COMP 446 / 546 ALGORITHM DESIGN AND ANALYSIS

LECTURE 9 GREEDY ALGORITHMS

ALPTEKİN KÜPÇÜ

Based on slides of David Luebke, Jennifer Welch, and Cevdet Aykanat

GREEDY ALGORITHM PARADIGM

- **Greedy algorithms:**
  - Make a sequence of choices
  - Each choice is the one that seems to be the best at that point
  - Choice only depends on what has been done so far
    - No looking ahead
  - Choice produces a smaller problem to be solved
- **In order for greedy algorithm to solve a problem optimally, the optimal solution to the problem must be made up of optimal solutions to sub-problems**

DESIGNING A GREEDY ALGORITHM

- Cast the problem so that we make a greedy (locally optimal) choice and are left with one smaller sub-problem
- Prove there is always a (globally) optimal solution to the original problem that makes the greedy choice
- Show that the greedy choice together with an optimal solution to the sub-problem gives a (globally) optimal solution to the original problem

GREEDY ALGORITHMS

- **A *greedy algorithm* always makes the choice that looks best at the moment**
  - Some examples:
    - Walking to the cafeteria
    - Playing Halflife
  - The hope: a locally optimal choice will lead to a globally optimal solution

ACTIVITY SELECTION PROBLEM

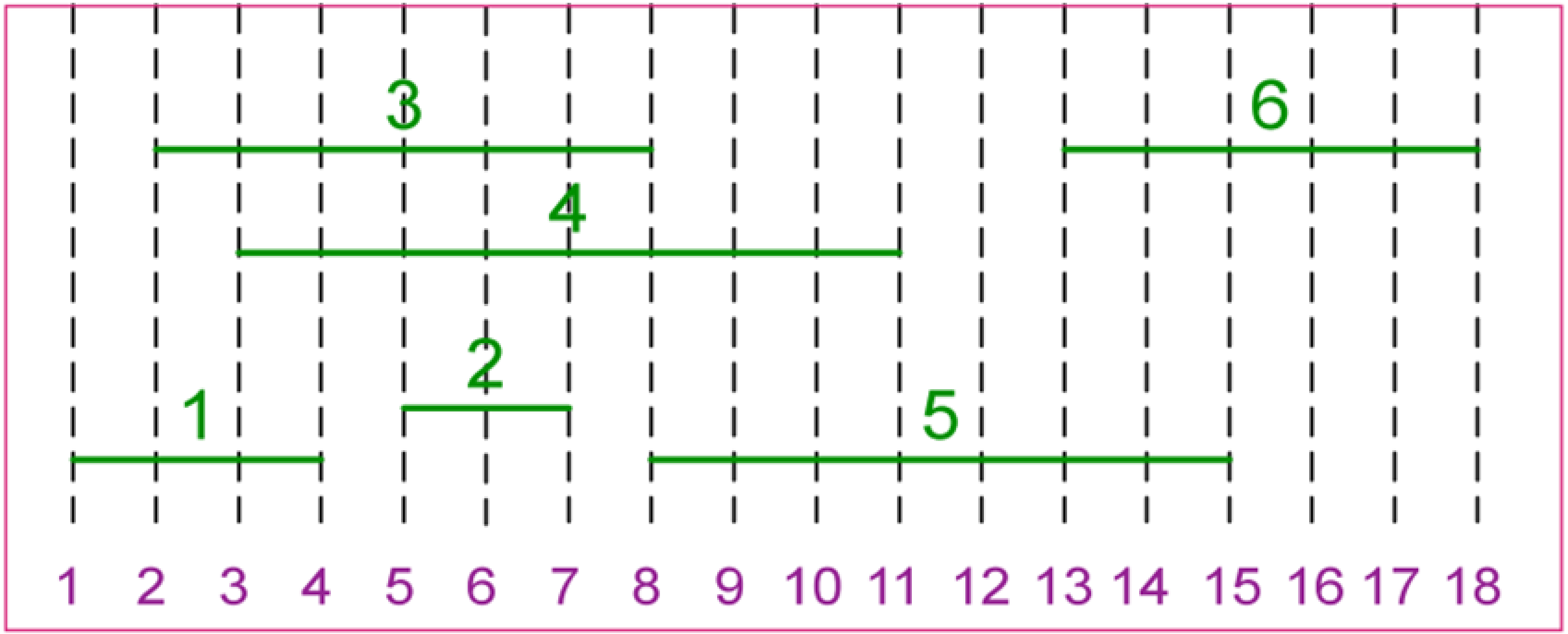
- **Problem: Get your money's worth out of a lunapark**
  - You have a ticket that lets you play any game within the day
  - Lots of possible games, each starting and ending at different times
  - Goal: Join as many games as possible
- ***Any other similar examples?***

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**
  - One activity starts after the other finishes
  - Either $s_i \geq f_j$ or $s_j \geq f_i$

ACTIVITY SELECTION PROBLEM EXAMPLE

- S={ [1,4), [5,7), [2,8), [3,11), [8,15), [13,18) }
- **Assume (w.l.o.g.) that $f_1 \leq f_2 \leq \ldots \leq f_n$**



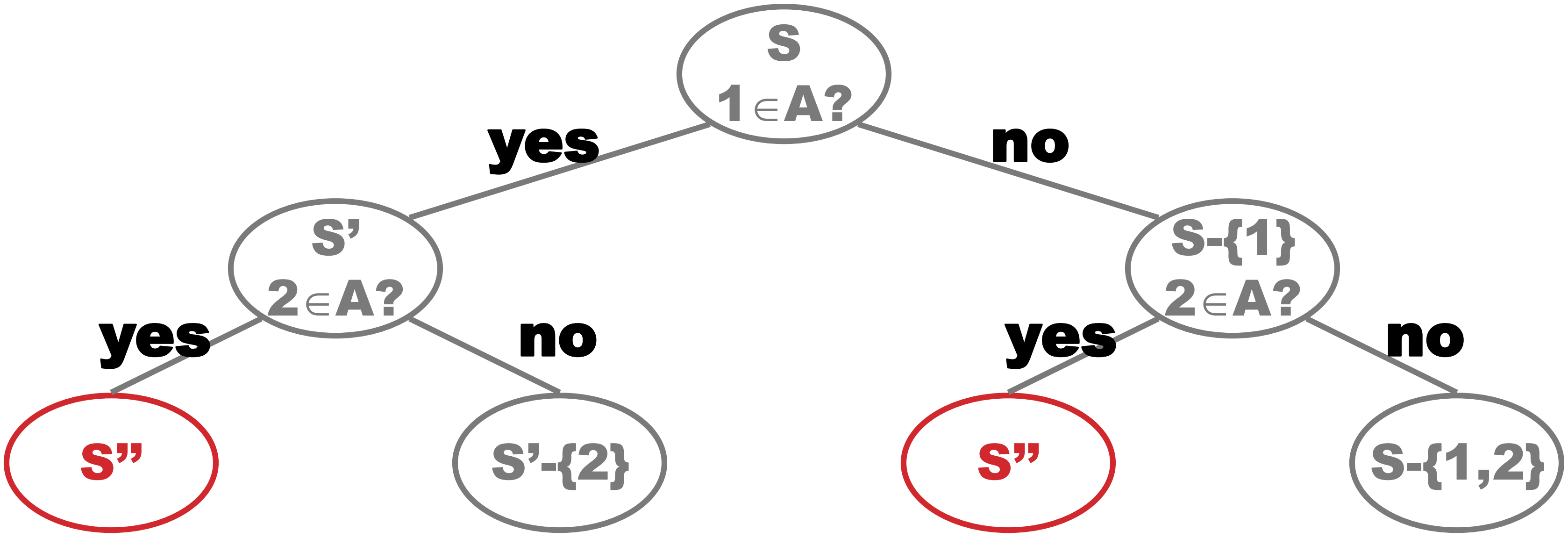
OPTIMAL SUBSTRUCTURE

- **Theorem: let k be the activity with the earliest finish time in an optimal solution $A \subseteq S$ then,**
- **$A - \{k\}$ is an optimal solution to sub-problem $S_k' = \{ i \in S : s_i \geq f_k \}$**

- **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

REPEATED SUBPROBLEMS

- **Consider a recursive algorithm that tries all possible compatible subsets to find a maximal set, and notice repeated sub-problems:**



GREEDY CHOICE PROPERTY IN ACTIVITY SELECTION

- **Theorem: There exists an optimal solution A ⊆ S such that 1 ∈ A (Remember $f_1 \le f_2 \le \cdots \le f_n$)**
- **Proof: Let A = { k, l, m … } be an optimal solution such that $f_k \le f_l \le f_m \le \cdots$**

GREEDY CHOICE PROPERTY IN ACTIVITY SELECTION

- **Theorem: There exists an optimal solution A ⊆ S such that 1 ∈ A (Remember $f_1 \leq f_2 \leq \cdots \leq f_n$)**
- **Proof: Let A = { k, l, m ... } be an optimal solution such that $f_k \leq f_l \leq f_m \leq \cdots$**
  - If $k = 1$ then schedule A begins with the greedy choice, done.

