CS473-Algorithms I

Lecture 11

Greedy Algorithms

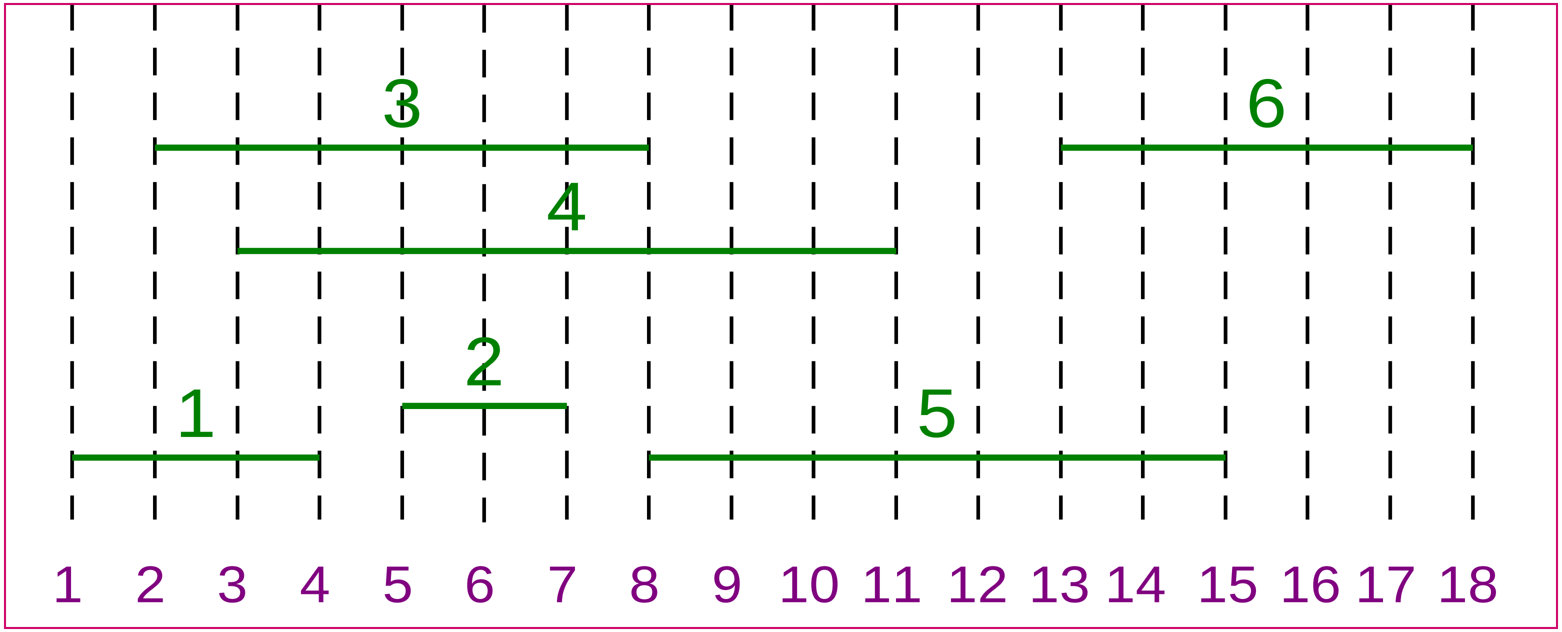
Activity Selection Problem

- Input: a set S ={1, 2, …, n} of n activities
  - s_i=Start time of activity i,
  - f_i= Finish time of activity i

  Activity i takes place in $[s_i, f_i)$
- Aim: Find max-size subset A of mutually *compatible* activities
  - Max number of activities, not max time spent in activities
  - Activities i and j are compatible if intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap, i.e., either $s_i \geq f_j$ or $s_j \geq f_i$

Activity Selection Problem: An Example

S={[1, 4), [5, 7), [2, 8), [3, 11), [8, 15), [13, 18)}



Optimal Substructure

Theorem: Let k be the activity with the earliest finish time in an optimal soln $A \subseteq S$ then

$A - \{k\}$ is an optimal solution to subproblem

$S_k' = \{i \in S: s_i \geq f_k\}$

Proof (by contradiction):

- ▹ Let B' be an optimal solution to S_k' and $|B'| > |A - \{k\}| = |A| - 1$
- ▹ Then, $B = B' \cup \{k\}$ is compatible and

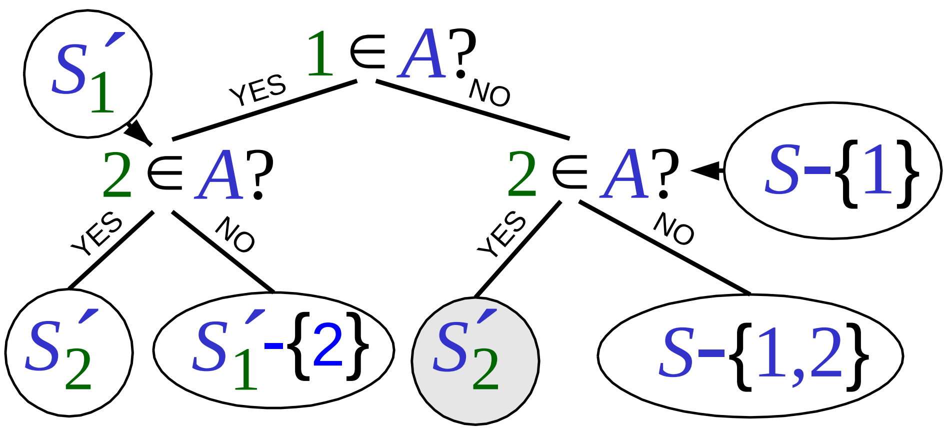
$|B| = |B'| + 1 > |A|$

Contradiction to the optimality of A Q.E.D.

Repeated Subproblems

- Consider recursive algorithm that tries all possible compatible subsets
- Notice repeated subproblems (e.g., S_2')

(let $f_1 \leq \ldots \leq f_n$)



Greedy Choice Property

- Repeated subproblems and optimal substructure properties hold in activity selection problem
- Dynamic programming?

  Memoize?

  Yes, but…
- Greed choice property: a sequence of locally optimal (greedy) choices $\Rightarrow$ an optimal solution
- Assume (without loss of generality) $f_1 \le f_2 \le \ldots \le f_n$
  - If not sort activities according to their finish times in non-decreasing order

Greedy Choice Property in Activity Selection

Theorem: There exists an optimal solution $A \subseteq S$ such that $1 \in A$ (Remember $f_1 \leq f_2 \leq \ldots \leq f_n$)

Proof: Let $A = \{k, \ell, m, \ldots\}$ be an optimal solution such that $f_k \leq f_\ell \leq f_m \leq \ldots$

- ▷ If $k = 1$ then schedule A begins with the greedy choice
- ▷ If $k > 1$ then show that $\exists$ another optimal soln that begins with the greedy choice 1
  - ▷ Let $B = A - \{k\} \cup \{1\}$, since $f_1 \leq f_k$ activity 1 is compatible with $A - \{k\}$; B is compatible
  - ▷ $|B| = |A| - 1 + 1 = |A|$
  - ▷ Hence B is optimal Q.E.D.

Activity Selection Problem

j: specifies the index of most recent activity added to A

$f_j = \text{Max}\{f_k : k \in A\}$, max finish time of any activity in A; because activities are processed in non-decreasing order of finish times

Thus, "$s_i \geq f_j$ "checks the compatibility of i to current A

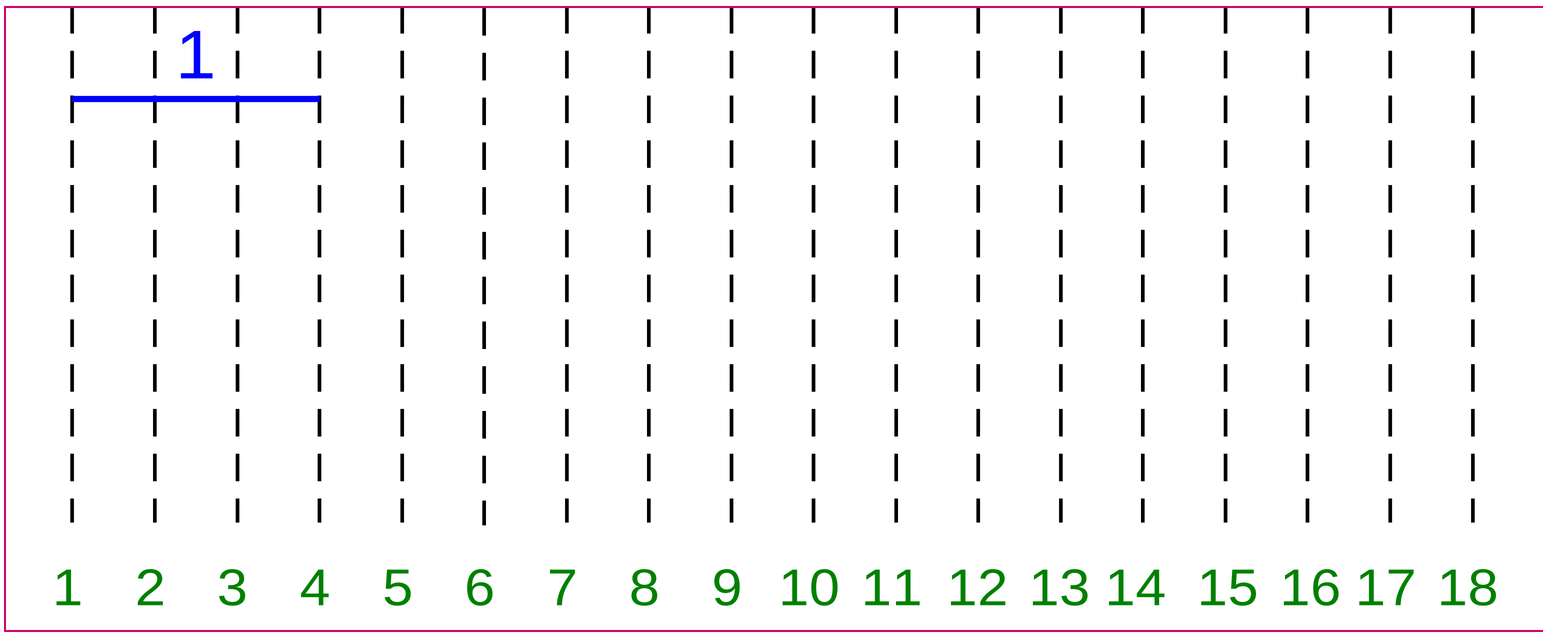
Running time: $\Theta(n)$ assuming that the activities were already sorted

