Greedy Algorithms

Greedy Algorithm

- Like dynamic programming, used to solve optimization problems.
- Problems exhibit optimal substructure (like DP).
- Problems also exhibit the **greedy-choice** property.
  - » When we have a choice to make, make the one that looks best *right now*. (i.e., locally)
  - » Make a **locally optimal choice** in hope of getting a **globally optimal solution**.

Greedy Technique

- Constructs a solution to an optimization problem piece by piece through a sequence of choices that are:
  - » feasible, i.e., it has to satisfy the problem's constraints
  - » locally optimal, i.e., it has to be the best local choice among all feasible choices available on that step
  - » Irrevocable, i.e., once made, it cannot be changed on subsequent steps of the Algorithm
- For some problems, yields an optimal solution.
- For most, does not but can be useful for fast approximations.

Applications of the Greedy Strategy

- Optimal solutions:
  - » minimum spanning tree (MST)
  - » single-source shortest paths
  - » simple scheduling or activity selection problems
  - » Huffman codes

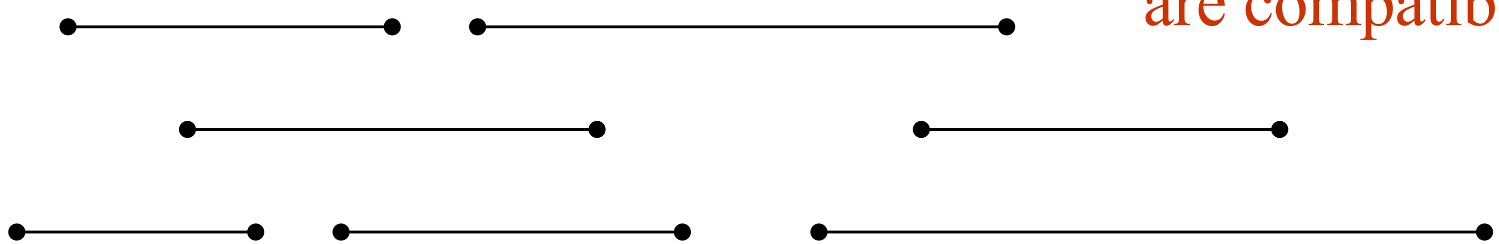
- Approximations:
  - » traveling salesman problem (TSP)
  - » knapsack problem
  - » other combinatorial optimization problems

Activity-Selection Problem

- Input: Set S of n activities, a_1, a_2, …, a_n.
  - » s_i = start time of activity i.
  - » f_i = finish time of activity i.
- Output: Subset A of maximum number of compatible activities.
  - » Two activities are compatible if their intervals don't overlap.

Example:

Activities in each line are compatible.



Optimal Substructure

- Assume activities are sorted by finishing times.
  - » $f_1 \leq f_2 \leq \ldots \leq f_n$.
- Suppose an optimal solution includes activity a_k.
  - » This generates two subproblems.
  - » Selecting from $a_1, \ldots, a_{k-1}$, activities compatible with one another, and that finish before a_k starts (compatible with a_k).
  - » Selecting from $a_{k+1}, \ldots, a_n$, activities compatible with one another, and that start after a_k finishes.
  - » The solutions to the two subproblems must be optimal.
    - Prove using the cut-and-paste approach.

Recursive Solution

- Let S_{ij} = subset of activities in S that start after a_i finishes and finish before a_j starts.
- Subproblems: Selecting maximum number of mutually compatible activities from S_{ij}.
- Let $c[i, j]$ = size of maximum-size subset of mutually compatible activities in S_{ij}.

Recursive Solution:

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

Greedy-Choice Property

- The problem also exhibits the greedy-choice property.
  - » There is an optimal solution to the subproblem S_{ij} that includes the activity with the smallest finish time in set S_{ij}.(intuition: this leaves more time or resource for other tasks)
  - » Can be proved easily.

- Hence, there is an optimal solution to S that includes a_1.
- Therefore, Greedy algorithm is: earliest finish time first.
  - » Make this greedy choice without solving subproblems first.
  - » Solve the subproblem resulted from this greedy choice.
  - » Combine the greedy choice and solution to the subproblem.

This is a top-down fashion instead of bottom-up!

Recursive Algorithm

Recursive-Activity-Selector (s, f, i, j)

1. $m \leftarrow i+1$

2. **while** $m < j$ and $s_m < f_i$

3. **do** $m \leftarrow m+1$

4. **if** $m < j$

5. **then return** $\{a_m\} \cup$ Recursive-Activity-Selector(s, f, m, j)

6. else return ϕ

Initial Call: Recursive-Activity-Selector (s, f, 0, n+1)

Complexity: Θ(n), considering already sorted based on finish time.

Straightforward to convert the algorithm to an iterative one.

Typical Steps

- Cast the optimization problem as one in which we make a choice and are left with one subproblem.
- Show that greedy choice and optimal solution to subproblem $\Rightarrow$ optimal solution to the problem.
- Make the greedy choice and **solve top-down**.
  - » E.g., put an activity in optimal solution, then solve a smaller problem.
- May have to preprocess input to put into greedy order.
  - » Example: Sorting activities by finish time.

Elements of Optimal Greedy Alg

- Greedy-choice Property.
  - » A globally optimal solution can be arrived at by making a locally optimal (greedy) choice.
- Optimal Substructure.