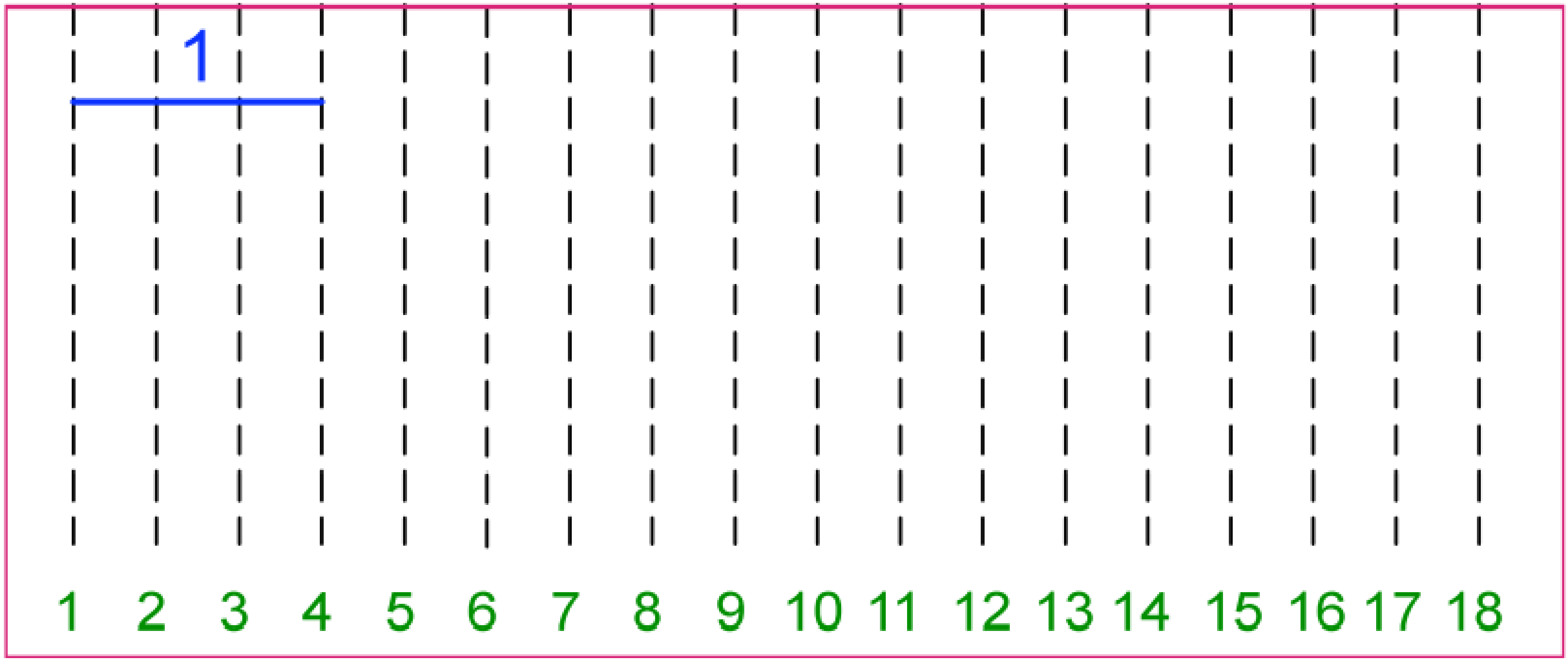
GREEDY CHOICE PROPERTY IN ACTIVITY SELECTION

- **Theorem: There exists an optimal solution A ⊆ S such that 1 ∈ A (Remember $f_1 \leq f_2 \leq \cdots \leq f_n$)**
- **Proof: Let A = { k, l, m ... } be an optimal solution such that $f_k \leq f_l \leq f_m \leq \cdots$**
  - If $k = 1$ then schedule A begins with the greedy choice, done.
  - If k > 1 then show that ∃ another optimal solution 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}
    - Hence B is compatible
    - |B| = |A| - 1 + 1 = |A|
    - Therefore B is optimal, and contains 1, done.

GREEDY ALGORITHM FOR ACTIVITY SELECTION

- **Simple algorithm:**
  - Sort the activities by finish time
  - Schedule the first activity
  - Then schedule the next activity in sorted list which starts after previous activity finishes (compatible)
  - Repeat until no more activities can be added
- **Intuition is even more simple:**
  - Always pick the shortest game available at the time
- ***Pseudocode?***
- ***Runtime?***