GAS(s, f , n)
    $A \leftarrow \{1\}$
    $j \leftarrow 1$
    for $i \leftarrow 2$ **to** n **do**
        if $s_i \geq f_j$ **then**
            $A \leftarrow A \cup \{i\}$
            $j \leftarrow i$
    return A

Activity Selection Problem: An Example

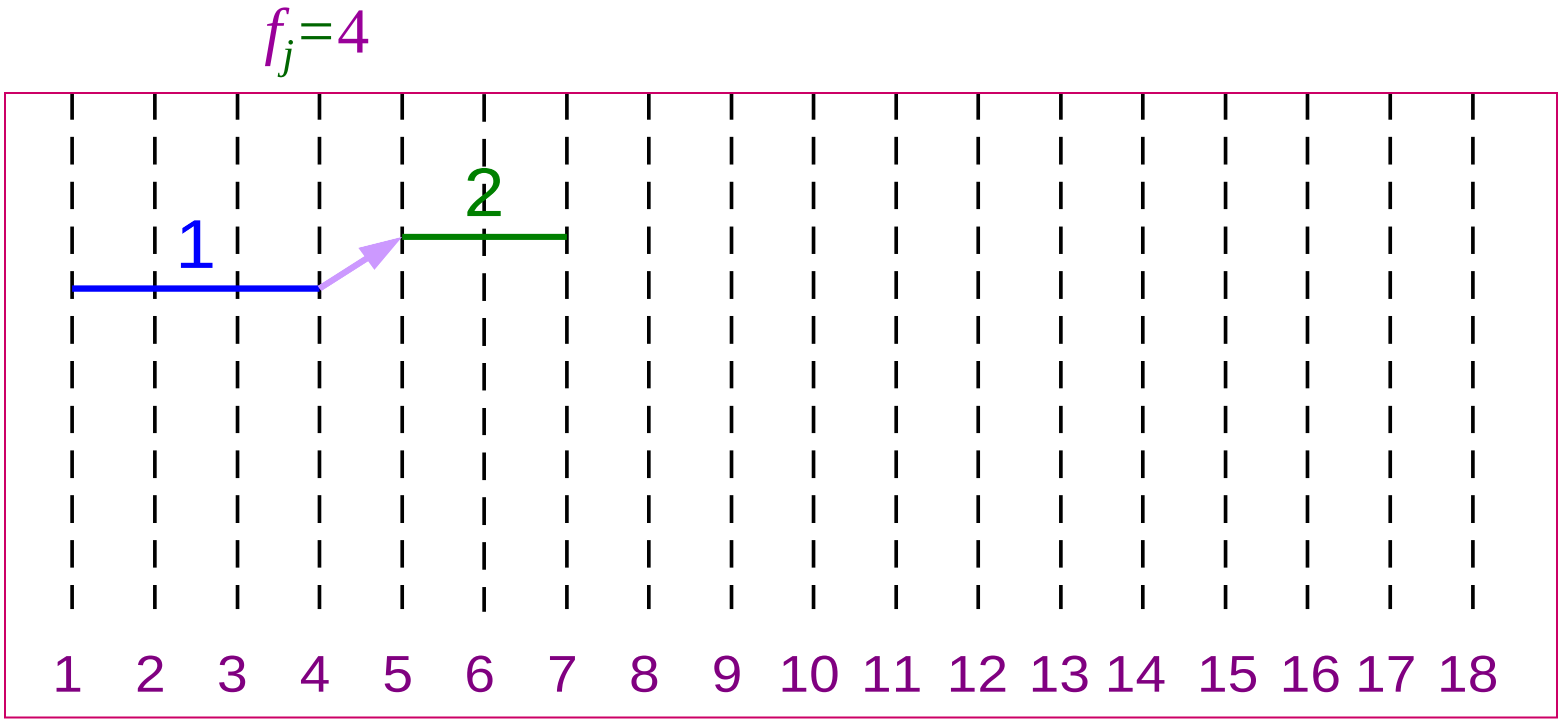
$S=\{[1, 4), [5, 7), [2, 8), [3, 11), [8, 15), [13, 18)\}$

$f_j=0$



Activity Selection Problem: An Example

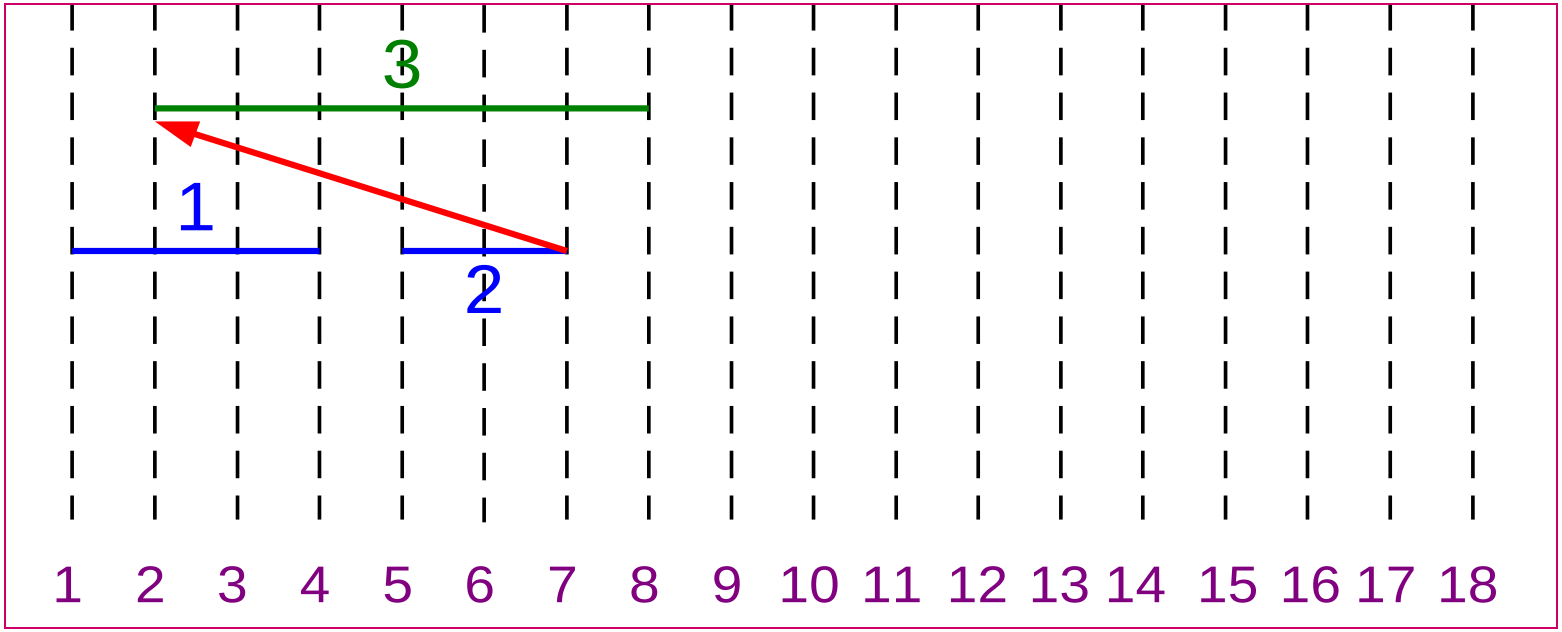
S={[1, 4), [5, 7), [2, 8), [3, 11), [8, 15), [13, 18)}