0/1 Knapsack Problem

- Given n items of

  integer weights: $w_1 \quad w_2 \; \ldots \; w_n$

  values: $v_1 \quad v_2 \; \ldots \; v_n$

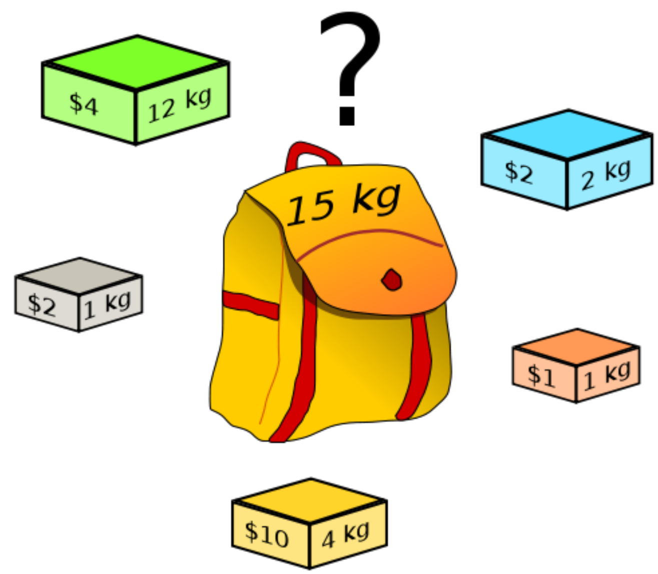
  a knapsack of integer capacity W.

- Find most valuable subset of the items that fit into the knapsack.

- You take an item (1) or

do not take (0) → cannot take

A fraction of an item.



Optimal Substructure

- *Let F(i, j)* be the value of an optimal solution i.e., the value of the most valuable subset of the first *i items that fit into* the knapsack of capacity *j*.
- We can divide all the subsets of the first *i* items that fit the knapsack of capacity *j* into two categories:

  » those that do not include the *i-th* item

  » and those that do.

Our goal is to find F(n, W)

Optimal Substructure

1. Among the subsets that do not include the *i-th item, the value of an optimal* subset is, by definition, $F(i - 1, j)$

2. Among the subsets that do include the *i-th item*

(hence, $j - w_i \geq 0$), an optimal subset is made up of this item and an optimal subset of the first $i - 1$ *items* that fits into the knapsack of capacity $j - w_i$.

The value of such an optimal subset is $v_i + F(i - 1, j - w_i)$

- These observations lead to the following recurrence:

$$F(i, j) = \begin{cases} \max\{F(i-1, j), v_i + F(i-1, j-w_i)\} & \text{if } j - w_i \geq 0, \\ F(i-1, j) & \text{if } j - w_i < 0. \end{cases}$$

It is convenient to define the initial conditions as follows:

$$F(0, j) = 0 \text{ for } j \geq 0 \quad \text{and} \quad F(i, 0) = 0 \text{ for } i \geq 0.$$

Greedy Alg for 0/1 and Fractional

- Greedy choice does not work for 0/1 knapsack
  - » Does not exhibit greedy choice property
  - » Need to find optimal solution using DP (Use table of $\boldsymbol{n}$x$\boldsymbol{W}$)
- Fractional knapsack: Can take any fraction of item
  - » Has greedy choice property.
  - » Optimal solution: take items in decreasing order of unit value: O(n log n) time.

Greedy Alg for 0/1 and Fractional

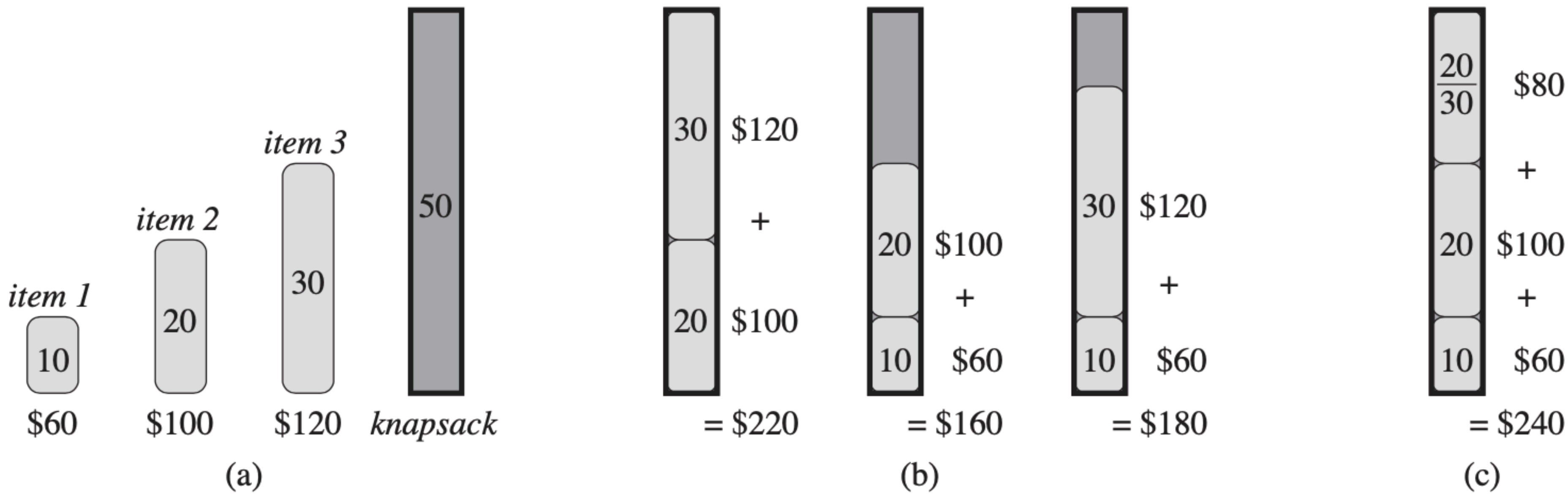


Figure 16.2 An example showing that the greedy strategy does not work for the 0-1 knapsack problem. **(a)** The thief must select a subset of the three items shown whose weight must not exceed 50 pounds. **(b)** The optimal subset includes items 2 and 3. Any solution with item 1 is suboptimal, even though item 1 has the greatest value per pound. **(c)** For the fractional knapsack problem, taking the items in order of greatest value per pound yields an optimal solution.