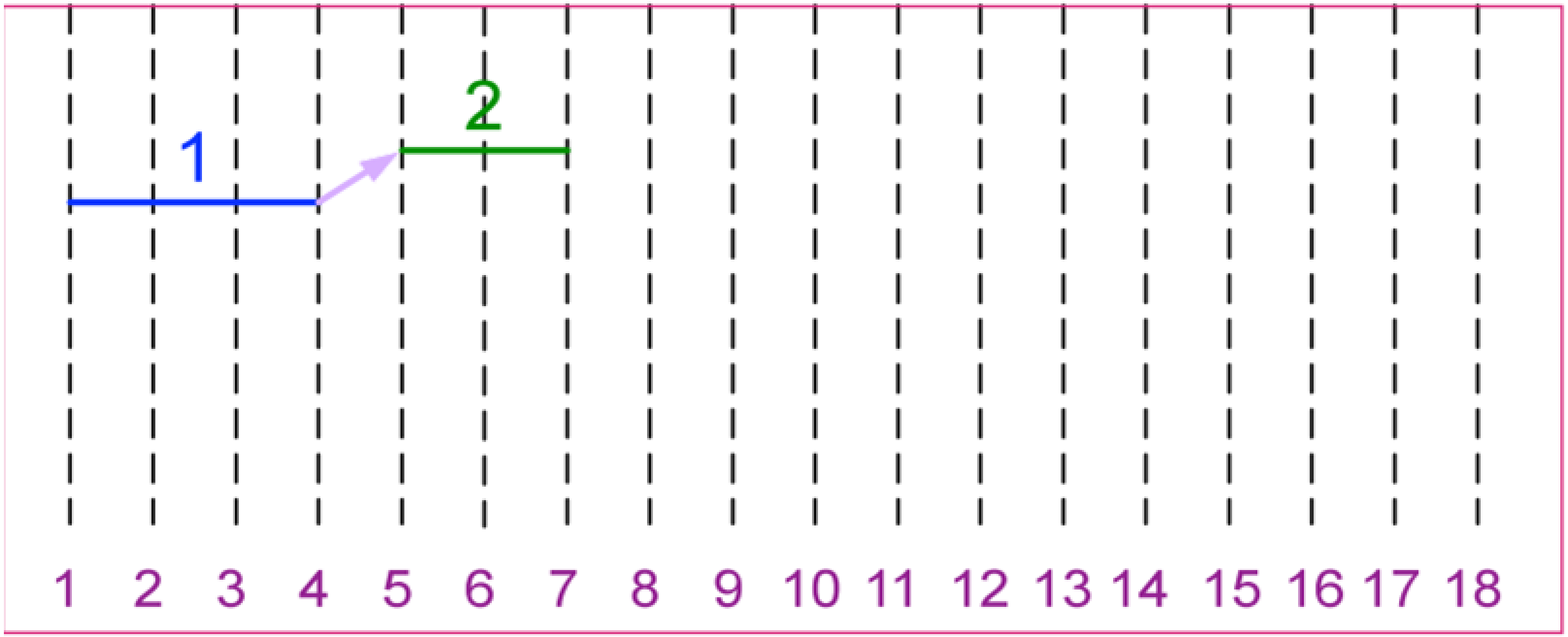
ACTIVITY SELECTION PROBLEM: AN EXAMPLE



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

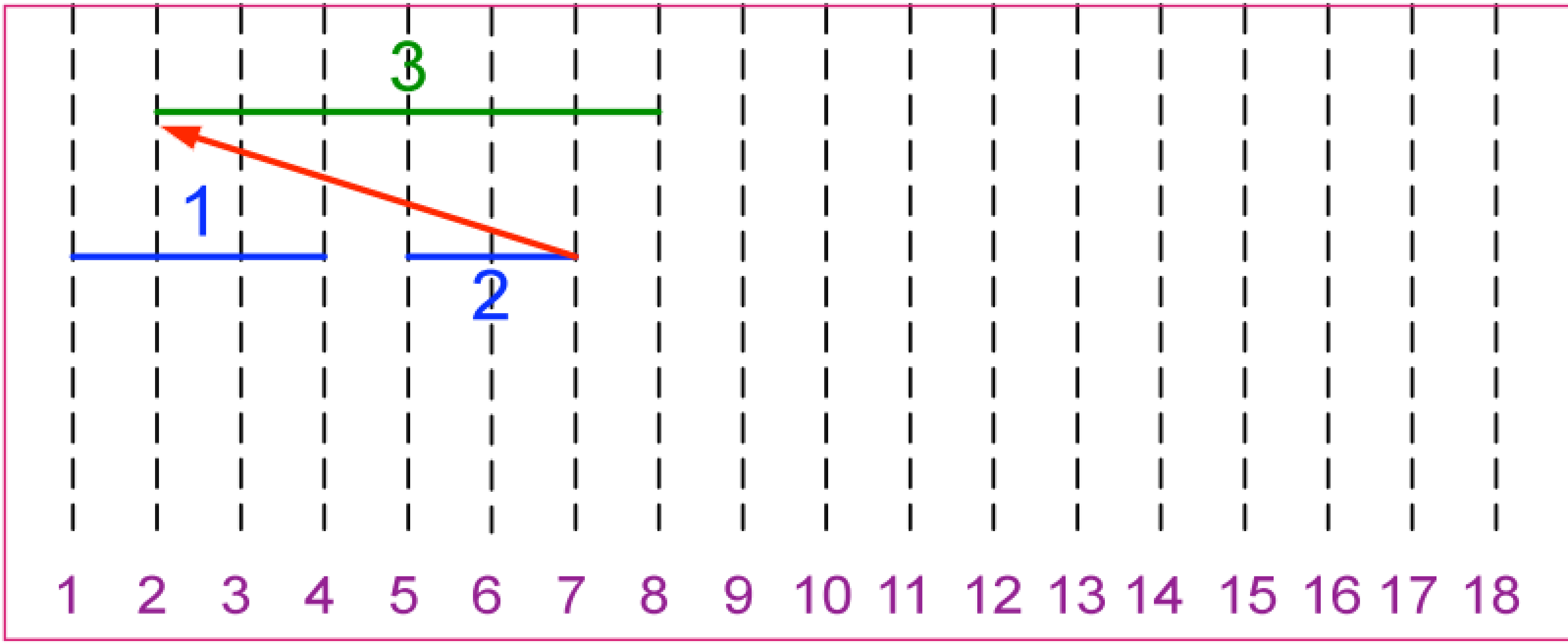
f_{last} **= 0**

ACTIVITY SELECTION PROBLEM: AN EXAMPLE



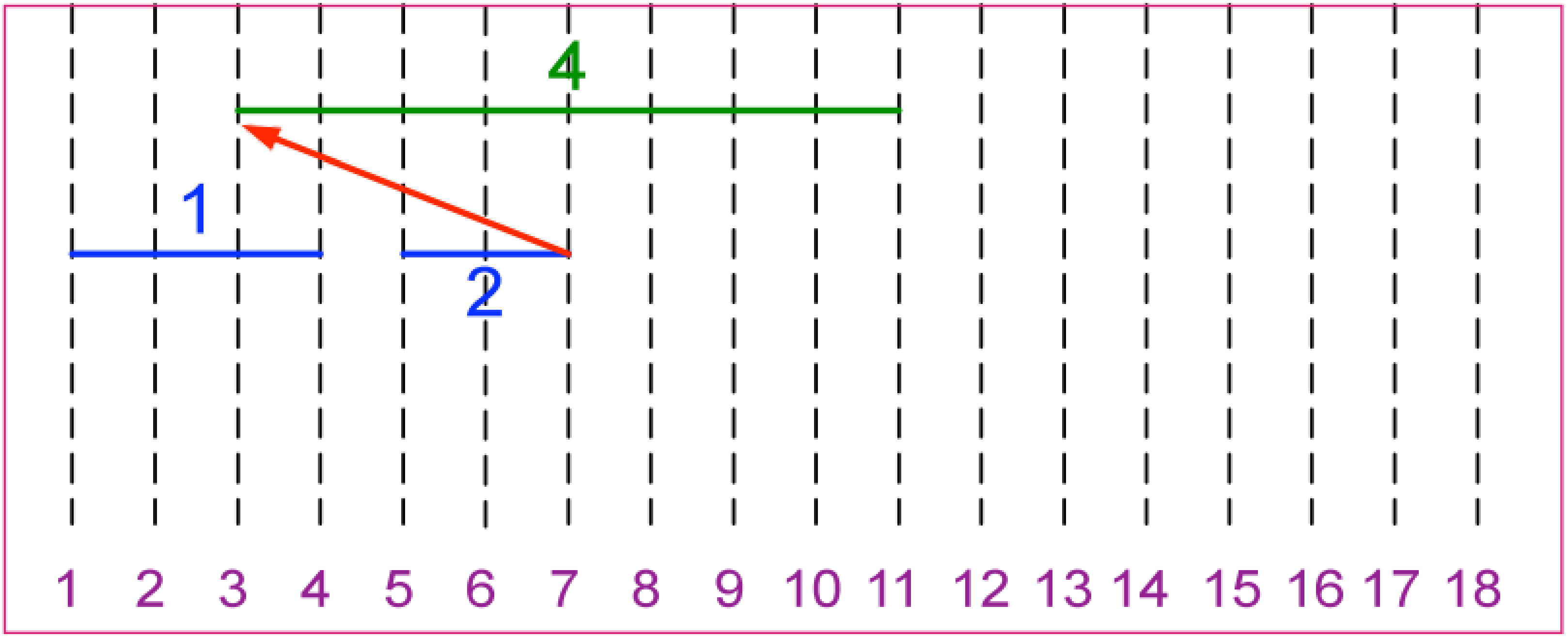
S = { [1,4), [5,7), [2,8), [3,11), [8,15), [13,18) }
f_{last} **= 4**

ACTIVITY SELECTION PROBLEM: AN EXAMPLE



S = { [1,4), [5,7), [2,8), [3,11), [8,15), [13,18) }
f_{last} = 7
Activity 3 is incompatible

