Greedy Algorithm

Textbook, Chapter 17, Sections 17.2 and 17.2

CSCE310: Data Structures and Algorithms

`www.cse.unl.edu/~choueiry/S01-310/`

Berthe Y. Choueiry (Shu-we-ri)
Ferguson Hall, Room 104
`choueiry@cse.unl.edu`, Tel: (402)472-5444

For many optimization problems

Dynamic Programming may be an overkill

- Its performance depends on the number of subproblems
- Choice of subproblem remains an art

Greedy algorithms always make the choice that

- looks best at the moment
- Locally optimal choices
- We need not know solutions to subproblems to make a choice
- Sometimes yield globally optimal solutions

Greedy algorithms: Applicability

1. **Optimal substructure:** optimal solution contains optimal subsolutions
2. **Greedy choice property:** optimal solution can be obtained by making the 'greedy' choice at each step

Outline

1. Activity-selection problem: optimal solution with a greedy algorithm
2. Basic elements of the greedy strategy
   illustration on the knapsack problem
3. Design of data-compression code: optimal solution with a greedy algorithm, Huffman code

Many algorithms can be viewed as applications of the greedy method: coloring of interval graphs, minimum-spanning tree, etc.

Activity-selection Problem

Given:

- A set $S = \{1, 2, \ldots, n\}$ activities
- Each activity i starts at s_i (start time) and finishes at f_i (finish time)
  Naturally, $s_i \leq f_i$
- Activities need to be scheduled on a single resource: resource has capacity one, is non-sharable

Two activities i, j are compatible if they do not overlap: $s_i \geq f_j$ (j before i) or $s_j \geq f_i$ (i before j)

Find: the maximum set of mutually compatible activities

Activities are fixed in time $\longrightarrow$ a resource allocation problem (vs. scheduling problem)

Greedy algorithm for Activity-selection Problem

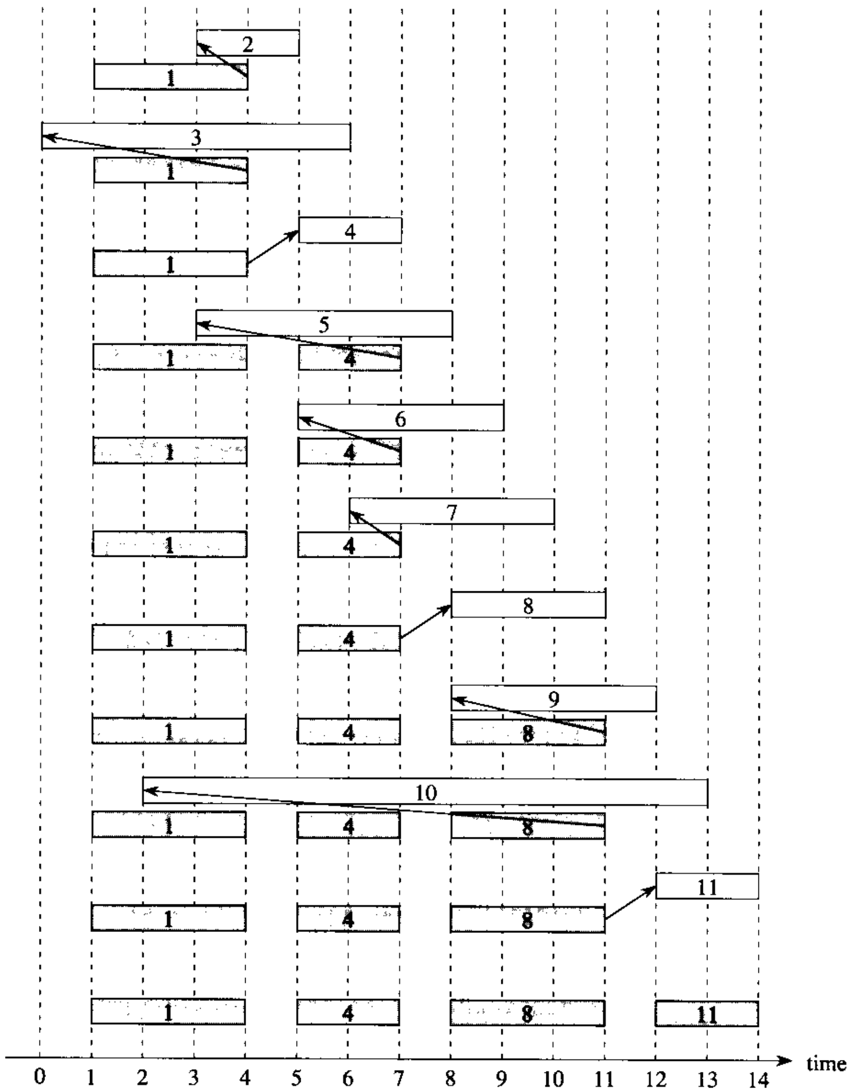
- Assume: activities ordered by increasing finishing time:
  $f_1, f_2, f_3, \cdots, f_n$
  otherwise, can be sorted in $O(n \lg n)$
- s, f represented as arrays

```
GREEDY-ACTIVITY-SELECTOR(s, f)
n ← length[s]
A ← {1}
j ← 1
for i ← 2 to n
     do if s_i ≥ f_j
           then A ← A ∪ {i}
                j ← i
return A
```