Data Compression Problem

- Input: A file of characters from set $C = \{c_1, c_2, \ldots, c_n\}$.
  - » $f(c_i)$ denotes the frequency (number of times) that c_i appears in the input file.
- Output: Binary character encoding for C that minimizes the file size.
  - » Encoding may be variable-length:
    - Example : `a=0`, `b=101;`

Example: Optimal Data Compression

c_i	a	b	c	d	e	f
Frequency $f(c_i)$ (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

File-Size:

- Fixed-length:

$$3 \cdot (45 + 13 + 12 + 16 + 9 + 5) \cdot 1000 = 300{,}000 \text{ bits}$$

- Variable-length:

$$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1000 = 224{,}000 \text{ bits}$$

Example: Optimal Data Compression

c_i	a	b	c	d	e	f
Frequency $f(c_i)$ (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

Prefix(free) Code:
No code is a prefix of another code.

File-Size:

- Fixed-length:

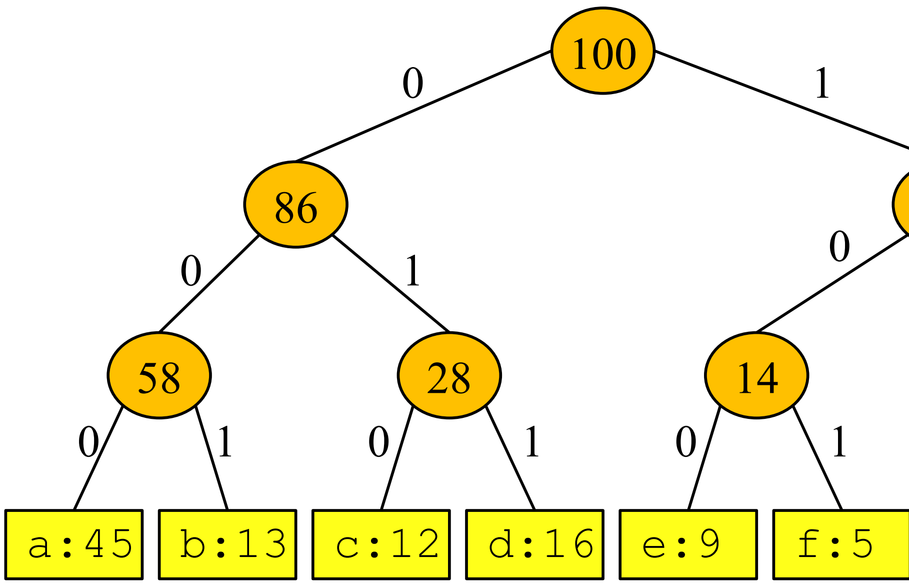
$$3 \cdot (45 + 13 + 12 + 16 + 9 + 5) \cdot 1000 = 300{,}000 \text{ bits}$$

- Variable-length:

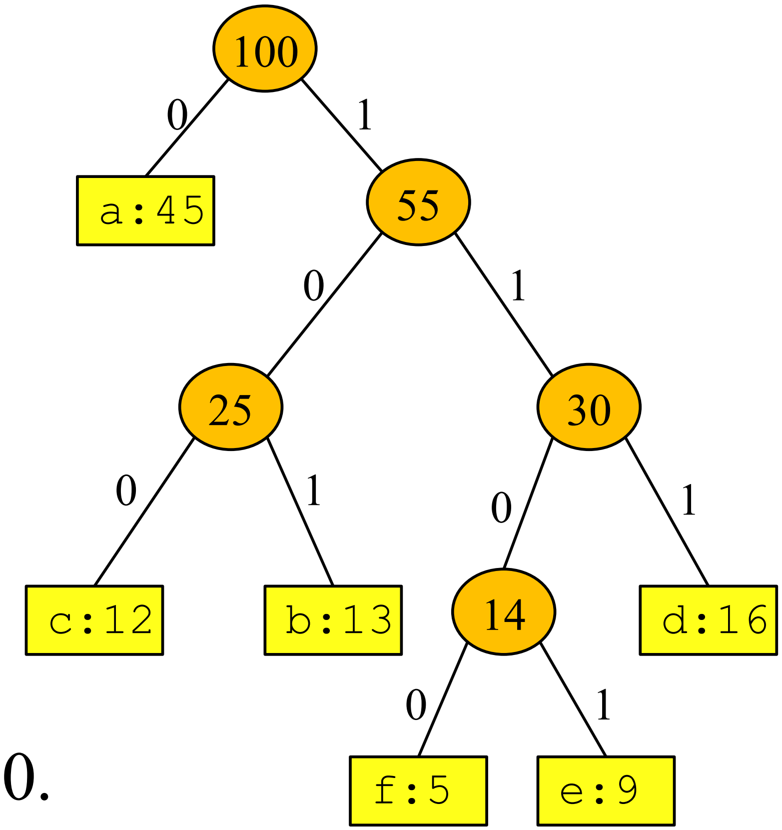
$$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1000 = 224{,}000 \text{ bits}$$