Activity Selection Problem: An Example

S={[1, 4), [5, 7), [2, 8), [3, 11), [8, 15), [13, 18)}

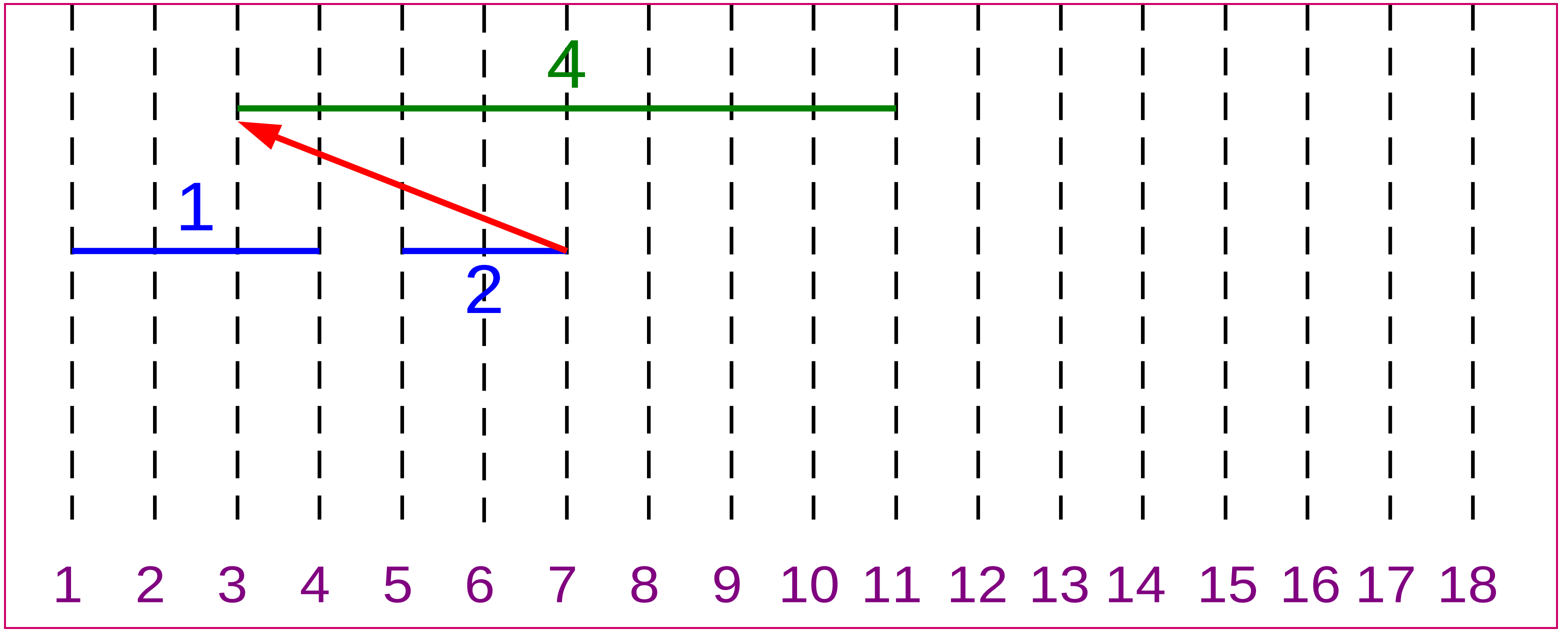
f_j=7



Activity Selection Problem: An Example

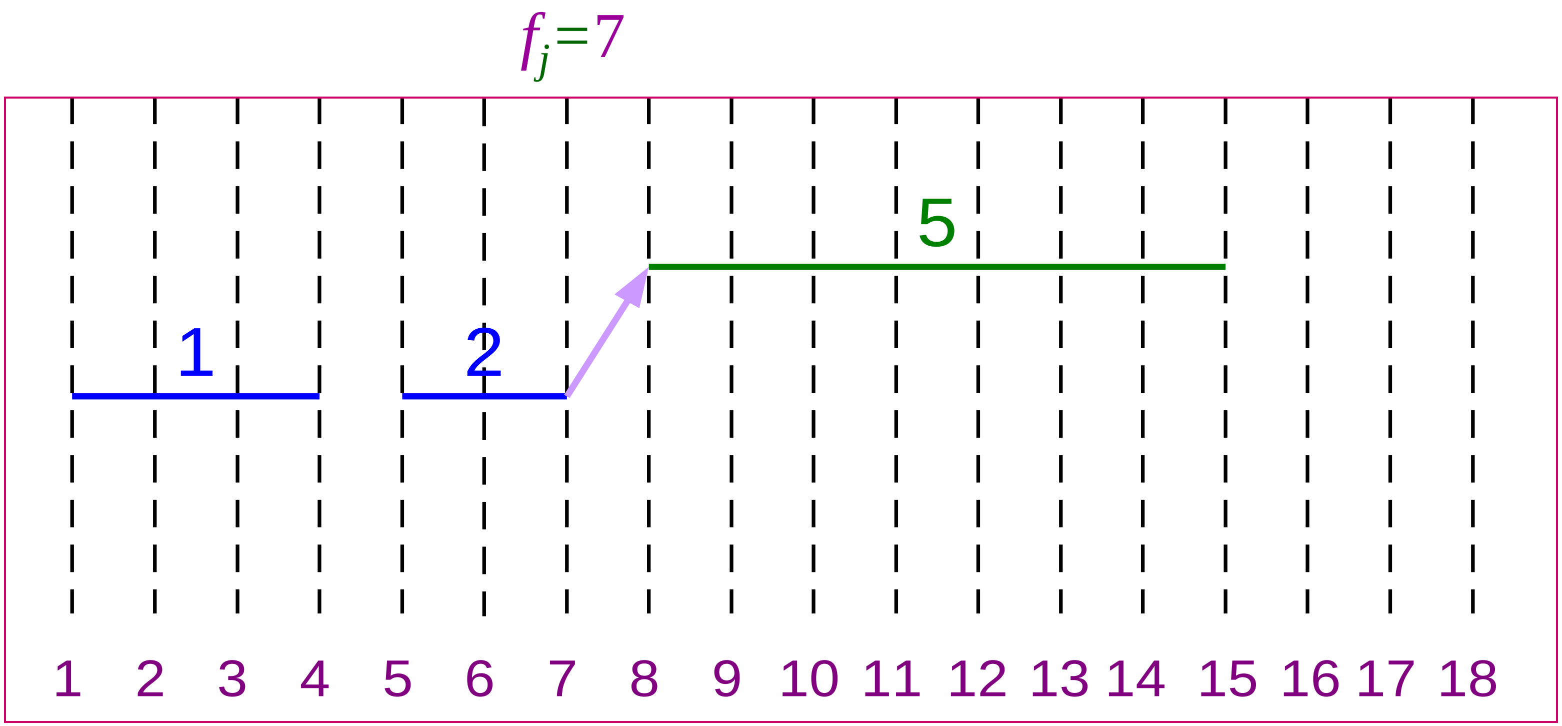
S={[1, 4), [5, 7), [2, 8), [3, 11), [8, 15), [13, 18)}