ACTIVITY SELECTION PROBLEM: AN EXAMPLE

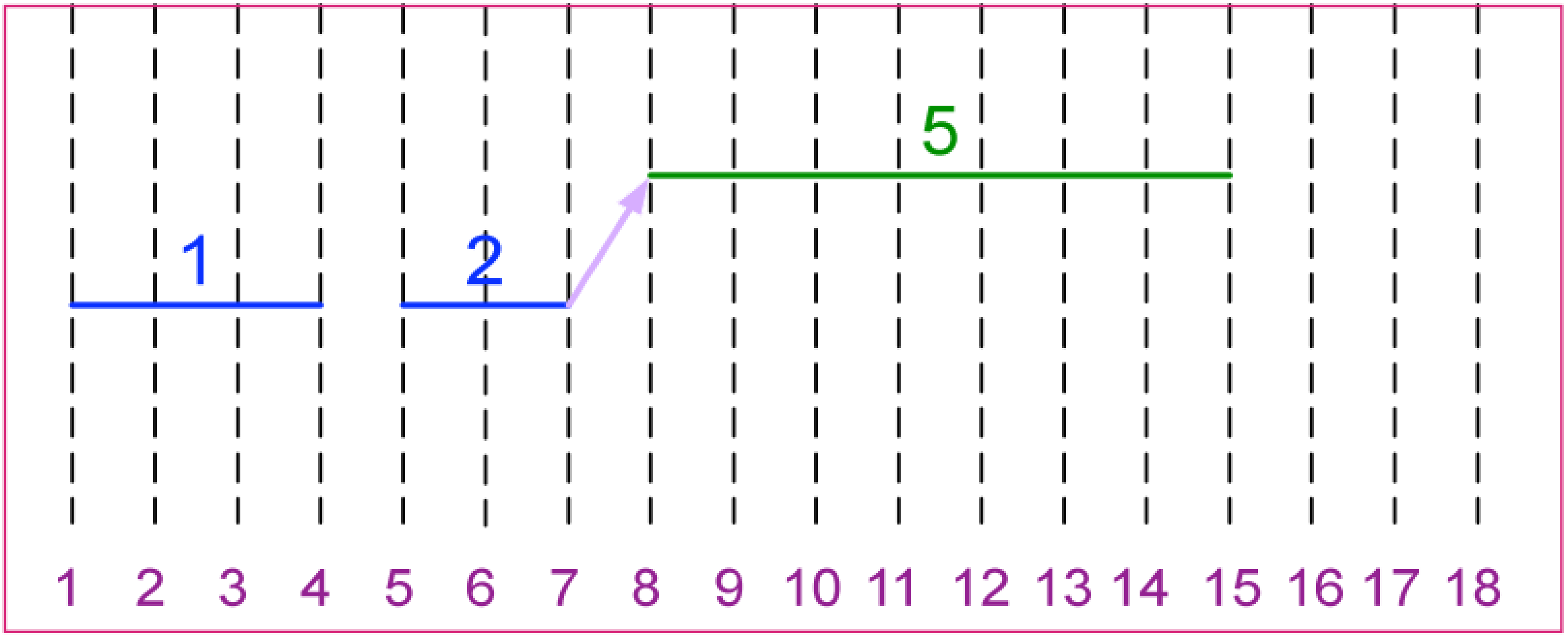


S = { [1,4), [5,7), [2,8), [3,11), [8,15), [13,18) }

$f_{last} = 7$

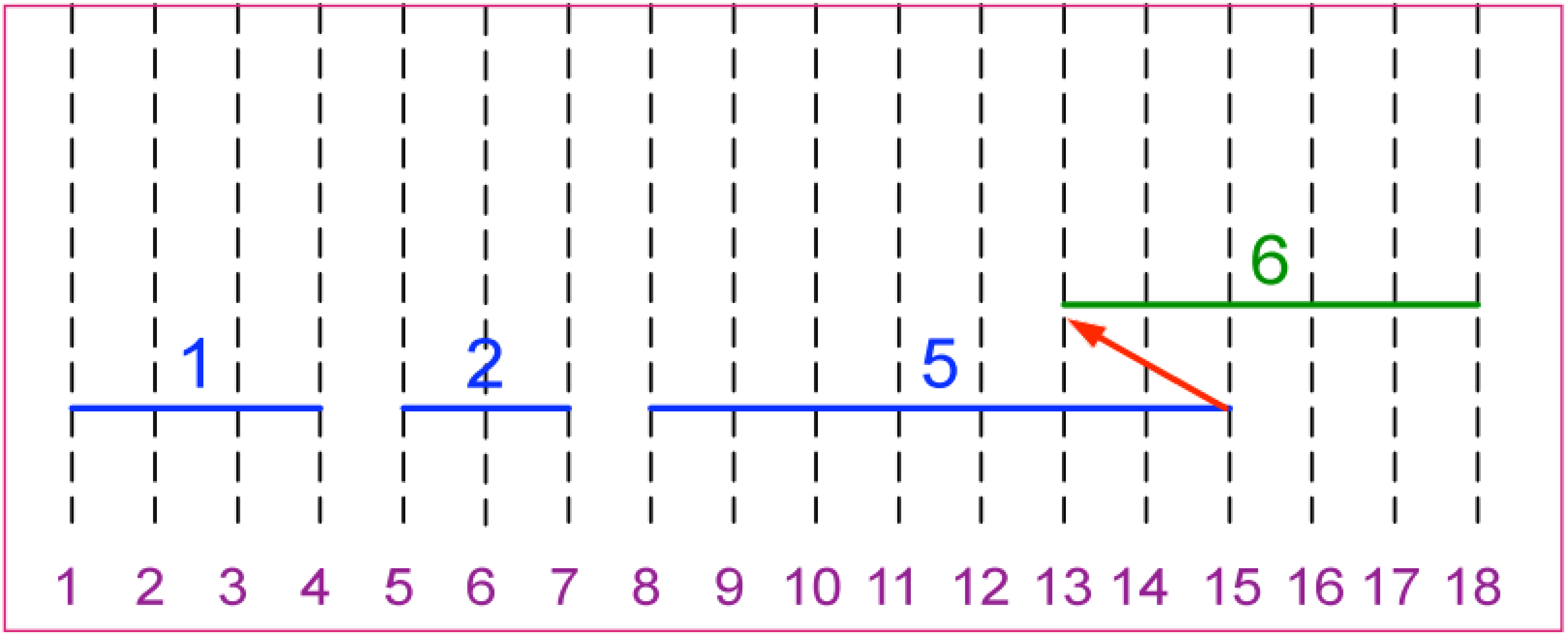
Activity 4 is incompatible

ACTIVITY SELECTION PROBLEM: AN EXAMPLE



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

ACTIVITY SELECTION PROBLEM: AN EXAMPLE

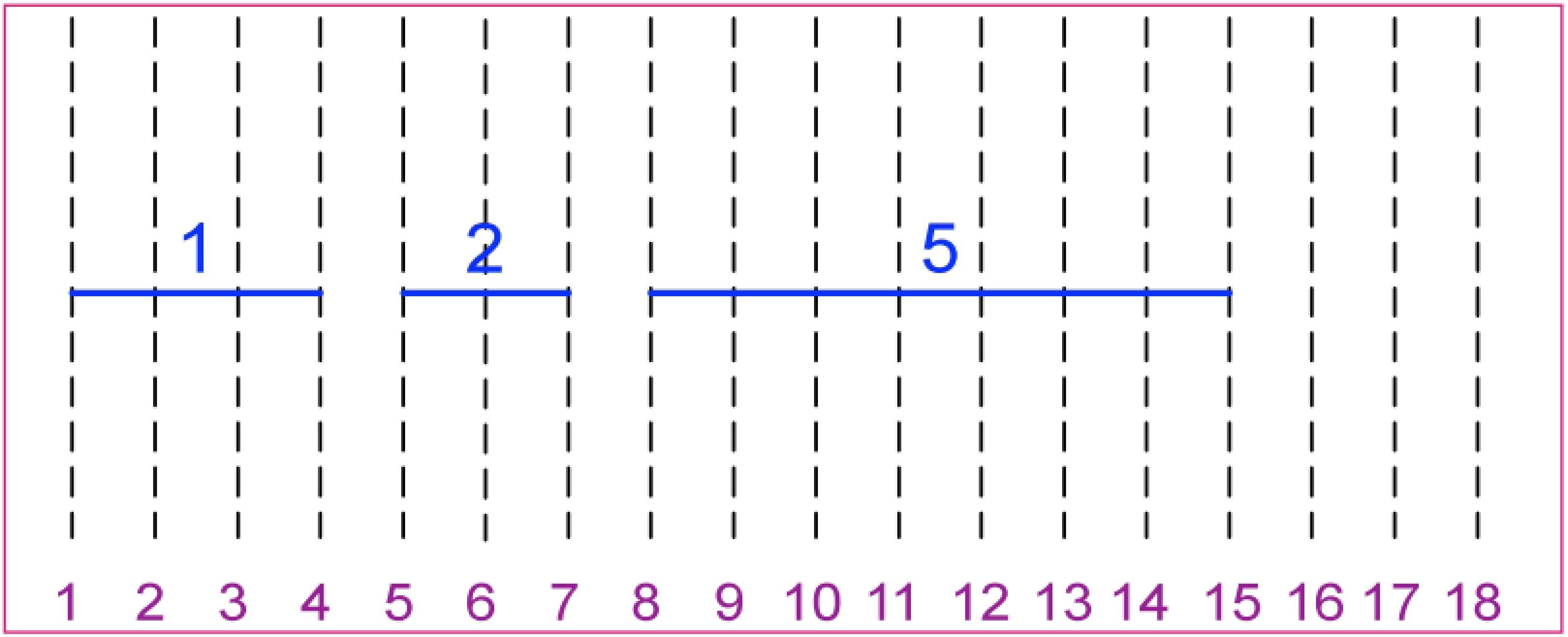


S = { [1,4), [5,7), [2,8), [3,11), [8,15), [13,18) }

f_{last} = 15

Activity 6 is incompatible

ACTIVITY SELECTION PROBLEM: AN EXAMPLE



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

GREEDY ALGORITHM DESIGN

- **1. Cast the optimization problem as one in which we make a choice and are left with one sub-problem to solve.**
- **2. Prove that there's always an optimal solution that makes the greedy choice, so that the greedy choice is always safe.**
- **3. Demonstrate optimal substructure by showing that, combining an optimal solution to the remaining sub-problem with the greedy choice gives an optimal solution to the original problem.**
- **Greedy algorithm:**
  - Make a choice at each step.
  - Make the choice ***before*** solving the sub-problems.
  - Solve **top-down**.

GREEDY ALGORITHM CORRECTNESS

- **Two key ingredients to the optimality of greedy algorithms are**
  - Greedy-choice property
  - Optimal substructure property
- **Typically show the greedy-choice property by what we did for activity selection:**
  - Look at an optimal solution.
    - If it includes the greedy choice, done.
    - Otherwise, modify the optimal solution to include the greedy choice, yielding another optimal solution (pareto-optimality).
- **Can get efficiency gains from greedy-choice property.**
  - Pre-process (sort) input to put it into greedy order.
  - If the data is dynamic, use a priority queue.

OPTIMAL ENCODING

- **Input:**
  - Data file of characters
  - Number of occurrences of each character
- **Output:**
  - A binary encoding of each character so that the data file can be represented as efficiently as possible ("optimal code").
- **Fixed-length encoding:**
  - n unique characters can be encoded with $m = \lceil \log n \rceil$ bits each
  - Easy decoding: each m bits of encoded data identify a character

HUFFMAN CODE

- **Idea:** **Use short codes for frequent characters and long codes for infrequent characters.**

char	a	b	c	d	e	f	total
#	45	13	12	16	9	5	100 chars
fixed	000	001	010	011	100	101	300 bits
variable	0	101	100	111	1101	1100	224 bits

- ***How can we decode?***

PREFIX CODES

- **Prefix codes: No codeword is also a prefix of some other codeword**
- **It can be shown that**
  - Optimal data compression achievable by any character encoding can always be achieved with a prefix code
  - Prefix codes simplify encoding and decoding

- **Encoding: Concatenate the codewords representing each character of the file**

- **Ex: "abc"** $\xrightarrow{\textbf{encoded as}}$ **0||101||100= 0101100**

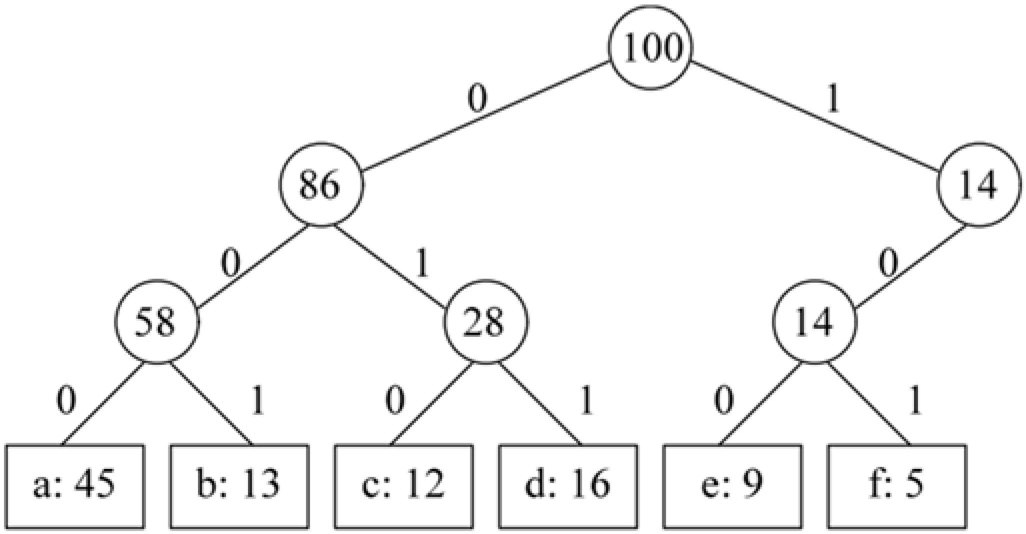
PREFIX CODES

- **Decoding: The codewords are 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

- **Ex: 001011101 parses uniquely as 0||0||101||1101 $\xrightarrow{\text{decoded}}$ "aabe"**