Decoding

Fixed-length

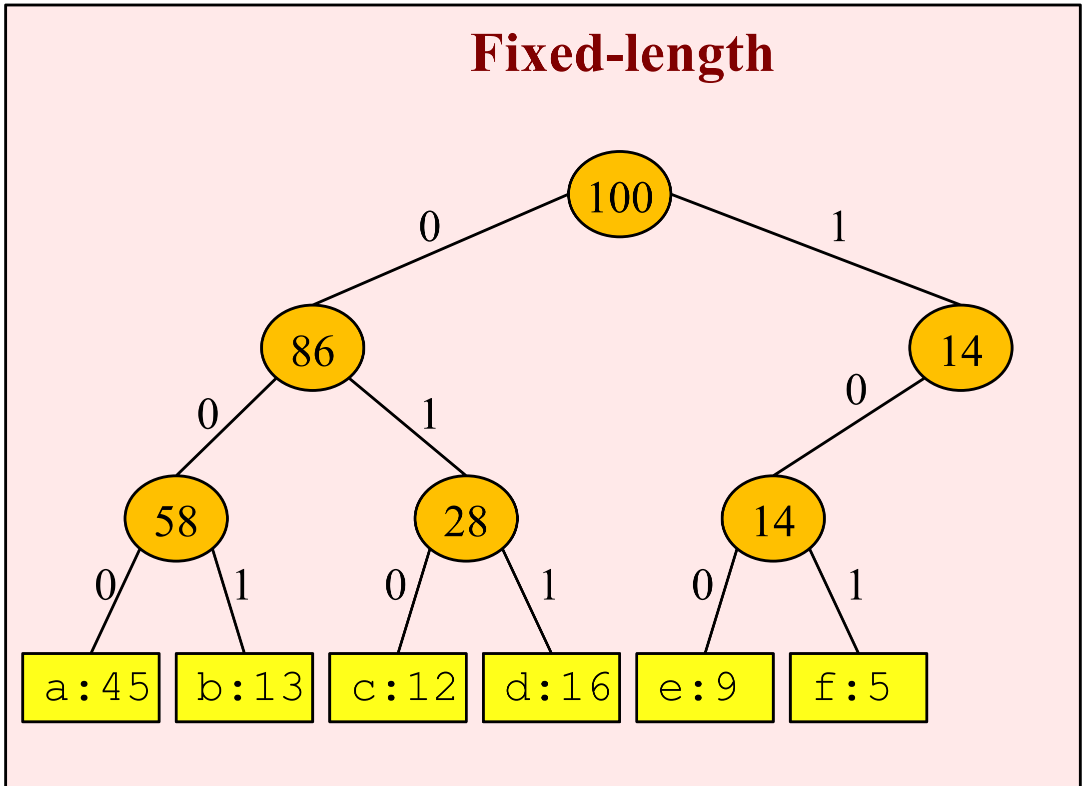


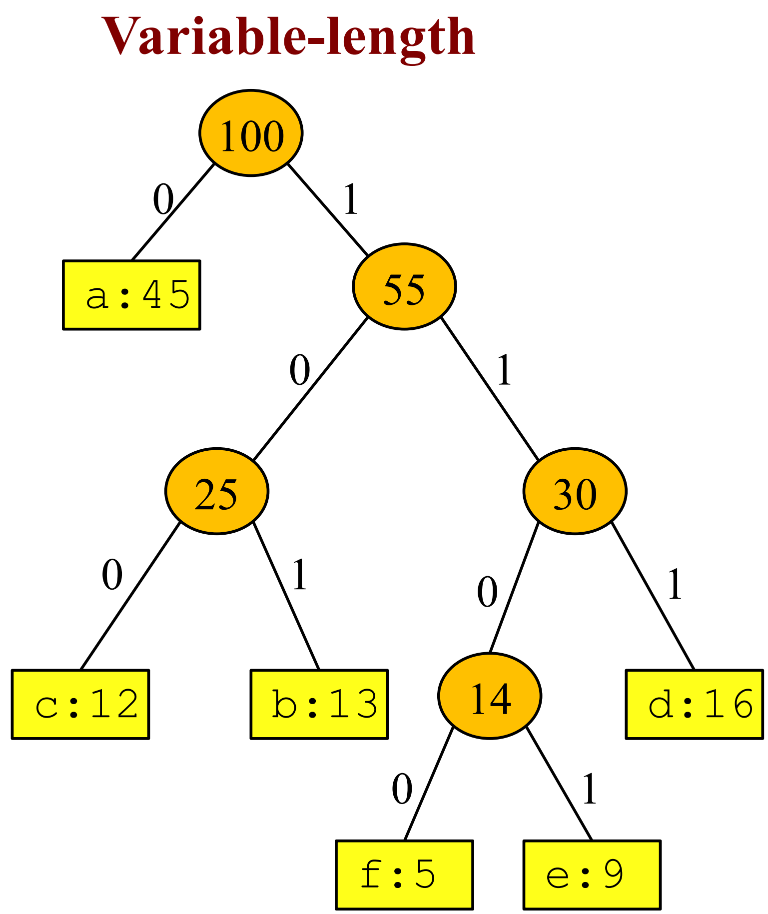
Variable-length



"Going left" in tree corresponds to 0.
"Going right" in tree corresponds to 1.