f_j=7



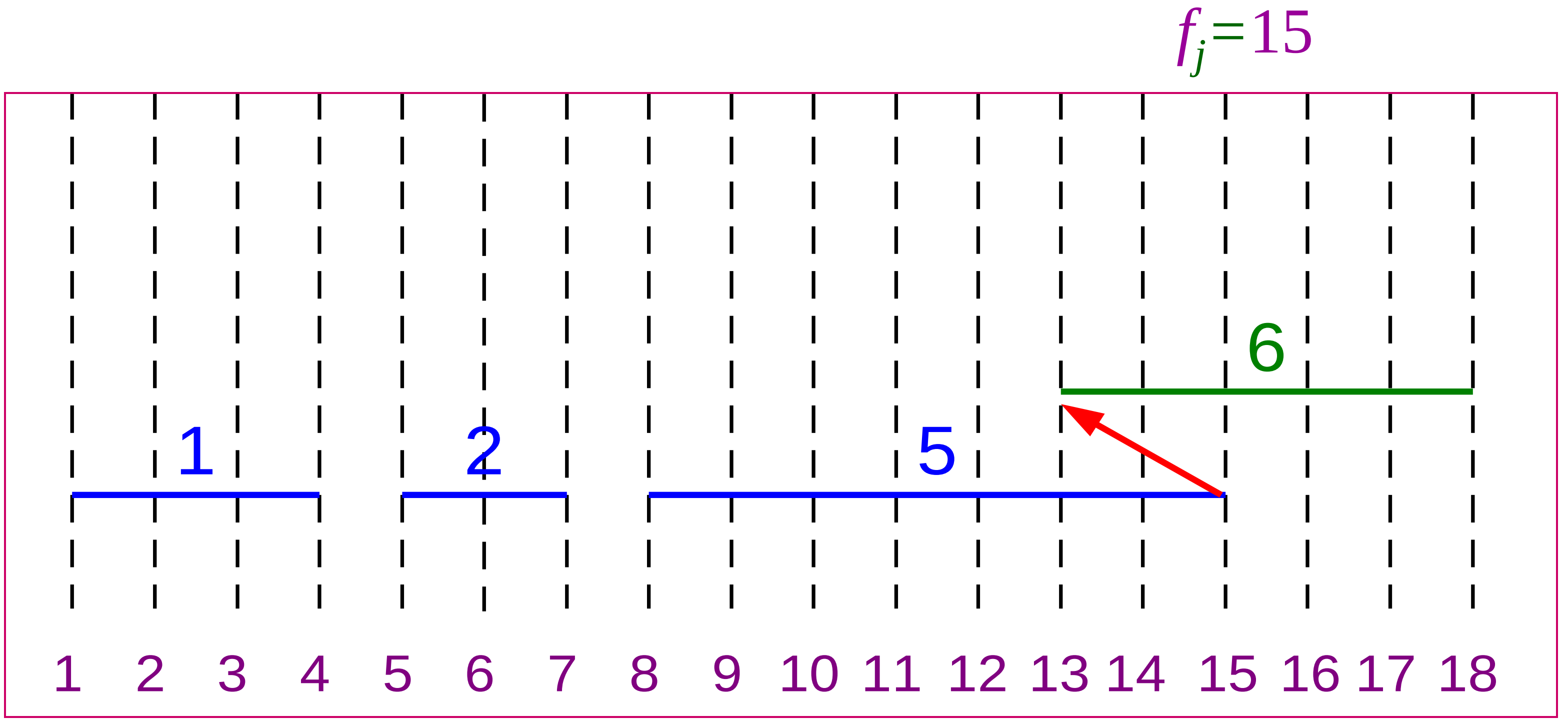
Activity Selection Problem: An Example

S={[1, 4), [5, 7), [2, 8), [3, 11), [8, 15), [13, 18)}



Activity Selection Problem: An Example

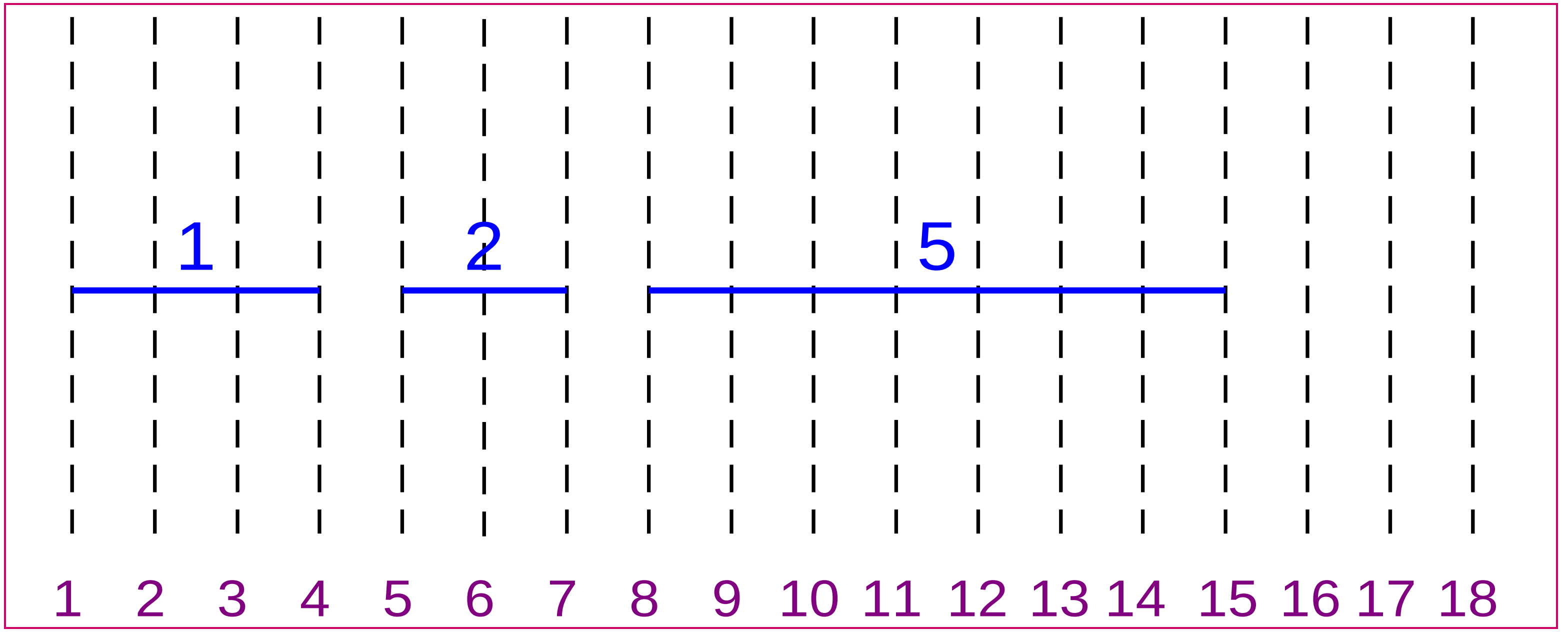
S={[1, 4), [5, 7), [2, 8), [3, 11), [8, 15), [13, 18)}



Activity Selection Problem: An Example

S={[1, 4), [5, 7), [2, 8), [3, 11), [8, 15), [13, 18)}

A={1, 2, 5}



Greedy vs Dynamic Programming

- Optimal substructure property exploited by both Greedy and DP strategies
- Greedy Choice Property: A sequence of locally optimal choices $\Rightarrow$ an optimal solution
  - We make the choice that seems best at the moment
  - Then solve the subproblem arising after the choice is made
- DP: We also make a choice/decision at each step, but the choice may depend on the optimal solutions to subproblems
- Greedy: The choice may depend on the choices made so far, but it cannot depend on any future choices or on the solutions to subproblems

Greedy vs Dynamic Programming

- DP is a bottom-up strategy
- Greedy is a top-down strategy
  - each greedy choice in the sequence iteratively reduces each problem to a similar but smaller problem

Proof of Correctness of Greedy Algorithms

- Examine a globally optimal solution
- Show that this soln can be modified so that
  1) A greedy choice is made as the first step
  2) This choice reduces the problem to a similar but smaller problem
- Apply induction to show that a greedy choice can be used at every step
- Showing (2) reduces the proof of correctness to proving that the problem exhibits optimal substructure property

Elements of Greedy Strategy

- How can you judge whether
- A greedy algorithm will solve a particular optimization problem?

Two key ingredients

- Greedy choice property
- Optimal substructure property

Key Ingredients of Greedy Strategy

- Greedy Choice Property: A globally optimal solution can be arrived at by making locally optimal (greedy) choices
- In DP,we make a choice at each step but the choice may depend on the solutions to subproblems
- In Greedy Algorithms, we make the choice that seems best at that moment then solve the subproblems arising after the choice is made
  - The choice may depend on choices so far, but it cannot depend on any future choice or on the solutions to subproblems
- DP solves the problem bottom-up
- Greedy usually progresses in a top-down fashion by making one greedy choice after another reducing each given problem instance to a smaller one

Key Ingredients: Greedy Choice Property

- We must prove that a greedy choice at each step yields a globally optimal solution
- The proof examines a globally optimal solution
- Shows that the soln can be modified so that a greedy choice made as the first step reduces the problem to a similar but smaller subproblem
- Then induction is applied to show that a greedy choice can be used at each step
- Hence, this induction proof reduces the proof of correctness to demonstrating that an optimal solution must exhibit optimal substructure property

Key Ingredients: Optimal Substructure

- A problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to subproblems

Example: Activity selection problem S

If an optimal solution A to S begins with activity 1 then the set of activities

$$A' = A - \{1\}$$

is an optimal solution to the activity selection problem

$$S' = \{ i \in S : s_i \geq f_1 \}$$

Key Ingredients: Optimal Substructure

- Optimal substructure property is exploited by both Greedy and dynamic programming strategies
- Hence one may
  - Try to generate a dynamic programming soln to a problem when a greedy strategy suffices
  - Or, may mistakenly think that a greedy soln works when in fact a DP soln is required

Example:Knapsack Problems(S, w)

Knapsack Problems

- The 0-1Knapsack Problem(S, W)
  - A thief robbing a store finds n items $S = \{I_1, I_2, \ldots, I_n\}$, the ith item is worth v_i dollars and weighs w_i pounds, where v_i and w_i are integers
  - He wants to take as valuable a load as possible, but he can carry at most W pounds in his knapsack, where W is an integer
  - The thief cannot take a fractional amount of an item
- The Fractional Knapsack Problem (S, W)
  - The scenario is the same
  - But, the thief can take fractions of items rather than having to make binary (0-1) choice for each item

0-1 and Fractional Knapsack Problems

- Both knapsack problems exhibit the optimal substructure property

The 0-1Knapsack Problem(S, W)

- Consider a most valuable load L where $W_L \leq W$
- If we remove item j from this optimal load L

  The remaining load

  $$L_j' = L - \{I_j\}$$

  must be a most valuable load weighing at most

  $$W_j' = W - w_j$$

  pounds that the thief can take from

  $$S_j' = S - \{I_j\}$$

- That is, L_j' should be an optimal soln to the

  0-1 Knapsack Problem(S_j', W_j')

0-1 and Fractional Knapsack Problems

The Fractional Knapsack Problem(S, W)

- Consider a most valuable load L where $W_L \leq W$
- If we remove a weight $0 < w \leq w_j$ of item j from optimal load L

  The remaining load

  $$L_j' = L - \{w \text{ pounds of } I_j\}$$

  must be a most valuable load weighing at most

  $$W_j' = W - w$$

  pounds that the thief can take from

  $$S_j' = S - \{I_j\} \cup \{w_j - w \text{ pounds of } I_j\}$$

- That is, L_j' should be an optimal soln to the

  Fractional Knapsack Problem(S_j', W_j')

Knapsack Problems

Although the problems are similar

- the Fractional Knapsack Problem is solvable by Greedy strategy
- whereas, the 0-1 Knapsack Problem is not

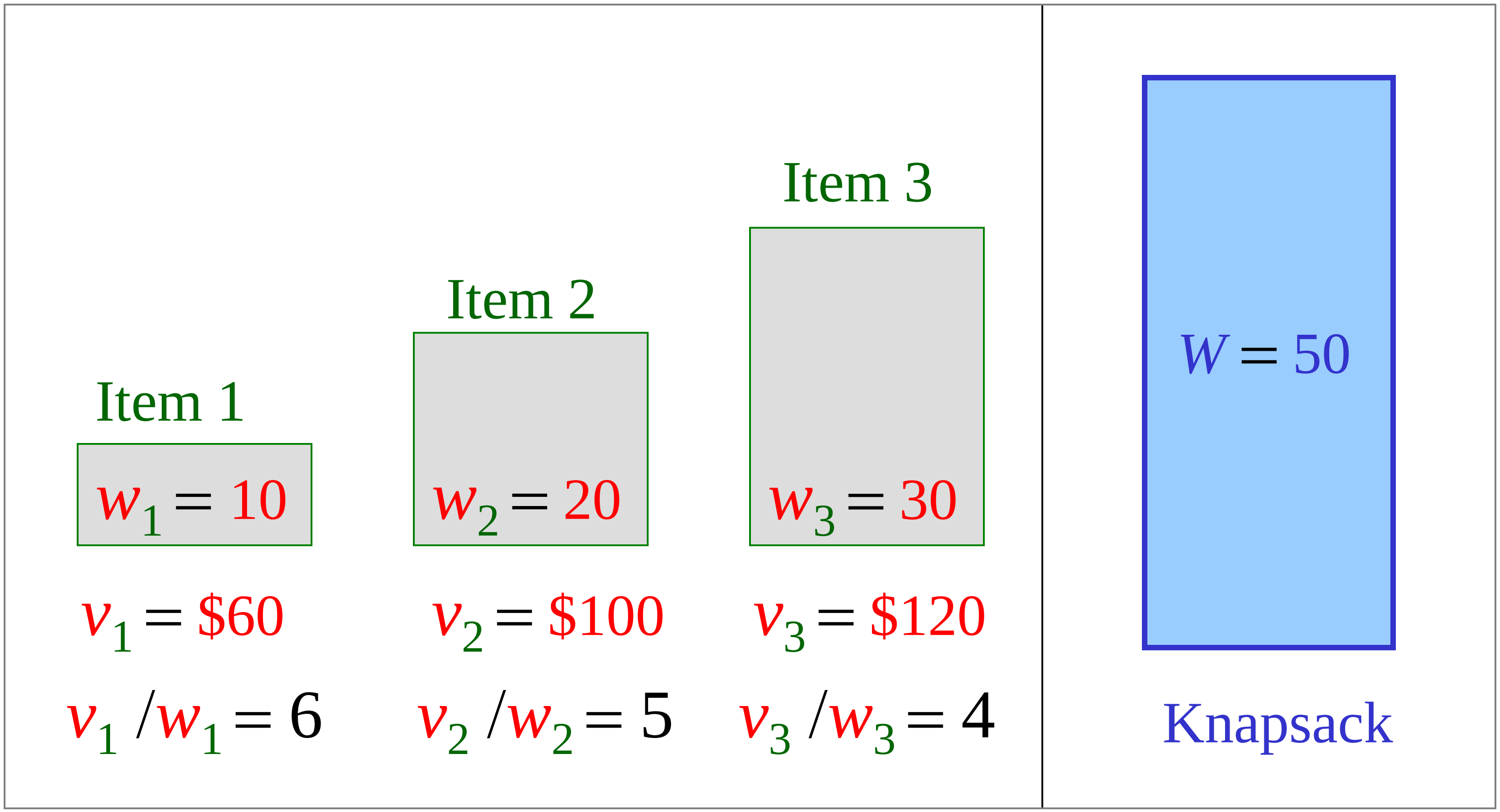
Greedy Solution to Fractional Knapsack

1) Compute the value per pound v_i / w_i for each item
2) The thief begins by taking, as much as possible, of the item with the greatest value per pound
3) If the supply of that item is exhausted before filling the knapsack he takes, as much as possible, of the item with the next greatest value per pound
4) Repeat (2-3) until his knapsack becomes full