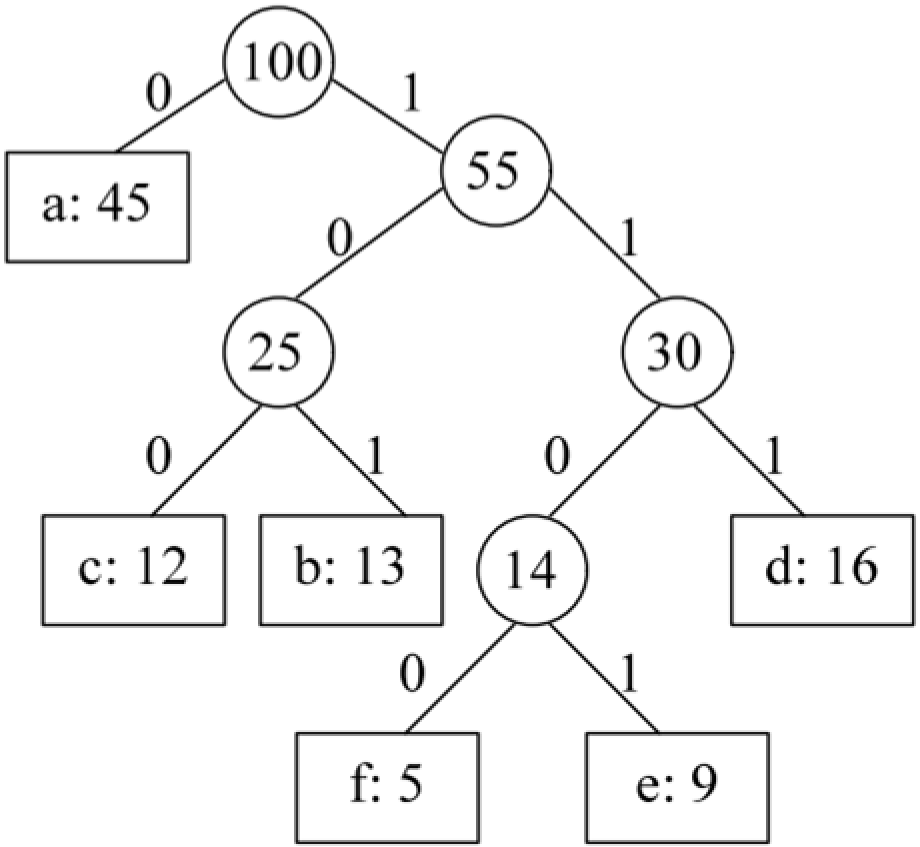
BINARY TREE REPRESENTATION OF PREFIX CODES

- **The binary tree corresponding to our fixed-length code**
  - 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”
  - Leaves are the characters



BINARY TREE REPRESENTATION OF PREFIX CODES

- **The binary tree corresponding to our optimal variable-length code**
- **An optimal code for a file is always represented by a full binary tree**
  - has |C| leaves (external nodes)
  - One leaf for each letter of the alphabet C
- **Lemma: A full binary tree (FBT) with |C| external nodes has exactly |C| – 1 internal nodes.**



BINARY TREE REPRESENTATION OF PREFIX CODES

- **Consider an FBT *T* corresponding to a prefix code**
- **Compute *B(T)*, the number of bits required to encode a file**
  - the cost of the tree T
- $f(c)$: **frequency of character *c* in the input file**
- $d_T(c)$: **depth of the leaf for character *c* in the FBT *T***
  - $d_T(c)$ also denotes length of the codeword for *c*

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

BINARY TREE REPRESENTATION OF PREFIX CODES

- **Let each internal node i be labeled with $w(i)$, the sum of the weights of the leaves in its sub-tree**
- **Lemma: $B(T) = \sum_{c \in C} f(c) d_T(c) = \sum_{i \in I_T} w(i)$ where I_T denotes the set of internal nodes in T.**
- **Proof:**
  - Consider a leaf node c with $f(c)$ & $d_T(c)$
  - $f(c)$ appears in the weights $w(i)$ of $d_T(c)$ internal nodes along the path from c to the root
  - Hence, $f(c)$ appears $d_T(c)$ times in both summations

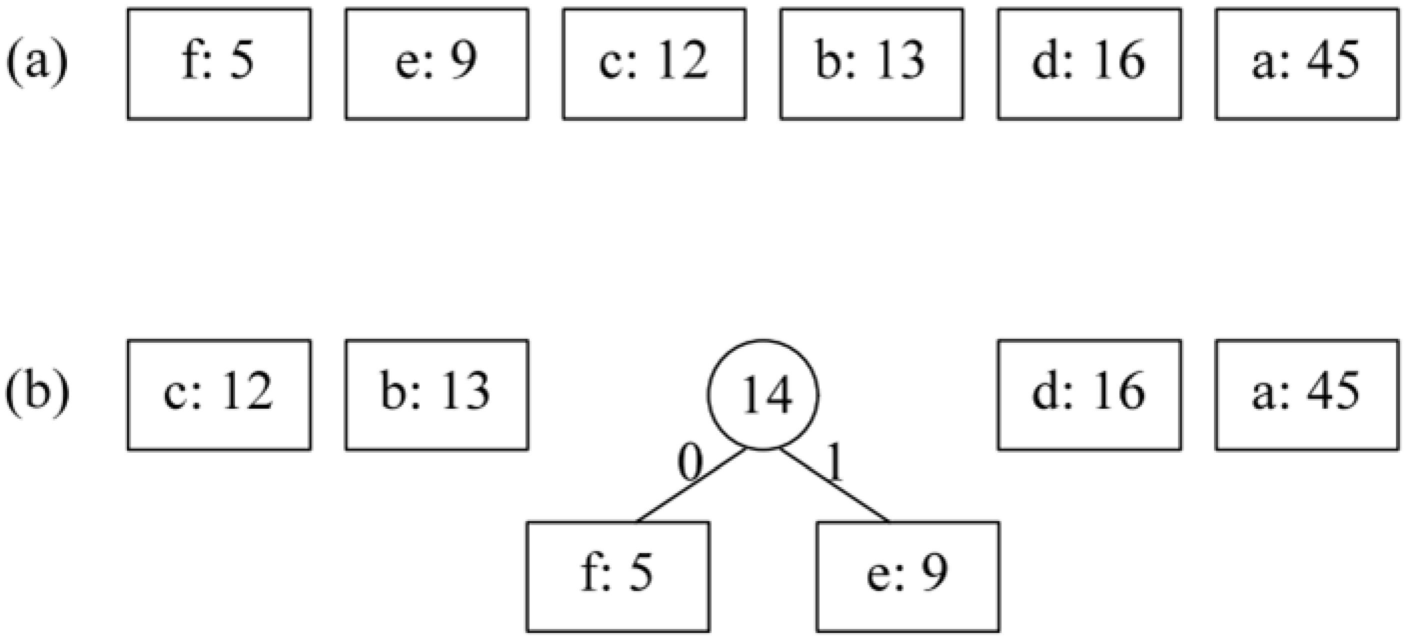
HUFFMAN CODE

- **Huffman code is a greedy algorithm that constructs an optimal prefix code.**
- **Idea: Build an FBT corresponding to the optimal code in a bottom-up manner**
  - Begin with a set of |C| leaves
  - Performs a sequence of |C| – 1 “merges” to create the final tree
- **Algorithm:**
  - Create a priority queue Q, keyed on frequency, to identify the two least-frequent objects to merge
  - The result of a merger of the two objects is a new object
    - Insert the new object into the priority queue according to the sum of the frequencies of the two objects merged
  - Repeat until Q is empty