Decoding

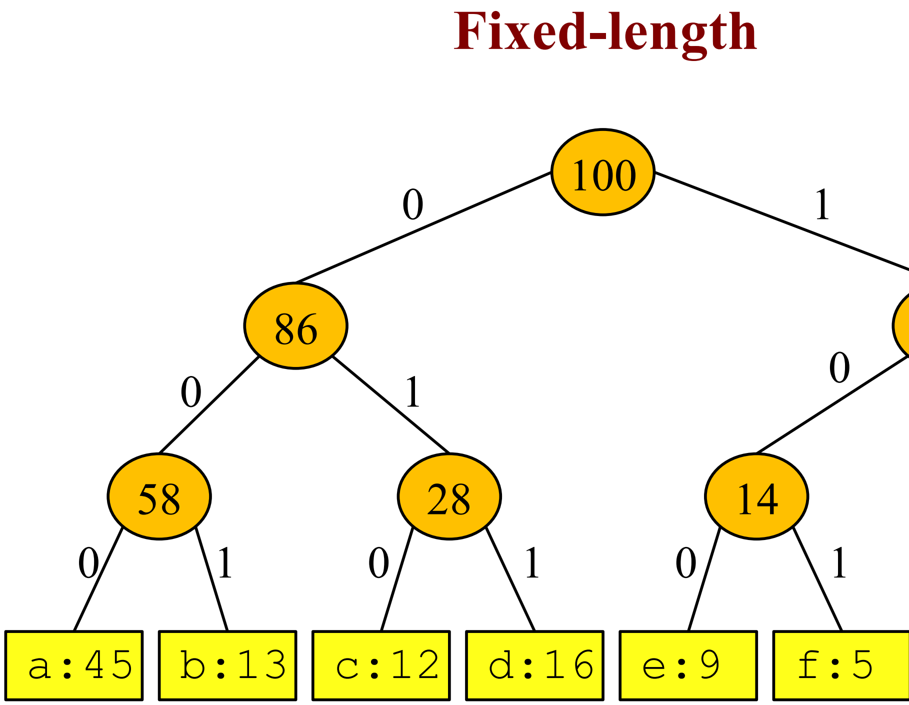


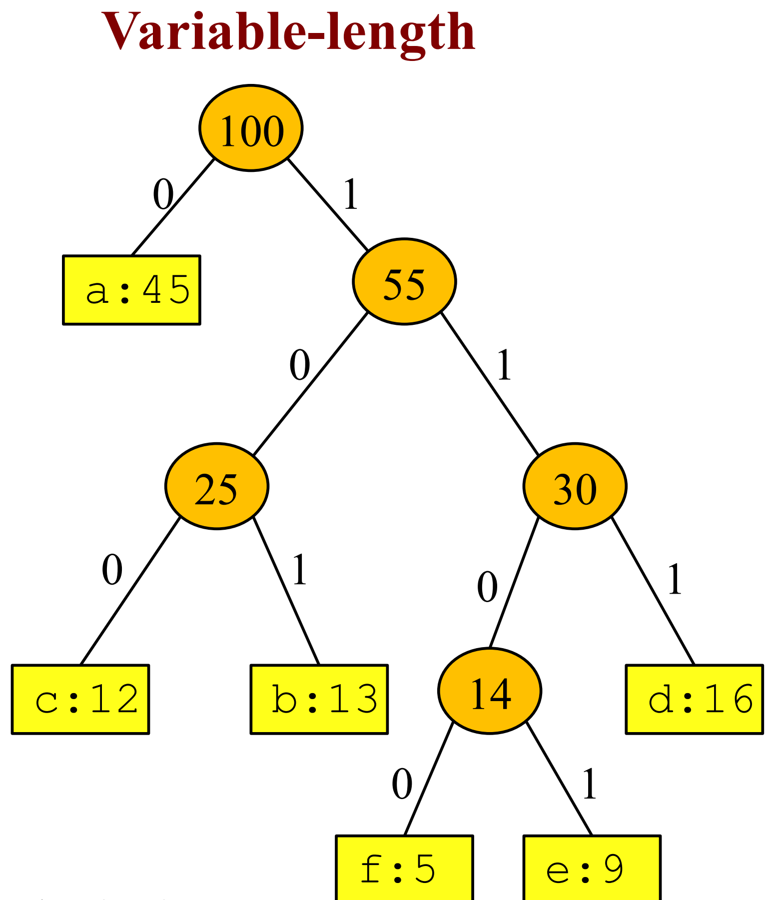


Not full tree – can't be optimal encoding.
Why?

-- Missing right most child indicates we could have a shorter unique code, say 11.

Decoding





Cost of Encoding for Tree T:

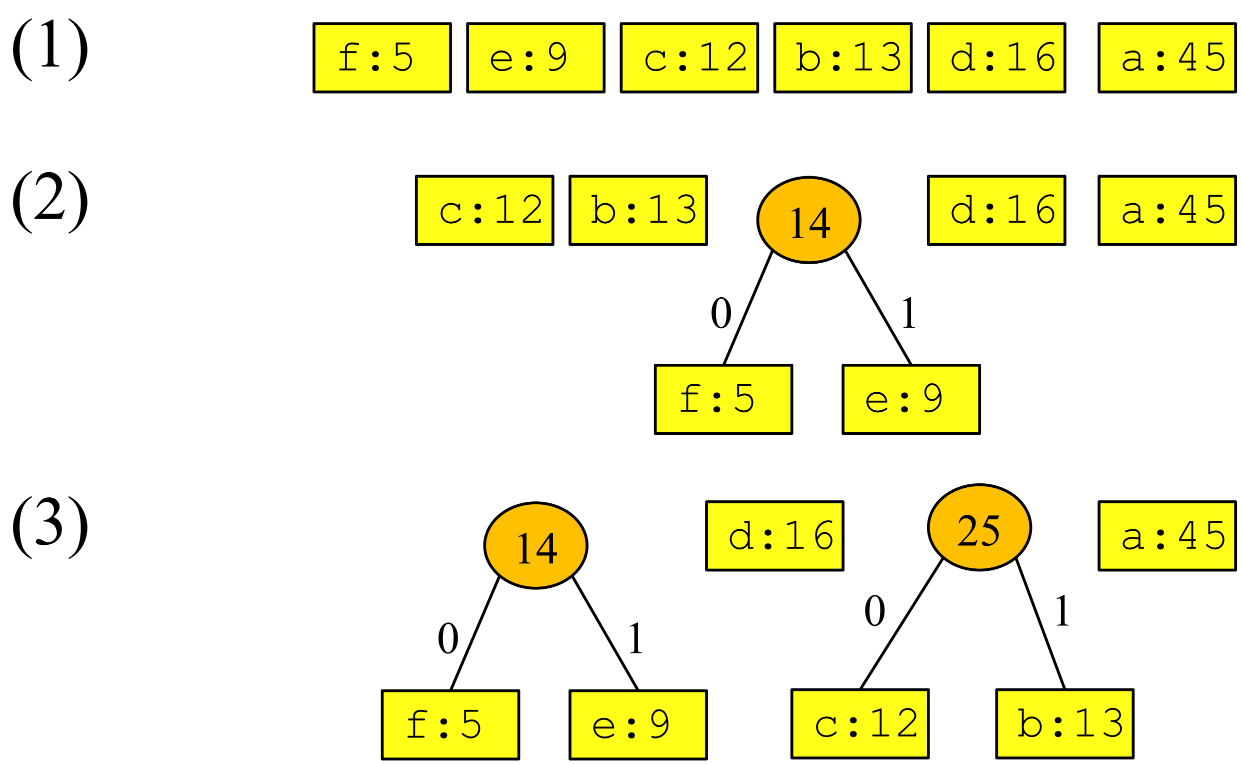
$$B(T) = \sum_{c_i \in C} f(c_i) d_T(c_i)$$

Depth of c_i in tree T.