- Thus, by sorting the items by value per pound the greedy algorithm runs in $O(n \lg n)$ time

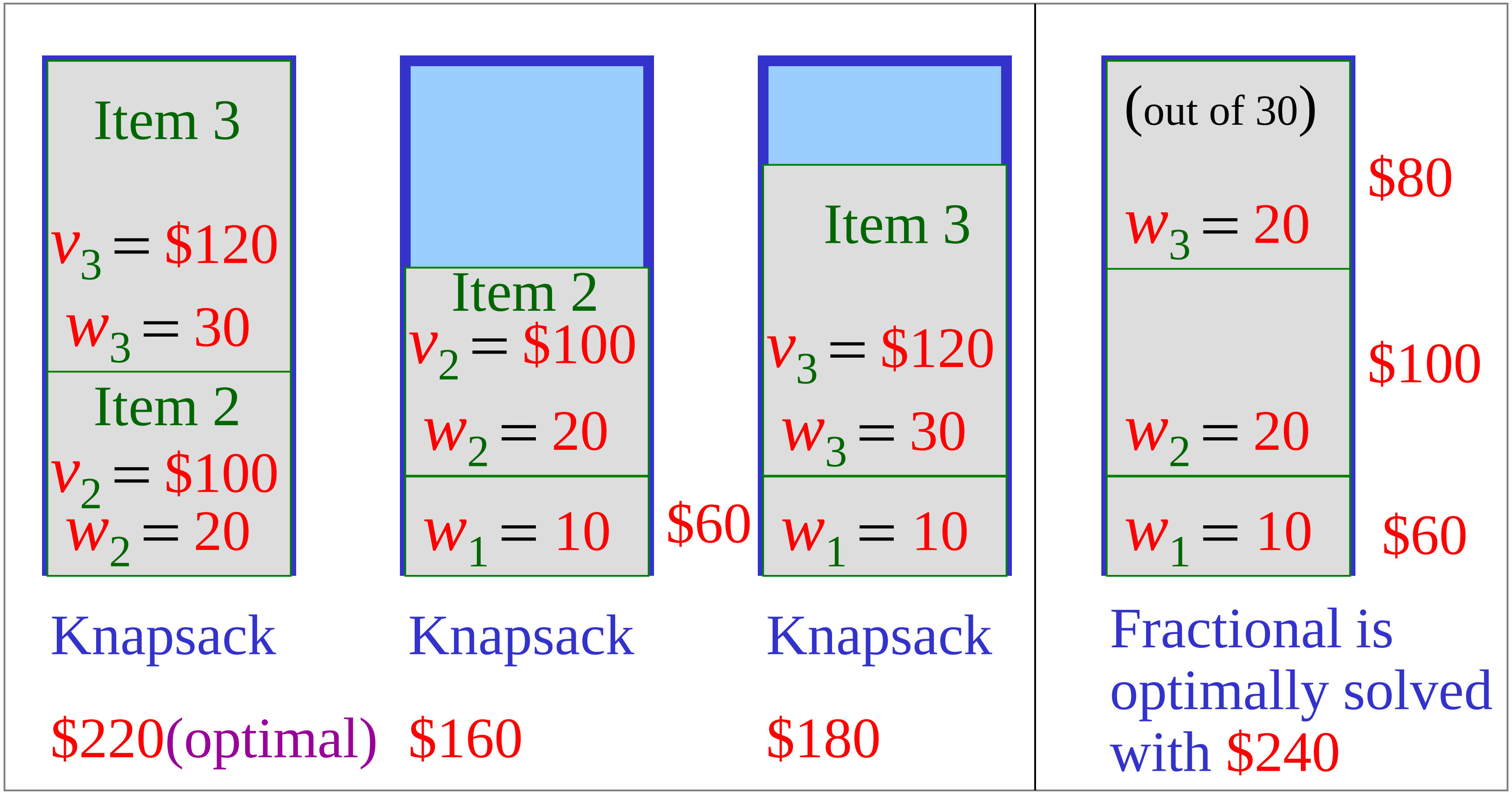
0-1 Knapsack Problem

- Greedy strategy does not work



0-1 Knapsack Problem

- Taking item 1 leaves empty space; lowers the effective value of the load



0-1 Knapsack Problem

- When we consider an item I_j for inclusion we must compare the solutions to two subproblems
  - Subproblems in which I_j is included and excluded
- The problem formulated in this way gives rise to many

  overlapping subproblems (a key ingredient of DP)

  In fact, dynamic programming can be used to solve the 0-1 Knapsack problem

0-1 Knapsack Problem

- A thief robbing a store containing n articles $\{a_1, a_2, \ldots, a_n\}$
  - The value of ith article is v_i dollars (v_i is integer)
  - The weight of ith article is w_i kg (w_i is integer)
- Thief can carry at most W kg in his knapsack
- Which articles should he take to maximize the value of his load?
- Let $K_{n,W}=\{a_1, a_2, \ldots, a_n{:}W\}$ denote 0-1 knapsack problem
- Consider the solution as a sequence of n decisions
  - i.e., ith decision: whether thief should pick a_i for optimal load

0-1 Knapsack Problem

Optimal substructure property:

- Let a subset S of articles be optimal for $K_{n,W}$
- Let a_i be the highest numbered article in S

Then

$$S' = S - \{a_i\}$$

is an optimal solution for subproblem

$$K_{i-1,W-w_i} = \{a_1, a_2, \ldots, a_{i-1} : W - w_i\} \qquad \text{with}$$

$$c(S) = v_i + c(S')$$

where $c(\cdot)$ is the value of an optimal load '$\cdot$'

0-1 Knapsack Problem

Recursive definition for value of optimal soln:

- Define $c[i,w]$ as the value of an optimal solution for $K_{i,w} = \{a_1, a_2, \ldots, a_i : w\}$

$$c[i,w] = \begin{cases} 0, & \text{if } i = 0 \text{ or } w = 0 \\ c[i-1,w], & \text{if } w_i > w \\ max\{v_i + c[i-1, w - w_i]\,,\, c[i-1,w]\} & \text{o.w} \end{cases}$$

0-1 Knapsack Problem

Recursive definition for value of optimal soln:

This recurrence says that an optimal solution $S_{i,w}$ for $K_{i,w}$

- either contains $a_i \Rightarrow c(S_{i,w}) = v_i + c(S_{i-1,w-w_i})$
- or does not contain $a_i \Rightarrow c(S_{i,w}) = c(S_{i-1,w})$

- If thief decides to pick a_i
  - He takes v_i value and he can choose from $\{a_1, a_2, \ldots, a_{i-1}\}$ up to the weight limit $w - w_i$ to get $c[i-1, w - w_i]$
- If he decides not to pick a_i
  - He can choose from $\{a_1, a_2, \ldots, a_{i-1}\}$ up to the weight limit w to get $c[i-1, w]$
- The better of these two choices should be made

DP Solution to 0-1 Knapsack

KNAP0-1(v, w, n, W)

for $\omega \leftarrow 0$ **to** W **do**
    $c[0, \omega] \leftarrow 0$
for $i \leftarrow 1$ **to** n **do**
    $c[i, 0] \leftarrow 0$
for $i \leftarrow 1$ **to** n **do**
    for $\omega \leftarrow 1$ **to** W **do**
        if $w_i \leq \omega$ **then**
            $c[i, \omega] \leftarrow max\{v_i + c[i-1, \omega - w_i], c[i-1, \omega]\}$
        else
            $c[i, \omega] \leftarrow c[i-1, \omega]$
return $c[n, W]$

c is an $(n+1)\times(W+1)$ array; $c[0..n : 0..W]$

Note: table is computed in row-major order

Run time: $T(n) = \Theta(nW)$

Finding the Set *S* of Articles in an Optimal Load

SOLKNAP0-1(a, v, w, n, W, c)

$i \leftarrow n$; $\omega \leftarrow W$
$S \leftarrow \varnothing$

while $i > 0$ **do**
  if $c[i, \omega] = c[i-1, \omega]$ **then**
    $i \leftarrow i - 1$
  else
    $S \leftarrow S \cup \{a_i\}$
    $\omega \leftarrow \omega - w_i$
    $i \leftarrow i - 1$
return S

Huffman Codes

- Widely used and very effective technique for compressing data
- Savings of 20% to 90% are typical
- Depending on the characteristics of the file being compressed Huffman's greedy algorithm
  - uses a table of the frequencies of occurrence of each character
  - to build up an optimal way of representing each character as a binary string

Example: A 100,000-character data file that is to be compressed only 6 characters {a, b, c, d, e, f} appear

	a	b	c	d	e	f
frequency (in thousands)	45K	13K	12K	16K	9K	5K
fixed-length codeword	000	001	010	011	100	101
variable-length codeword	0	101	100	111	1101	1100
variable-length codeword	0	10	110	1110	11110	11111

Huffman Codes

Binary character code:

- each character is represented by a unique binary string

Fixed-length code:

- needs 3 bits to represent 6 characters
- requires $100.000 \times 3 = 300{,}000$ bits to code the entire file

Variable-length code:

- can do better by giving frequent characters short codewords & infrequent words long codewords
- requires $45 \times 1 + 13 \times 3 + 12 \times 3 + 16 \times 3 + 9 \times 4 + 5 \times 4$
  $= 224{,}000$ bits

Prefix Codes

Prefix codes: No codeword is also a prefix of some other codeword

It can be shown that:

optimal data compression achievable by a character code can always be achieved with a prefix code

Prefix codes simplify encoding (compression) and decoding

Encoding: Concatenate the codewords representing each character of the file

e.g. 3 char file "abc" $\xrightarrow{\text{encoded}}$ 0.101.100 = 0101100

Prefix Codes

Decoding: is quite simple with a prefix code

the codeword that begins an encoded file is unambigious since no codeword is a prefix of any other

- identify the initial codeword
- translate it back to the original character
- remove it from the encoded file
- repeat the decoding process on the remainder of the encoded file

e.g. string 001011101 parses uniquely as

$$0.0.101.1101 \xrightarrow{\text{decoded}} \text{aabe}$$

Prefix Codes

Convenient representation for the prefix code:

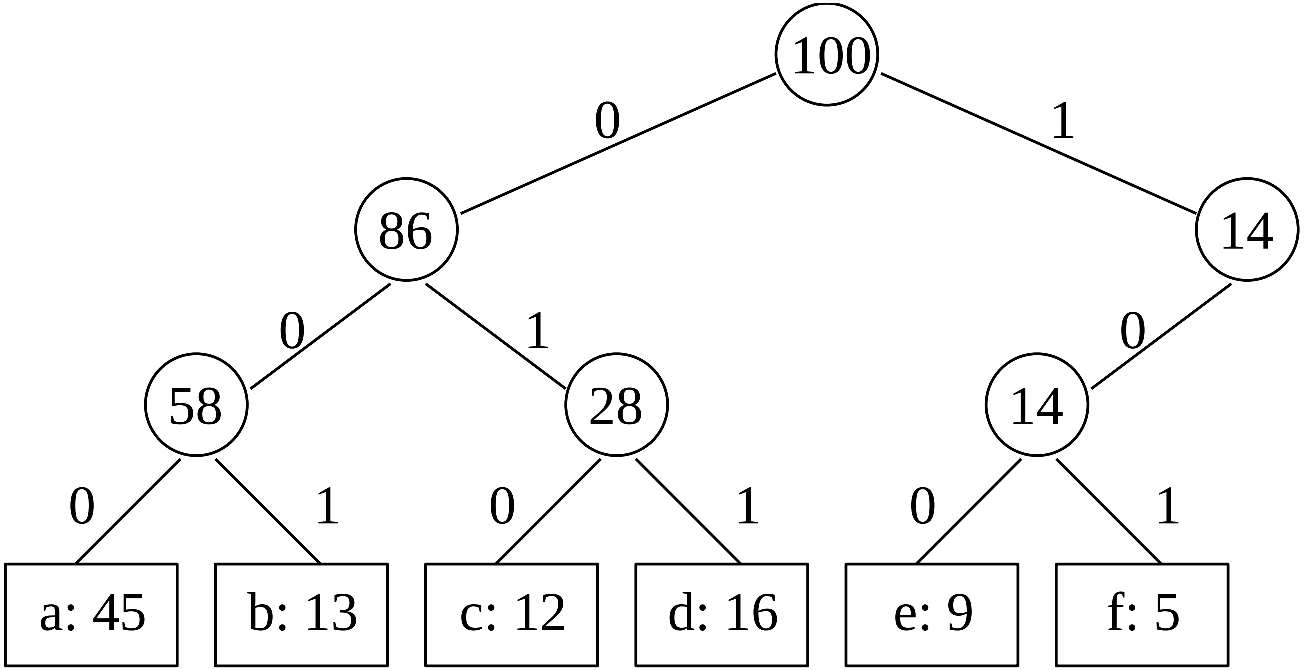
a binary tree whose leaves are the given characters

Binary codeword for a character is the path from the root to that character in the binary tree

"0" means "go to the left child"

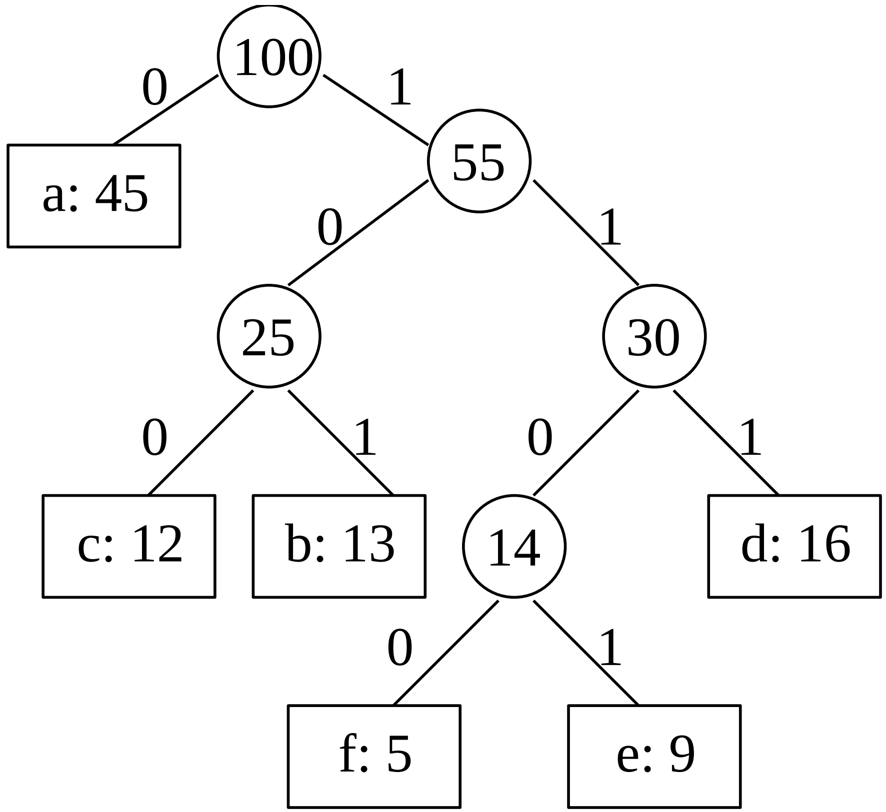
"1" means "go to the right child"

Binary Tree Representation of Prefix Codes



The binary tree corresponding to the fixed-length code

Binary Tree Representation of Prefix Codes



The binary tree corresponding to the optimal variable-length code

An optimal code for a file is always represented by a full binary tree

Full Binary Tree Representation of Prefix Codes

Consider an FBT corresponding to an optimal prefix code

It has $|C|$ leaves (external nodes)

One for each letter of the alphabet where C is the alphabet from which the characters are drawn

Lemma: An FBT with $|C|$ external nodes has exactly $|C|-1$ internal nodes

Full Binary Tree Representation of Prefix Codes

Consider an FBT T corresponding to a prefix code

How to compute, $B(T)$, the number of bits required to encode a file

$f(c)$: frequency of character c in the file

$d_T(c)$: depth of c's leaf in the FBT T

note that $d_T(c)$ also denotes length of the codeword for c

$$B(T) = \sum_{c \in C} f(c)\, d_T(c)$$

which we define as the cost of the tree T

Prefix Codes

Lemma: Let each internal node i is labeled with the sum of the weight $w(i)$ of the leaves in its subtree

Then $B(T) = \sum_{c \in C} f(c)\, d_T(c) = \sum_{i \in I_T} w(i)$ where

I_T denotes the set of internal nodes in T

Proof: Consider a leaf node c with $f(c)$ & $d_T(c)$

Then, $f(c)$ appears in the weights of $d_T(c)$ internal node along the path from c to the root

Hence, $f(c)$ appears $d_T(c)$ times in the above summation

Constructing a Huffman Code

Huffman invented a greedy algorithm that constructs an optimal prefix code called a Huffman code

The greedy algorithm

- builds the FBT corresponding to the optimal code in a bottom-up manner
- begins with a set of $|C|$ leaves
- performs a sequence of $|C|-1$ "merges" to create the final tree

Constructing a Huffman Code

A priority queue Q, keyed on f, is used to identify the two least-frequent objects to merge

The result of the merger of two objects is a new object

- inserted into the priority queue according to its frequency
- which is the sum of the frequencies of the two objects merged

Constructing a Huffman Code

HUFFMAN(C)

$n \leftarrow |C|$
$Q \leftarrow C$
for $i \leftarrow 1$ to $n - 1$ do
  $z \leftarrow$ ALLOCATE-NODE()
  $x \leftarrow \text{left}[z] \leftarrow$ EXTRACT-MIN(Q)
  $y \leftarrow \text{right}[z] \leftarrow$ EXTRACT-MIN(Q)
  $f[z] \leftarrow f[x] + f[y]$
  INSERT(Q, z)
return EXTRACT-MIN(Q) Δ only one object left in Q