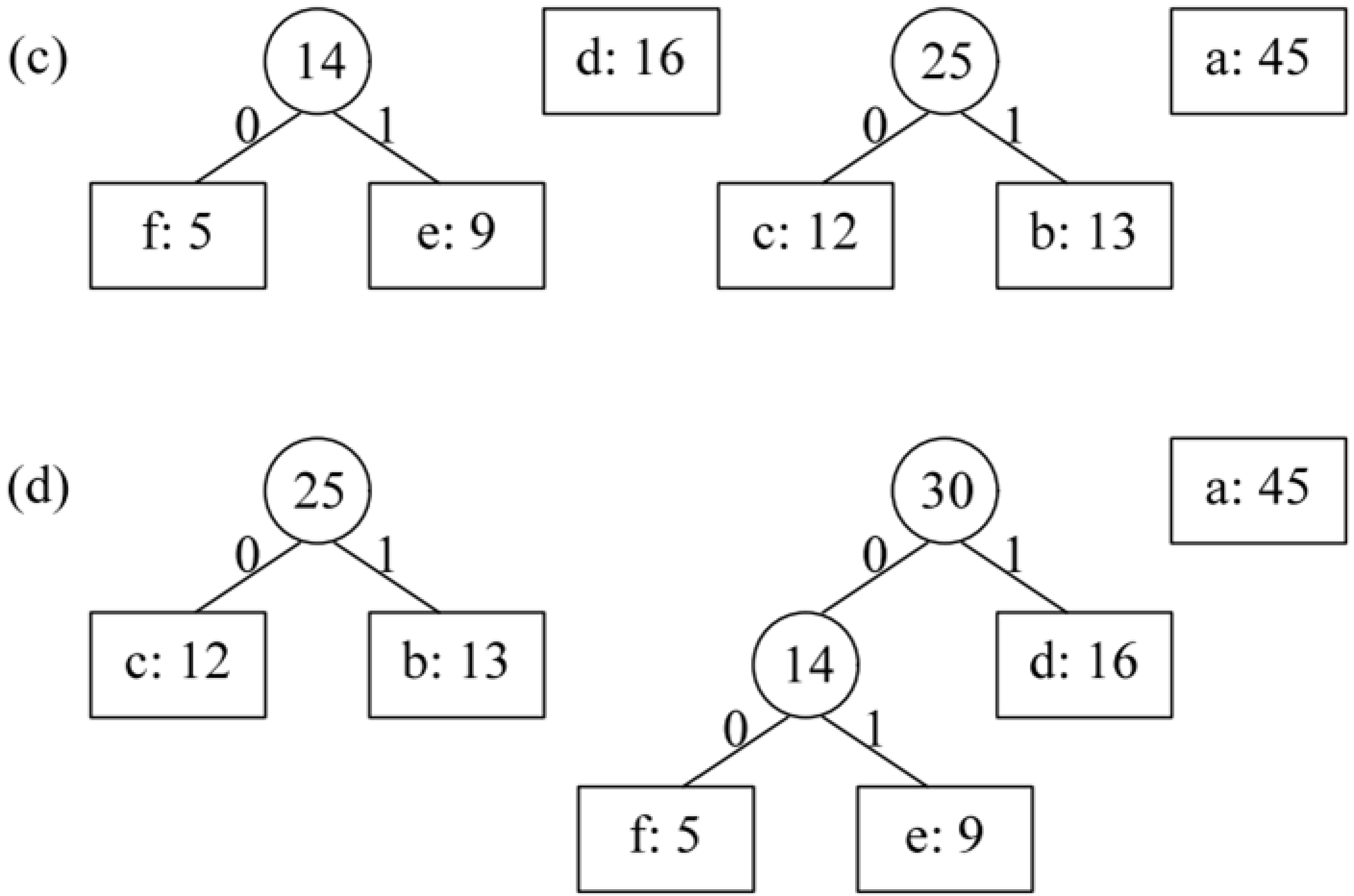
HUFFMAN CODE

- **Given: Set *C* of *n* chars, where *c* occurs *f(c)* times**
- **HUFFMAN-TREE()**
  - Insert each *c* into priority queue *Q* using *f(c)* as key (via BUILDHEAP())
  - for *i* = 1 to *n-1* do
    - x = extract-min(*Q*)
    - y = extract-min(*Q*)
    - make a new node *z* with left child *x* (and edge label 0), right child *y* (and edge label 1), and *f(z) = f(x) + f(y)*
    - insert *z* into *Q* according to *f(z)*
- ***Runtime?***

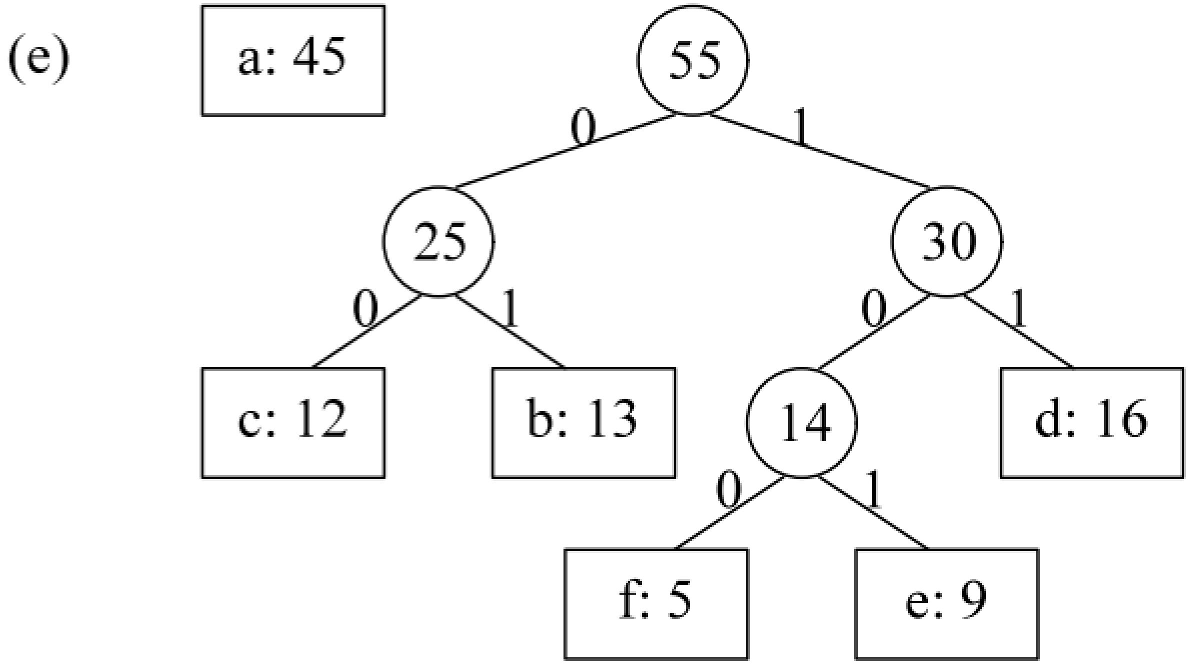
HUFFMAN CODE EXAMPLE



HUFFMAN CODE EXAMPLE

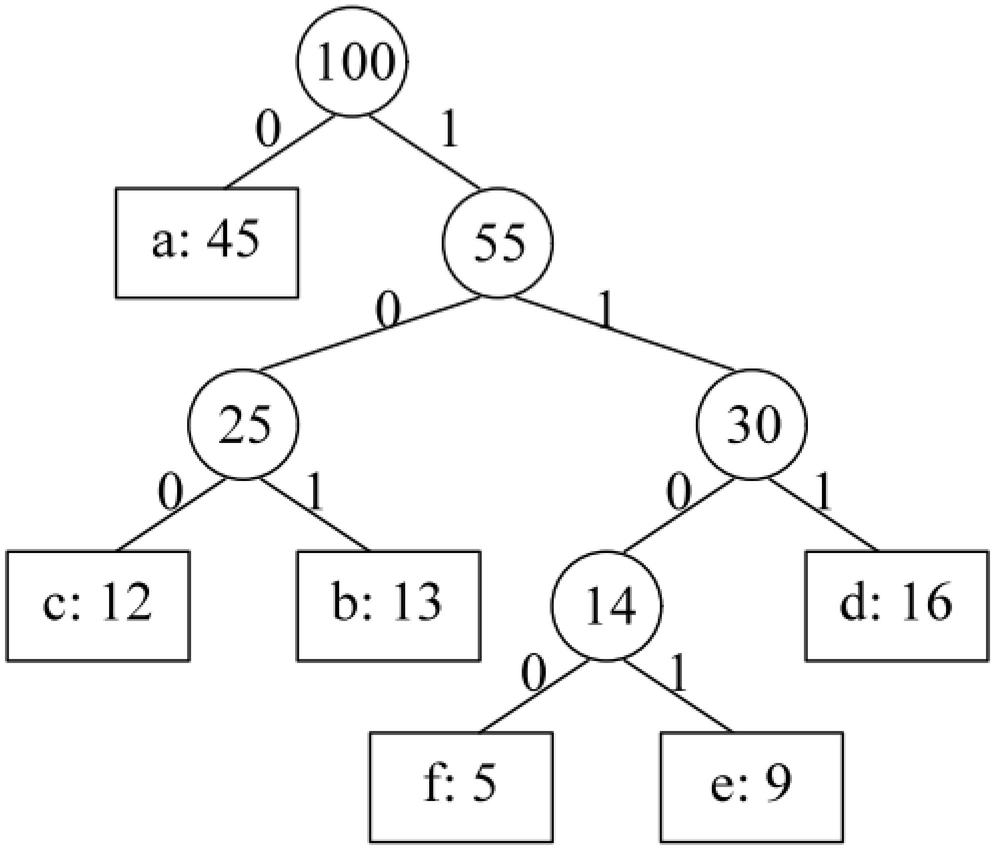


HUFFMAN CODE EXAMPLE



HUFFMAN CODE EXAMPLE

(f)



