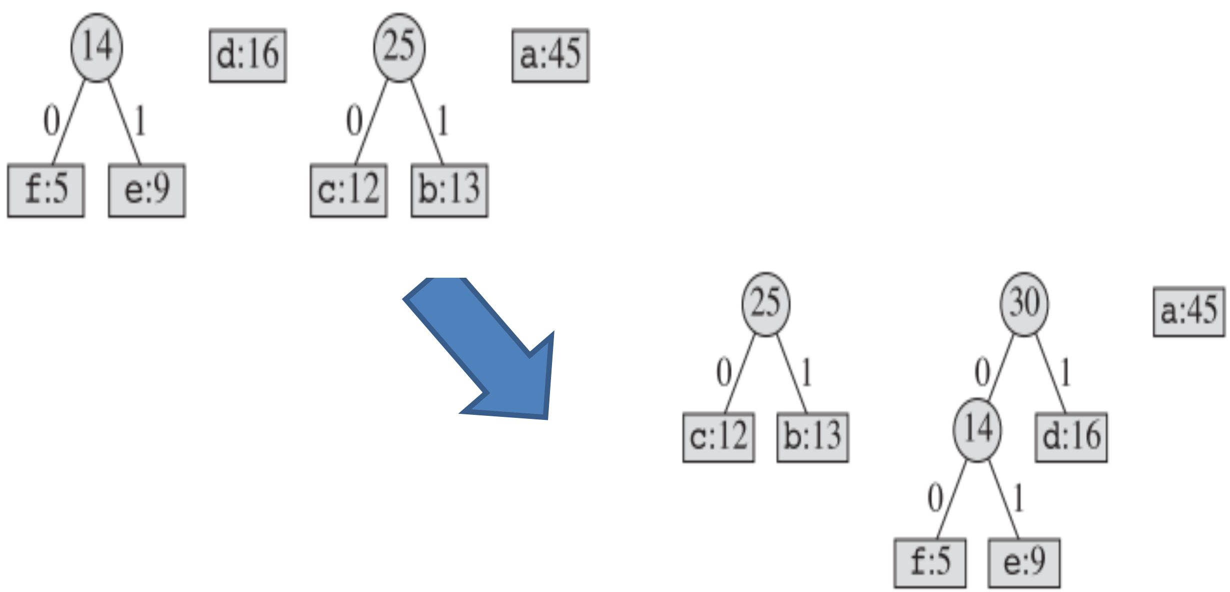
Huffman Coding

- Recursively combine two elements with lowest frequency



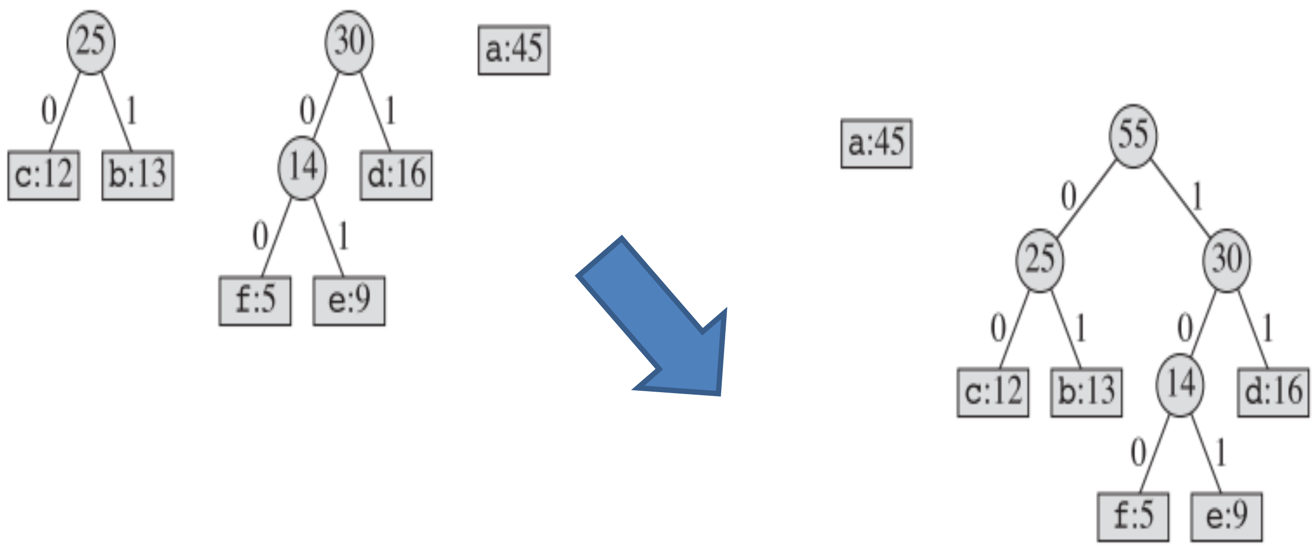
Huffman Coding

- Recursively combine two elements with lowest frequency



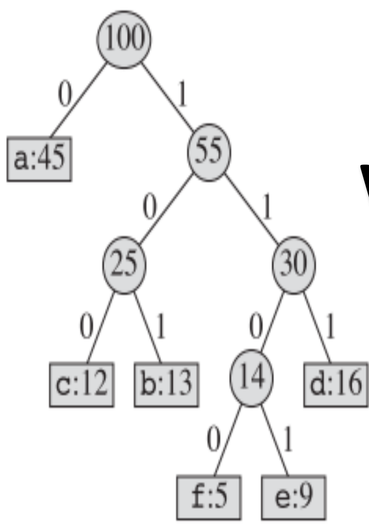
Huffman Coding

- Recursively combine two elements with lowest frequency



Huffman Coding

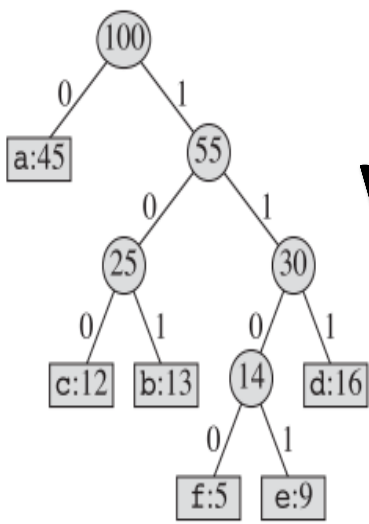
- Recursively combine two elements with lowest frequency



What is 0111100?

Huffman Coding

- Recursively combine two elements with lowest frequency



What is 0111100?

adc

Huffman Coding - Algorithm

we assume that Q is implemented as a binary min-heap

For a set C of n characters, we can initialize Q in line 2 in O(n) time using the BUILD-MIN-HEAP procedure.

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

O(log n)

O(nlog n)

The **for** loop in lines 3–8 exactly n - 1 times, and each heap operation requires time O(lg n), thus the loop contributes O(n lg n).

Exercise

Frequency	Value
5	a
7	b
10	c
13	d
20	e
45	f

How many bits are saved compared to fixed-length coding?