Solution: Huffman Codes

- Idea: Sort characters in monotonically nondecreasing order and "merge" two least-frequently-used characters into a subtree; repeat until all characters are in tree.

Step:

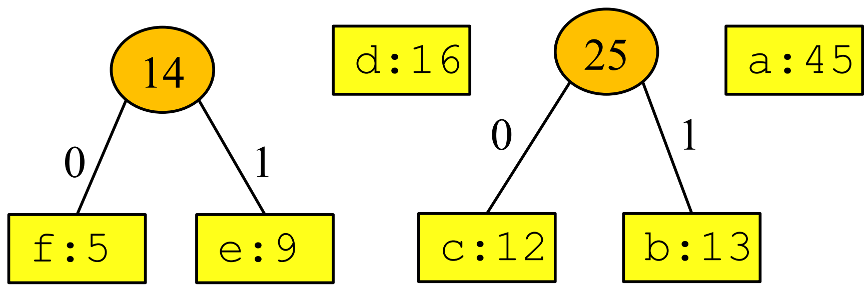


Solution: Huffman Codes

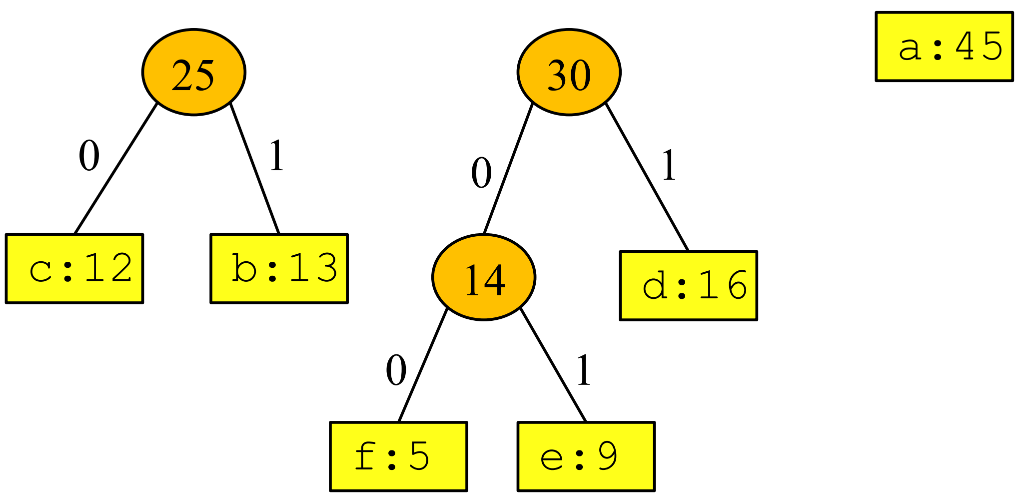
- Idea: Sort characters in monotonically nondecreasing order and "merge" two least-frequently-used characters into a subtree; repeat until all characters are in tree.

Step:

(3)



(4)

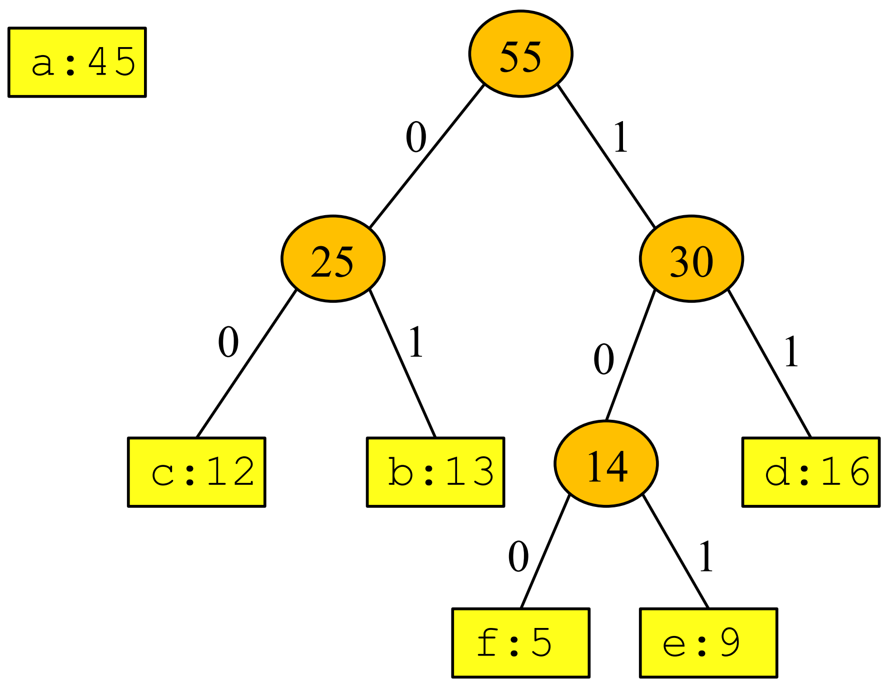


Solution: Huffman Codes

- Idea: Sort characters in monotonically nondecreasing order and "merge" two least-frequently-used characters into a subtree; repeat until all characters are in tree.