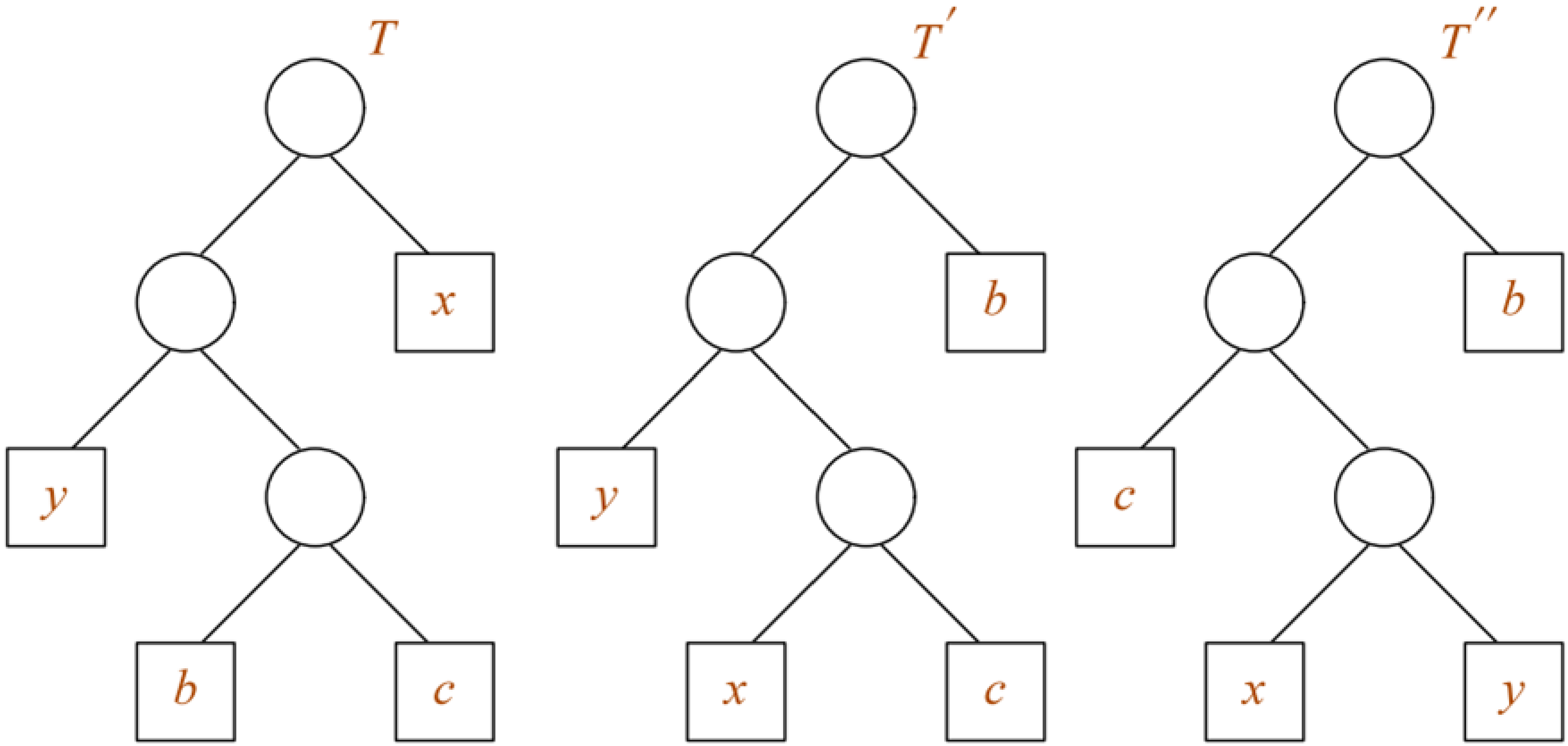
CORRECTNESS OF HUFFMAN'S ALGORITHM

- **We must show that the problem of determining an optimal prefix code exhibits**
  - Greedy choice property
  - Optimal substructure property
- **Lemma 1:**
  - Let x & 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 & 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 T'' representing another optimal code
    - Characters x & y appear as sibling leaves of max-depth in T''
  - Assume that $f(x)$ & $f(y)$ are two lowest leaf frequencies with $f(x) \leq f(y)$
  - Let $f(b)$ & $f(c)$ are two arbitrary leaf frequencies with $f(b) \leq f(c)$
  - Then, $f(x) \leq f(b)$ & $f(y) \leq f(c)$

CORRECTNESS OF HUFFMAN'S ALGORITHM



T => T' : exchange the positions of the leaves *b & x*
T' => T" : exchange the positions of the leaves *c & y*

CORRECTNESS OF HUFFMAN'S ALGORITHM

- **Proof of Lemma 1 (continued):**
- **The difference of the costs of *T* and *T'* is**
  - $B(T) - B(T') = \sum_{c \in C} \boldsymbol{f}(\boldsymbol{c})\boldsymbol{d}_{\boldsymbol{T}}(\boldsymbol{c}) - \sum_{c \in C} \boldsymbol{f}(\boldsymbol{c})\boldsymbol{d}_{\boldsymbol{T'}}(\boldsymbol{c})$

$$= f(\boldsymbol{x})d_T(\boldsymbol{x}) + f(\boldsymbol{b})d_T(\boldsymbol{b}) - f(\boldsymbol{x})d_{T'}(\boldsymbol{x}) - f(\boldsymbol{b})d_{T'}(\boldsymbol{b})$$

$$= f(\boldsymbol{x})d_T(\boldsymbol{x}) + f(\boldsymbol{b})d_T(\boldsymbol{b}) - f(\boldsymbol{x})d_T(\boldsymbol{b}) - f(\boldsymbol{b})d_T(\boldsymbol{x})$$

$$= f(\boldsymbol{b})(d_T(\boldsymbol{b}) - d_T(\boldsymbol{x})) - f(\boldsymbol{x})(d_T(\boldsymbol{b}) - d_T(\boldsymbol{x}))$$

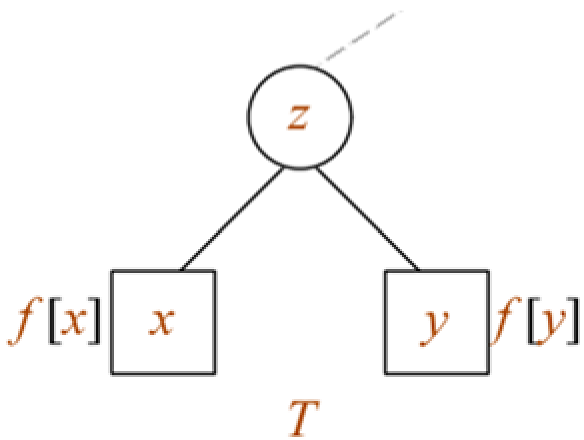
$$= (f(\boldsymbol{b}) - \boldsymbol{f}(\boldsymbol{x}))(d_T(\boldsymbol{b}) - d_T(\boldsymbol{x}))$$

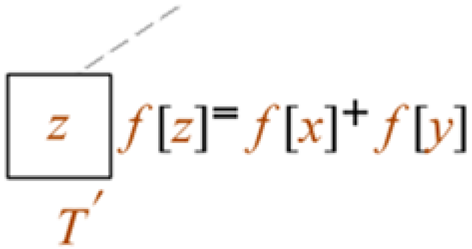
$$\geq \boldsymbol{0} \qquad since\, f(x) \leq f(b)\ \&\ d_T(x) \leq d_T(b)$$

  - Therefore $\boldsymbol{B}(\boldsymbol{T'}) \leq \boldsymbol{B}(\boldsymbol{T})$
- **Similarly we can show that** $\boldsymbol{B}(\boldsymbol{T'}) - \boldsymbol{B}(\boldsymbol{T''}) \geq \boldsymbol{0} \Rightarrow \boldsymbol{B}(\boldsymbol{T''}) \leq \boldsymbol{B}(\boldsymbol{T'})$
- **Together, they imply** $\boldsymbol{B}(\boldsymbol{T''}) \leq \boldsymbol{B}(\boldsymbol{T})$
- **Since *T* is assumed to be optimal** $\Rightarrow \boldsymbol{B}(\boldsymbol{T''}) = \boldsymbol{B}(\boldsymbol{T}) \Rightarrow$ ***T''*** **is also optimal**

CORRECTNESS OF HUFFMAN'S ALGORITHM

- **Lemma 2:**
  - Consider any two characters $\boldsymbol{x}$ & $\boldsymbol{y}$ that appear as sibling leaves in an optimal T and let $\boldsymbol{z}$ be their parent
  - Consider $\boldsymbol{z}$ as a character with frequency $\boldsymbol{f}(\boldsymbol{z}) = \boldsymbol{f}(\boldsymbol{x}) + \boldsymbol{f}(\boldsymbol{y})$
  - Then the tree $\boldsymbol{T}' = \boldsymbol{T} - \{\boldsymbol{x}, \boldsymbol{y}\}$ represents an optimal prefix code for the alphabet $\boldsymbol{C}' = \boldsymbol{C} - \{\boldsymbol{x}, \boldsymbol{y}\} \cup \{\boldsymbol{z}\}$





CORRECTNESS OF HUFFMAN'S ALGORITHM

- **Proof:**
  - Express cost of *T* in terms of cost of *T'*
  - For each $\boldsymbol{c} \in \boldsymbol{C} - \{\boldsymbol{x}, \boldsymbol{y}\}$ we have $d_T(\boldsymbol{c}) = d_{T'}(\boldsymbol{c})$
    - $\Rightarrow f(\boldsymbol{c}) d_T(\boldsymbol{c}) = f(\boldsymbol{c}) d_{T'}(\boldsymbol{c})$

  - $\boldsymbol{B}(\boldsymbol{T}) = \boldsymbol{B}(\boldsymbol{T}') + \boldsymbol{f}(\boldsymbol{x})\,(d_T(\boldsymbol{z}) + \mathbf{1}) + \boldsymbol{f}(\boldsymbol{y})\,(d_T(\boldsymbol{z}) + \mathbf{1}) - \boldsymbol{f}(\boldsymbol{z}) d_T(\boldsymbol{z})$
  - $\quad = \boldsymbol{B}(\boldsymbol{T}') + \boldsymbol{f}(\boldsymbol{z})\,(d_T(\boldsymbol{z}) + \mathbf{1}) - \boldsymbol{f}(\boldsymbol{z}) d_T(\boldsymbol{z})$
    - **Since** $\boldsymbol{f}(\boldsymbol{z}) = \boldsymbol{f}(\boldsymbol{x}) + \boldsymbol{f}(\boldsymbol{y})$
  - $\quad = \boldsymbol{B}(\boldsymbol{T}') + \boldsymbol{f}(\boldsymbol{z})$
  - $\quad = \boldsymbol{B}(\boldsymbol{T}') + \boldsymbol{f}(\boldsymbol{x}) + \boldsymbol{f}(\boldsymbol{y})$