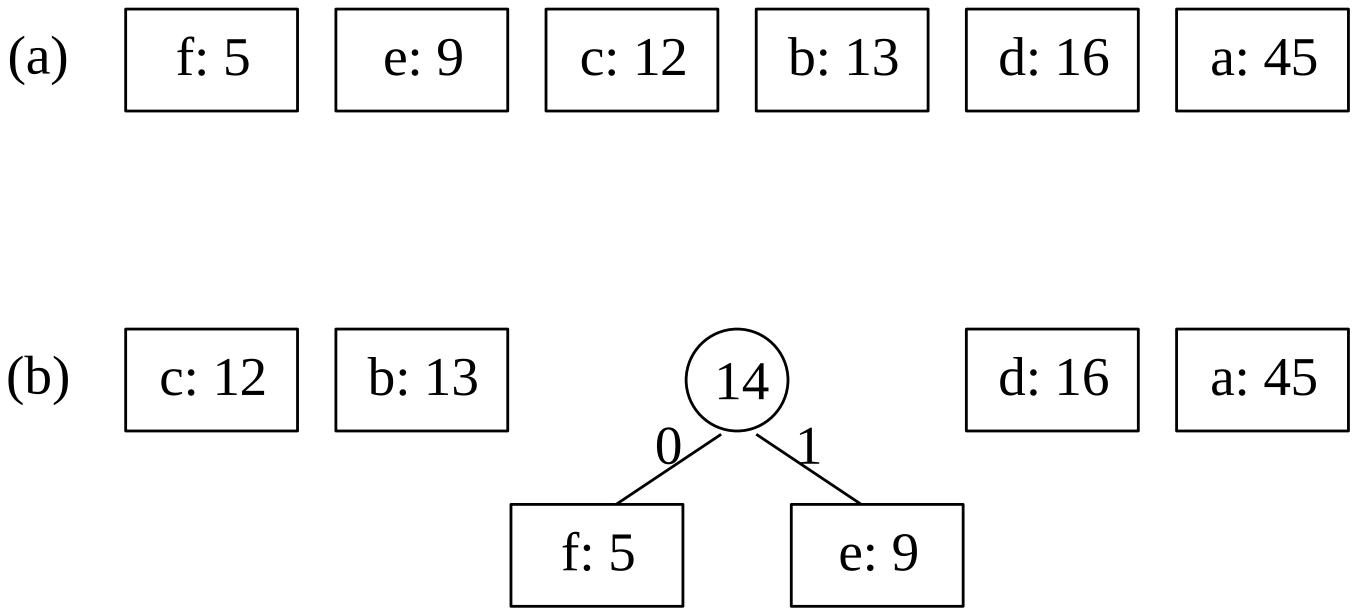
Priority queue is implemented as a binary heap

Initiation of Q (BUILD-HEAP): O(n) time

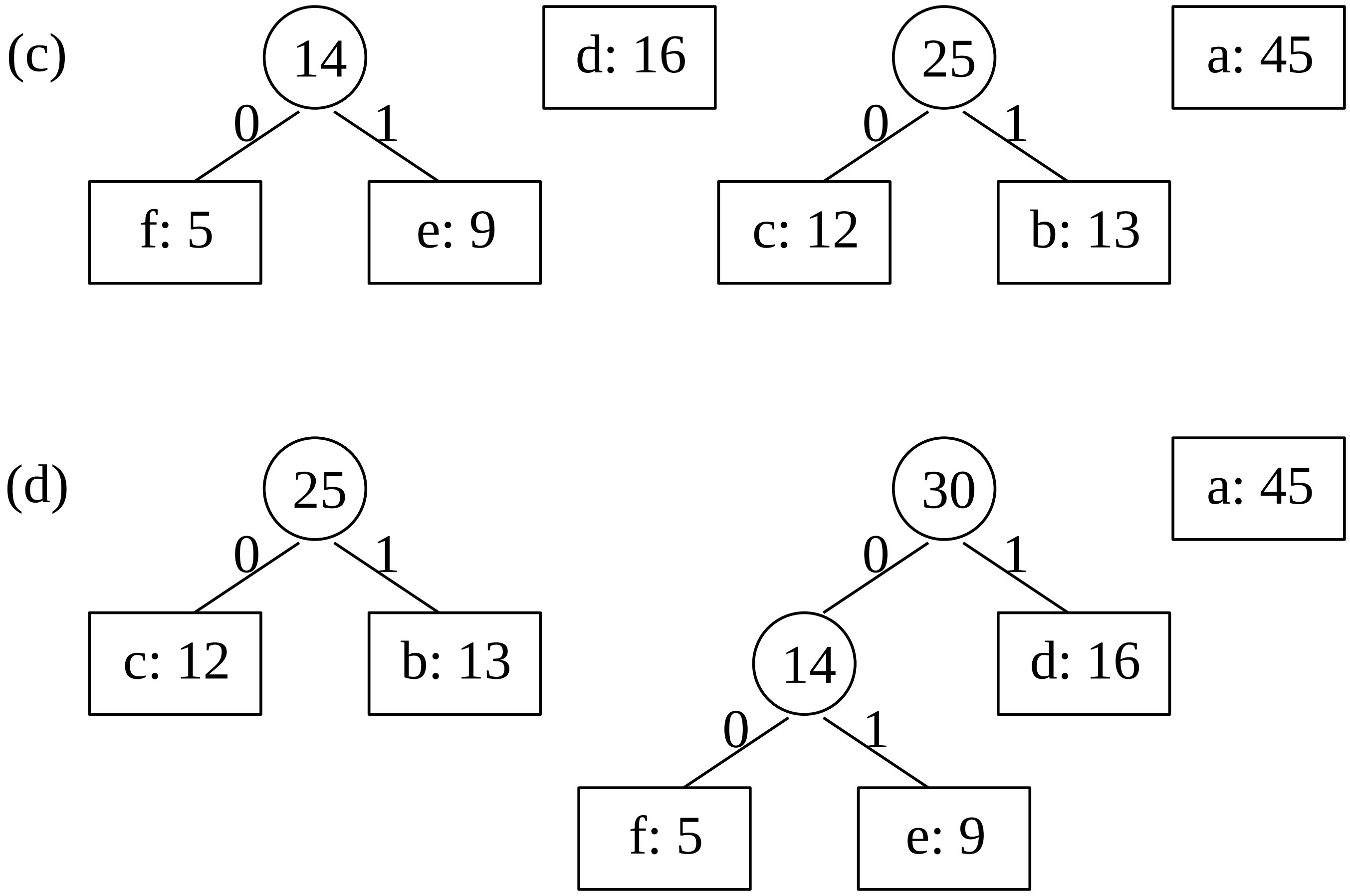
EXTRACT-MIN & INSERT take O(lg$n$) time on Q with n objects

$$T(n) = \sum_{i=1}^{n} \lg i = O(\lg(n!)) = O(n \lg n)$$

Constructing a Huffman Code

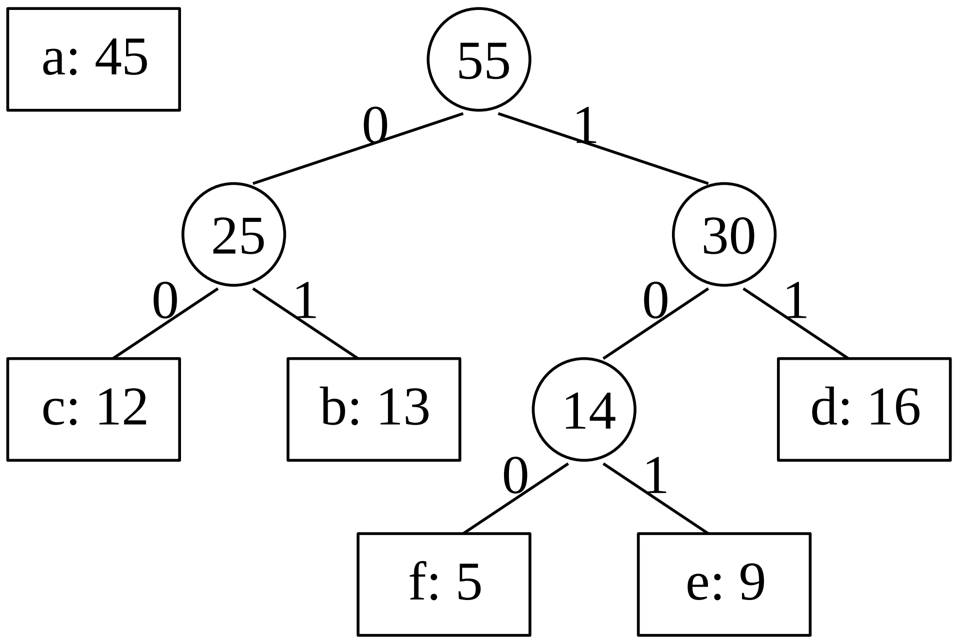


Constructing a Huffman Code



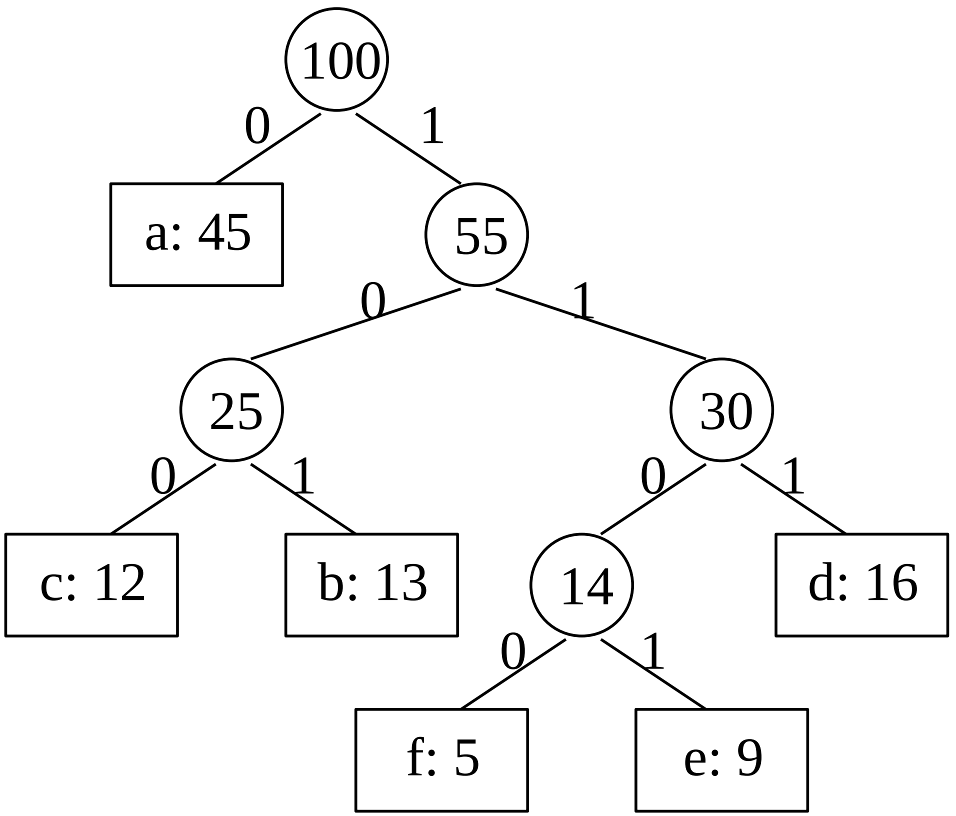
Constructing a Huffman Code

(e)