  Step:

  (5)

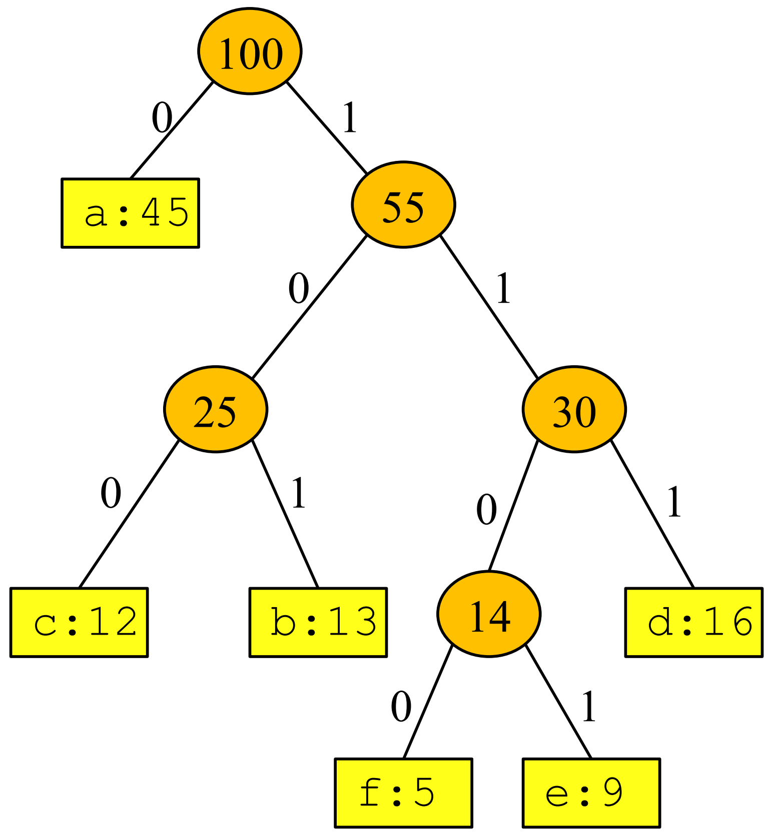


Solution: Huffman Codes

- Idea: Sort characters in monotonically nondecreasing order and "merge" two least-frequently-used characters into a subtree; repeat until all characters are in tree.

  Step:

  (6)



Solution: Huffman Codes

Huffman (C)

```
n ← |C|
Q ← C                //Make Min-Heap    ← O(n)
for i ← 1 to n - 1
    do allocate a new node z
        left[z] ← x ← Extract-Min(Q)    ← O(lg n)
        right[z] ← y ← Extract-Min(Q)
        f[z] ← f[x] + f[y]
        Insert(Q, z)
return Extract-Min(Q)   //return root of tree
```

What is running time?

Answer: O(n lg n).

Recall, we want to minimize:

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

Why does this approach yield minimum B(T)?

Why is it greedy?

Huffman Codes: Proof of Correctness

Lemma 1 ← "Greedy Choice" Property

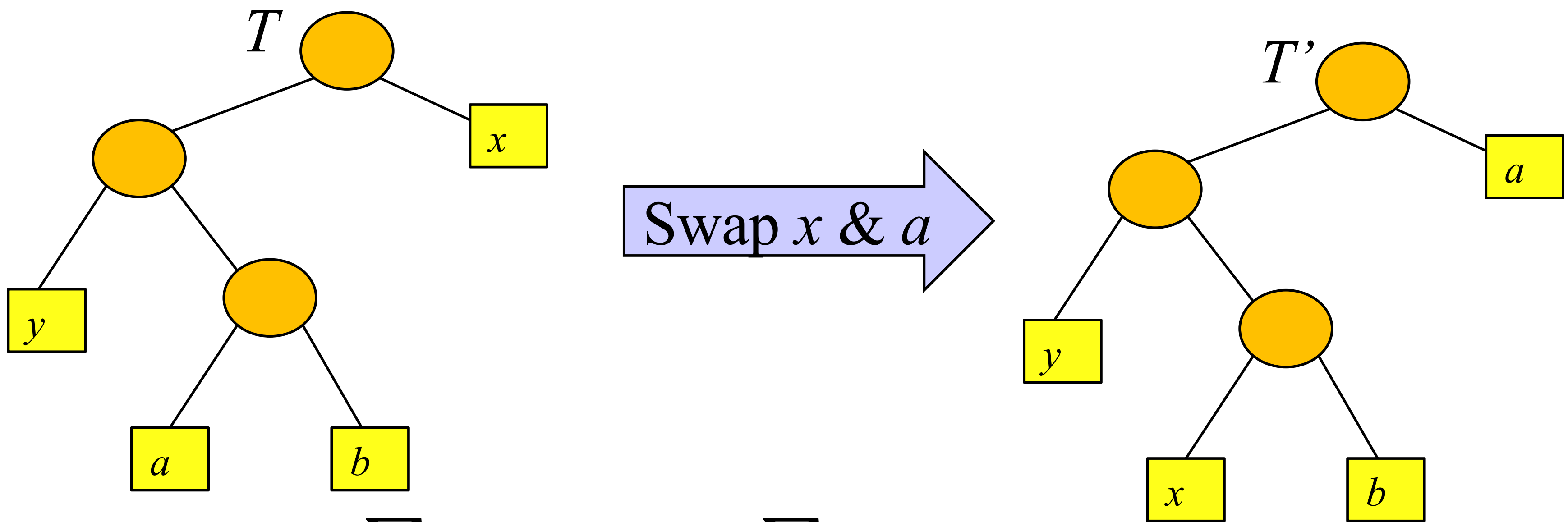
Let C be set of characters where each $c \in C$ has frequency $f(c)$. Let x and y be two characters having lowest frequencies. Then there exists an optimal prefix code for C in which the binary codewords for x and y have the same length and differ only in the last bit.