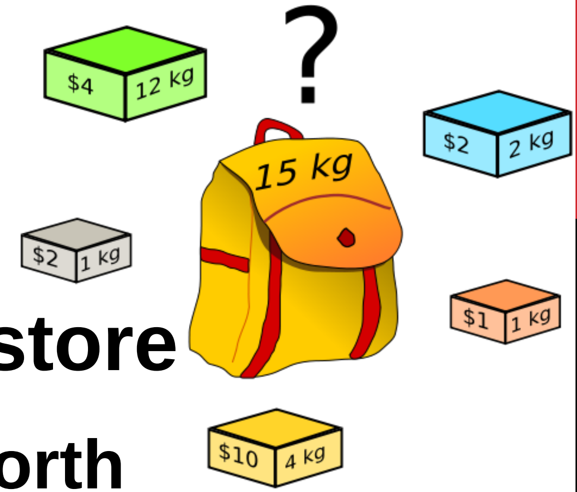
CORRECTNESS OF HUFFMAN'S ALGORITHM

- **Proof of Lemma 2 (continued):**
  - Assume *T'* represents a non-optimal prefix code for the alphabet *C'*
  - Then, ∃ a tree *T''* for *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* and *y* as children of *z* in *T''*
  - Then we obtain a prefix code for original alphabet *C* with cost $\boldsymbol{B(T'') + f(x) + f(y) < B(T') + f(x) + f(y) = \mathrm{B}(\mathrm{T})}$
  - Contradicting the optimality of *T*

CORRECTNESS OF HUFFMAN'S ALGORITHM

- **Lemma 1** tells us that optimal tree begins by merging the two **least-frequent** characters. This is the **greedy-choice** property used by Huffman's algorithm.
- **Lemma 2** shows that the problem has **optimal substructure** property.
- **Therefore, Huffman Encoding provides optimal encoding.**
- ***What are Zip, Rar, 7z, etc. doing then?***
- ***What about WinZip vs. gzip?***
- ***What about binary files?***

KNAPSACK PROBLEM



- **There are *n* different items in a store**
- **Item *i* weighs w_i kilograms and is worth $\$b_i$**
- **We can carry up to *W* kilograms in a knapsack**
- **An item must be taken as a whole or left behind.**
- **Problem: What should we take to maximize the total value ?**

0-1 VS. FRACTIONAL KNAPSACK

- **0-1 Knapsack Problem:**
  - Items cannot be divided
  - We must take either the entire item or leave it behind
- **Fractional Knapsack Problem:**
  - We can take partial items
    - e.g., items are liquids or powders
  - Solvable with a greedy algorithm since the problem has
    - Greedy-choice property
    - Optimal substructure property

OPTIMAL SUB-STRUCTURE

- **Both knapsack flavors exhibit the optimal sub-structure property**
- **0-1 Knapsack Problem (S,W):**
  - Consider a most valuable load (optimal solution) 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 $W_j' = W - w_j$ kilograms that we can take from the set of remaining items $S_j' = S - \{ I_j \}$
  - That is, L_j' should be an optimal solution to the 0-1 Knapsack Problem (S_j', W_j')

OPTIMAL SUBSTRUCTURE

- **Fractional Knapsack Problem (S,W):**
  - Consider a most valuable load L where $W_L \leq W$

OPTIMAL SUBSTRUCTURE

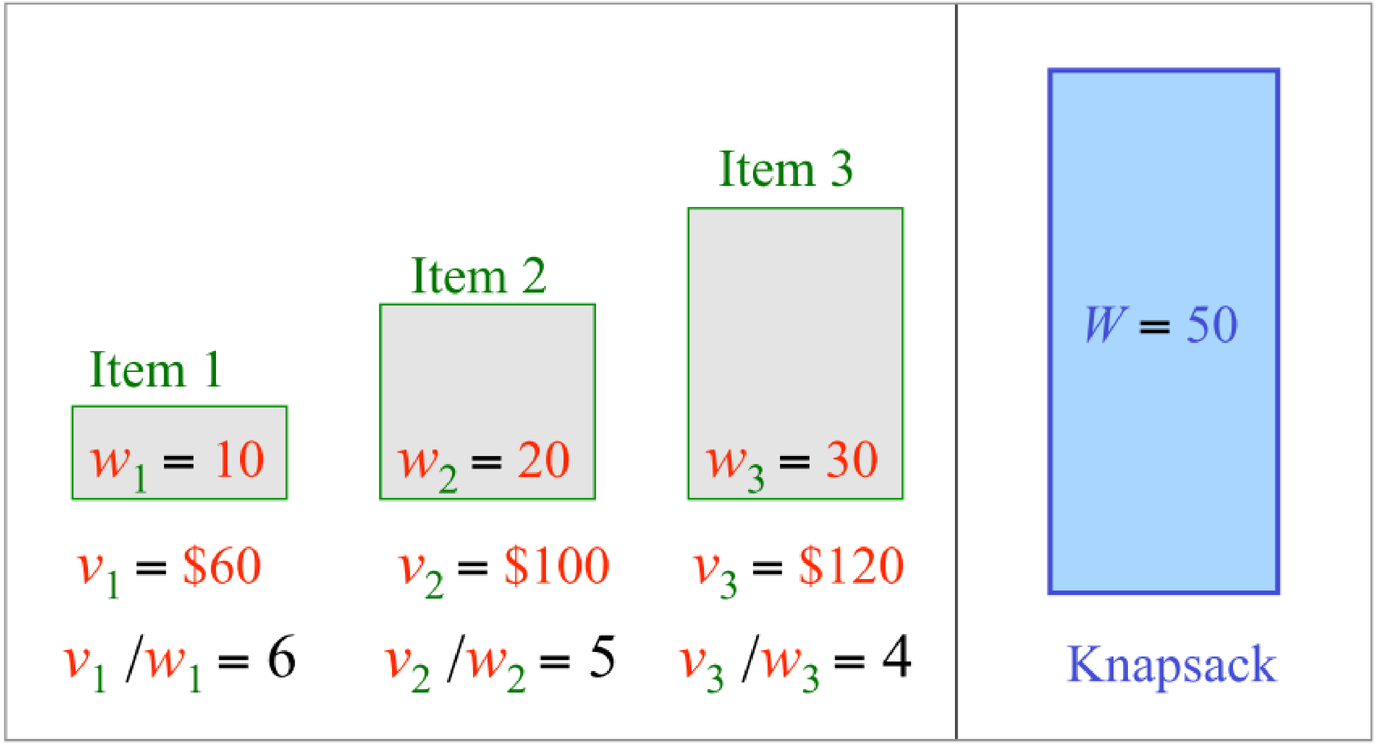
- **Fractional Knapsack Problem (S,W):**
  - Consider a most valuable load L where $W_L \leq W$
  - If we remove some item partially, i.e., a weight $0 \leq w \leq w_j$ of item j, from optimal load L
  - The remaining load $L_j' = L - \{ w \text{ kilograms of } I_j \}$ must be a most valuable load weighing at most $W_j' = W - w$ kilograms that we can take from the set of remaining items $S_j' = S - \{ I_j \} \cup \{ w_j\text{-}w \text{ kilograms of } I_j \}$
  - That is L_j' should be an optimal solution to the Fractional Knapsack Problem (S_j', W_j')

FRACTIONAL KNAPSACK PROBLEM

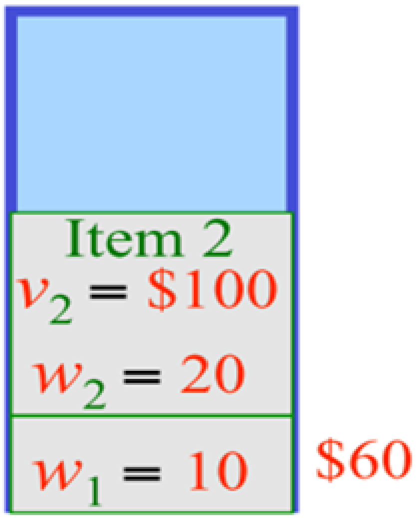
- **The optimal solution to the fractional knapsack problem can be found with a greedy algorithm**
- ***Algorithm?***
  - sort items in decreasing order of value per kilogram
  - while limit of W kilograms is not reached do
    - consider the next item in sorted list
    - take as much as possible (all there is or as much as will fit)
- ***Runtime?***
  - O(n log n) running time (limiting factor is the sort)

0-1 KNAPSACK PROBLEM

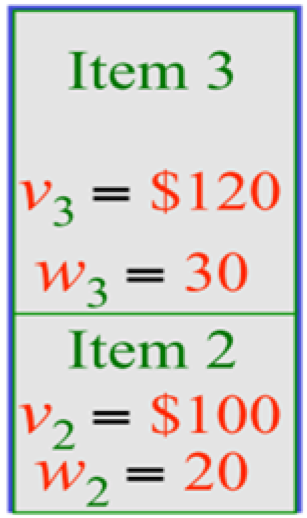
- The optimal solution to the 0-1 knapsack problem cannot be found using this greedy strategy



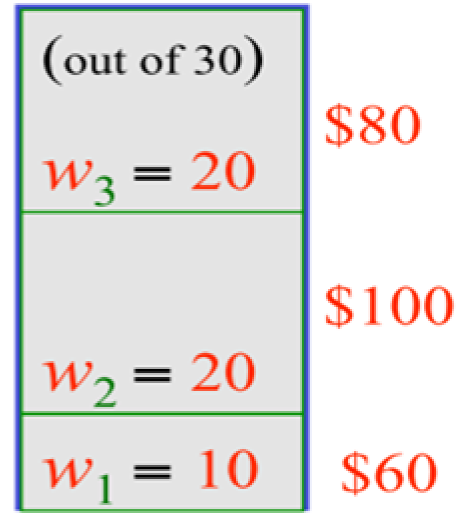
0-1 KNAPSACK PROBLEM



Greedy Solution
Total worth = $160



Optimal 0-1 Solution
Total worth = $220



Optimal Fractional Solution using the greedy algorithm
Total worth = $240

Different problems, incomparable solutions