Constructing a Huffman Code

(f)



Correctness of Huffman's Algorithm

We must show that the problem of determining an optimal prefix code

- exhibits the greedy choice property
- exhibits the optimal substructure property

Lemma 1: Let x & y be two characters in C having the lowest frequencies

Then, $\exists$ an optimal prefix code for C in which the codewords for x & y have the same length and differ only in the last bit

Correctness of Huffman's Algorithm

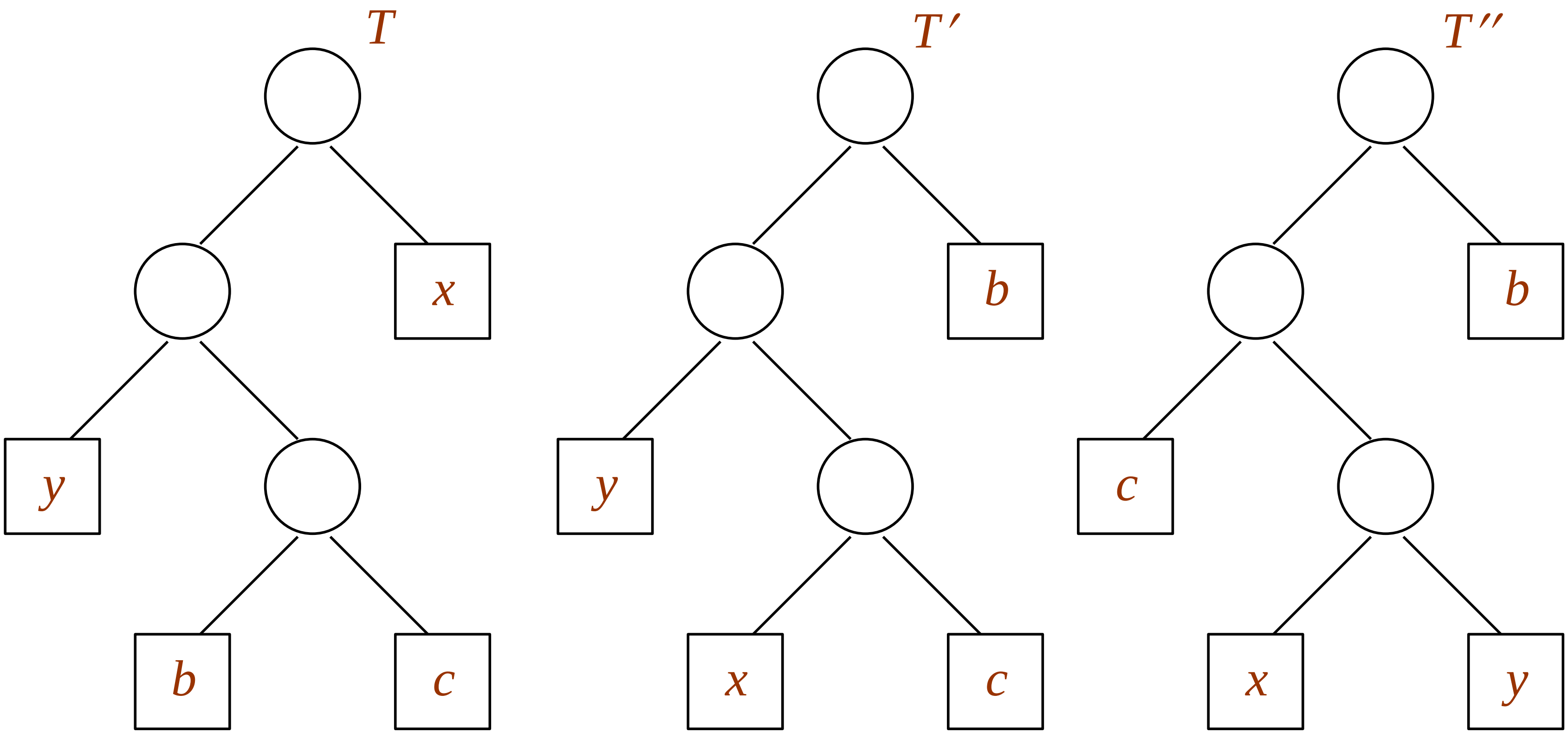
Proof: Take tree T representing an arbitrary optimal code

Modify T to make a tree representing another optimal code such that characters x & y appear as sibling leaves of max-depth in the new tree

Assume that $f[b] \leq f[c]$ & $f[x] \leq f[y]$

Since $f[x]$ & $f[y]$ are two lowest leaf frequencies, in order, and $f[b]$ & $f[c]$ are two arbitrary leaf frequencies, in order, $f[x] \leq f[b]$ & $f[y] \leq f[c]$

Correctness of Huffman's Algorithm



$T \Rightarrow T'$: exchange the positions of the leaves b & x

$T' \Rightarrow T''$: exchange the positions of the leaves c & y

Greedy-Choice Property of Determining an Optimal Code

Proof of Lemma 1 (continued):

The difference in cost between T and T' is

$$
\begin{aligned}
B(T) = B(T') &= \sum_{c \in C} f(c) d_T(c) - \sum_{c \in C} f(c) d_{T'}(c) \\
&= f[x] d_T(x) + f[b] d_T(b) - f[x] d_{T'}(x) - f[b] d_{T'}(b) \\
&= f[x] d_T(x) + f[b] d_T(b) - f(x) d_T(b) - f[b] d_T(x) \\
&= f[b](d_T(b) - d_T(x)) - f[x](d_T(b) - d_T(x)) \\
&= (f[b] - f[x])(d_T(b) - d_T(x)) \geq 0
\end{aligned}
$$

Greedy-Choice Property of Determining an Optimal Code

Proof of Lemma 1 (continued):

Since $f[b] - f[x] \geq 0$ and $d_T(b) \geq d_T(x)$
therefore $B(T') \leq B(T)$

We can similary show that
$B(T') - B(T'') \geq 0 \Rightarrow B(T'') \leq B(T')$
which implies $B(T'') \leq B(T)$

Since T is optimal $\Rightarrow B(T'') = B(T) \Rightarrow T''$ is also optimal

Greedy-Choice Property of Determining an Optimal Code

Lemma 1 implies that
process of building an optimal tree by mergers can begin with the greedy choice of merging those two characters with the lowest frequency

We have already proved that $B(T) = \sum_{i \in I_T} w(i)$, that is,
the total cost of the tree constructed is the sum of the costs of its mergers (internal nodes) of all possible mergers

At each step Huffman chooses the merger that incurs the least cost

Greedy-Choice Property of Determining an Optimal Code

Lemma 2: Consider any two characters x & y that appear as sibling leaves in optimal T and let z be their parent

Then, considering z as a character with frequency $f[z] = f[x] + f[y]$

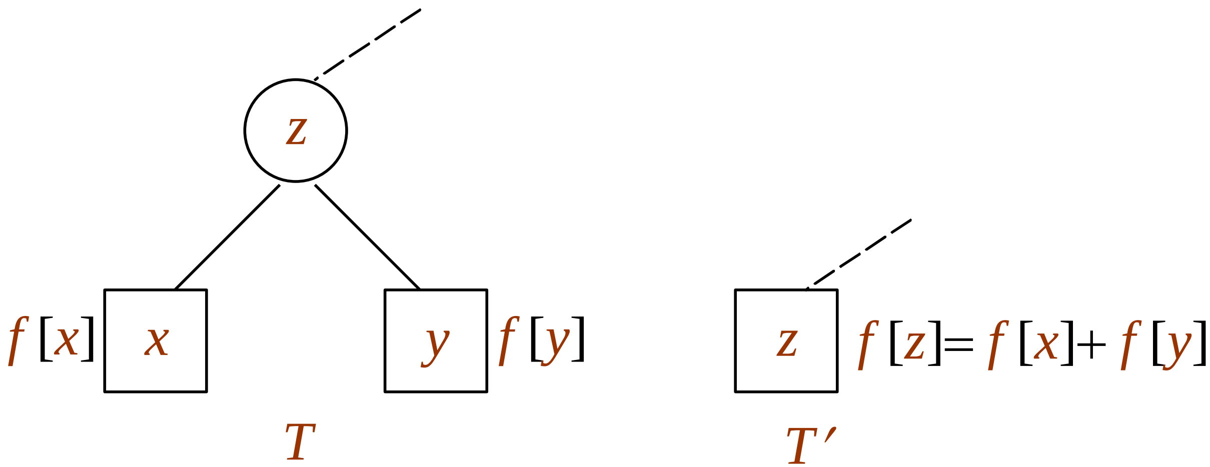
The tree $T' = T - \{x, y\}$ represents an optimal prefix code for the alphabet $C' = C - \{x, y\} \cup \{z\}$

Greedy-Choice Property of Determining an Optimal Code

Proof: Try to express cost of T in terms of cost of T'

For each $c \in C' = C - \{x, y\}$ we have

$$d_T(c) = d_{T'}(c) \Rightarrow f(c)d_T(c) = f(c)d_{T'}(c)$$



$$\begin{aligned} B(T) &= B(T') + f[x](d_T(z)+1) + f[y](d_T(z)+1) - f[z]d_T(z) \\ &= B(T') + f[z](d_T(z)+1) - f[z]d_T(z) \\ &= B(T') + f[z] = B(T') + f[x] + f[y] \end{aligned}$$

Greedy-Choice Property of Determining an Optimal Code

Proof (continued): If T' represents a nonoptimal prefix code for the alphabet C'

Then, $\exists$ a tree T'' whose leaves are characters in C' such that $B(T'') < B(T')$

Since z is a character in C', it appears as a leaf in T''

If we add x & y as children of z in T'' then we obtain a prefix code for x with cost

$$B(T'') + f[x] + f[y] < B(T') + f[x] + f[y] = B(T)$$

contradicting the optimality of T