Proof: Let T represent the tree for any optimal encoding of C. We will show we can modify any such T to create new tree T'', such that x and y are sibling leaves at maximum depth in T'' and $B(T) \geq B(T'')$.

Why does showing this prove the lemma?

Huffman Codes: Proof of Correctness

Proof (cont.): Let a and b be nodes at maximum depth in T and x has a lowest frequency. For example,



So, $B(T) - B(T') = \sum_{c_i \in C} f(c_i)d_T(c_i) - \sum_{c_i \in C} f(c_i)d_{T'}(c_i)$

$$= f(x)d_T(x) + f(a)d_T(a) - f(x)d_{T'}(x) - f(a)d_{T'}(a)$$

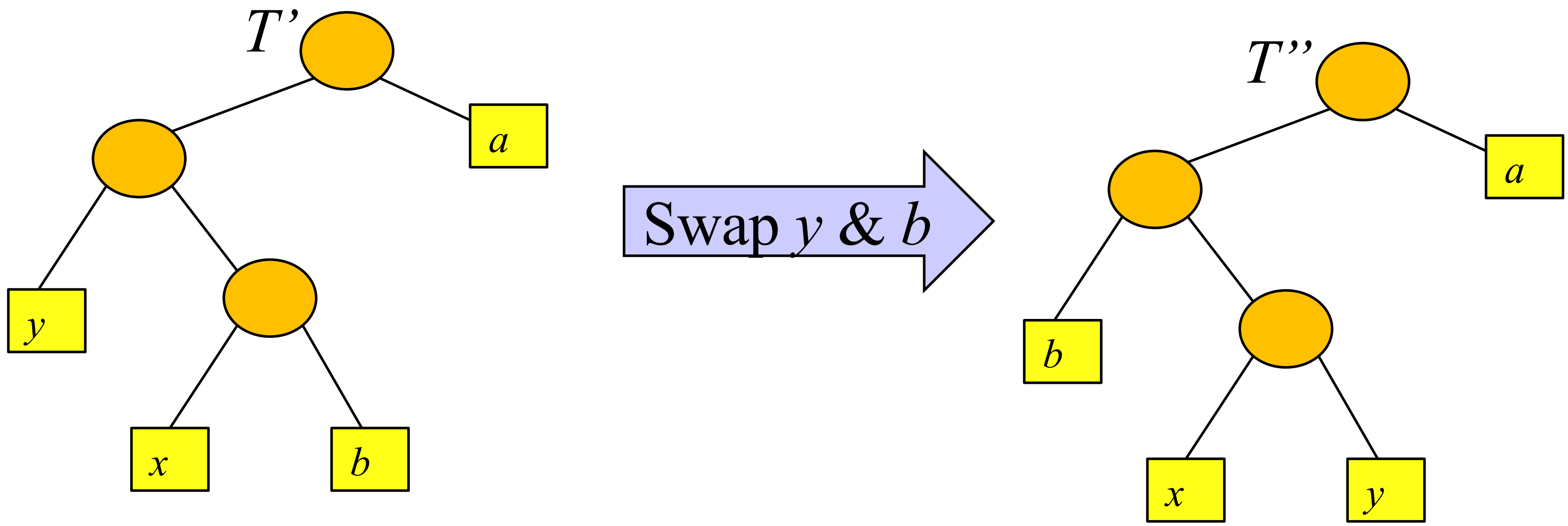
$$= f(x)d_T(x) + f(a)d_T(a) - f(x)d_T(a) - f(a)d_T(x)$$

$$= (f(a) - f(x))(d_T(a) - d_T(x))$$

$$\geq 0$$

Huffman Codes: Proof of Correctness

Proof (con't): Similarly,



and show that $B(T') - B(T'') \geq 0$.

Thus, $B(T) - B(T'') \geq 0$. ■

Huffman Codes: Proof of Correctness

Lemma 2 ← "Optimal Substructure" Property

Let C be set of characters, where each $c \in C$ has frequency $f(c)$. Let x and y be two characters having the lowest frequencies. Let $C' = C - \{x, y\} \cup \{z\}$, where z is a "new" character with $f(z) = f(x) + f(y)$. Let T' be any tree representing an optimal prefix code for C'. Then the tree T obtained from T' by replacing z with the subtree containing x and y represents an optimal prefix code for C.

Proof Sketch: First, write $B(T)$ in terms of $B(T')$. Then, assume that T does not represent an optimal prefix code for C to derive a contradiction.

Huffman Codes: Proof of Correctness

> ***Theorem***
>
> Procedure `Huffman` produces an optimal prefix code.

Proof: Follows immediately from Lemmas 1 & 2.

Matroid and Greedy Algs

- A matroid is a mathematical structure that generalizes the notion of linear independence from vector spaces to arbitrary sets.

- If an optimization problem has the structure of a matroid, then the appropriate greedy algorithm will solve it optimally

Matroid

A ***matroid*** is an ordered pair $M = (S, \mathcal{I})$ satisfying the following conditions.

1. S is a finite set.
2. $\mathcal{I}$ is a nonempty family of subsets of S, called the ***independent*** subsets of S, such that if $B \in \mathcal{I}$ and $A \subseteq B$, then $A \in \mathcal{I}$. We say that $\mathcal{I}$ is ***hereditary*** if it satisfies this property. Note that the empty set $\emptyset$ is necessarily a member of $\mathcal{I}$.
3. If $A \in \mathcal{I}$, $B \in \mathcal{I}$, and $|A| < |B|$, then there exists some element $x \in B - A$ such that $A \cup \{x\} \in \mathcal{I}$. We say that M satisfies the ***exchange property***.