Greedy algorithm for Activity-selection Problem

- A collects selected activities
- j specifies most recent addition to A
- f_j is the maximum finishing time of activities in A:
  $f_j = \max\{f_k \ : k \in A\}$
- Lines 2-3: select activity 1, initialize A, j
- Lines 4-7: consider each activity in turn and add it to A is it is compatible with the last activity in A, this compatible with all activities in A
- Lines 6-7: add the first such compatible activity to A and update j

i	s_i	f_i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14



`Greedy-Activity-Selector(s,f)`

- Assuming all activities were sorted, $\Theta(n)$
- Greedy: it leaves as much opportunity as possible for scheduling remaining activities
- Greedy choice: maximizes amount of unscheduled time remaining
- Theorem 17.1: `Greedy-Activity-Selector` produces optimal solution (i.e., maximum size) for activity-selection problem

Proof: `Greedy-Activity-Selector` finds optimal solution

Very informally:

- Activity 1 has the earliest finish time
- Greedy choice is activity 1
- **Goal:** prove there is an optimal solution that begins with a greedy choice, activity 1
- Suppose A is an optimal solution ($A \subseteq S$), elements in A ordered by increasing finish time
- Either $\texttt{First}(A) = 1 \longrightarrow A$ starts with the greedy choice or $\texttt{First}(A) = k$, and $k \neq 1$

- Suppose $k \neq 1$, and prove there is another optimal solution B starting with the greedy choice 1
- Since $f_1 \leq f_k$, I can add activity 1 to $A - \{k\}$ and all activities remain disjoint
- Let $B = A - \{k\} \cup \{1\}$ (same activities as in A plus activity 1 and except activity k), all activities in B are disjoint and B has the same number of activities as $A \longrightarrow B$ is an optimal solution
- Since f_1 is the smallest f_i, B is an optimal solution starting with activity 1
- There exists an optimal schedule starting with a greedy choice

- Make greedy choice, problem becomes: find optimal solutions for the activity-selection problem of activities in S <u>that are compatible with activity 1</u>
- Prove: A is optimal solution to $S \Rightarrow A' = A - \{1\}$ is optimal solution to $S' = \{i \in S \ : s_i \geq f_1\}$, which is the set of activities that are compatible with activity 1
- Suppose we could find a solution B' to S' with more activities than A'. Lets add activity 1 to B'.
- This yields a solution B to S with more activities than A.
- Contradicts optimality of A
- So, after each greedy choice, we encounter a similar optimization problem
- By induction on number of choices made, greedy choice at every step yields optimal solution

Greedy strategy: elements

- A greedy algorithm makes a sequence of greedy choices
- At every step, the greedy choice is the one that seems best
- Sometimes yields an optimal solution (sometimes, sub-optimal)

Ingredients exhibited by most problems suitable for a greedy strategy:

1. Greedy choice property
2. Optimal substructure: key property for Dynamic Programming and Greedy algorithms

Greedy choice property

A globally optimal solution can be derived by making a locally optimal choice

Dynamic programming: We make a choice at each step, but choice may depend on solutions to subproblems

Greedy algorithm: make whatever choice seems best at that moment, then solve subproblems arising after choice has been made

Choice may depend on previous choices made, but not on future choices or solutions to subproblems

Top-down approach: iteratively reducing each problem instance to a small one

Difficulty: prove greedy choice yields optimal solution, not a straightforward task

Dynamic programming vs. greedy algorithm

when is a greedy algorithm sufficient?
when is a dynamic programming required?

Illustration with two variants of famous knapsack problem

- 0-1 knapsack: dynamic programming required (Exercise 17.2-2, ask instructor for solution or check Dr. Cusack notes)
- Fractional knapsack: greedy algorithm suffices

Knapsack problem

A thief robbing a store:

— finds n items

— i^{th} item is worth v_i dollars and weighs w_i pounds, $v_i, w_i \in \mathbb{N}$

— he has a knapsack that can carry W pounds, $W \in \mathbb{N}$

Question: what items should he take (maximize gain)?

0-1 Knapsack

- thief can choose either to take or not to take an item
- thief cannot choose to take fraction of an item or take item more than once

Fractional knapsack problem

Same set up, but thief can take fraction of items

Optimal-substructure property

Satisfied by both knapsack problems **0-1 Knapsack**

- Consider optimal (most valuable) load of weight at most W
- Remove j from optimal load
- remaining load must be the most valuable load weighing $W - w_j$ that can be taken from $n - 1$ items (excluding j)

Fractional Knapsack

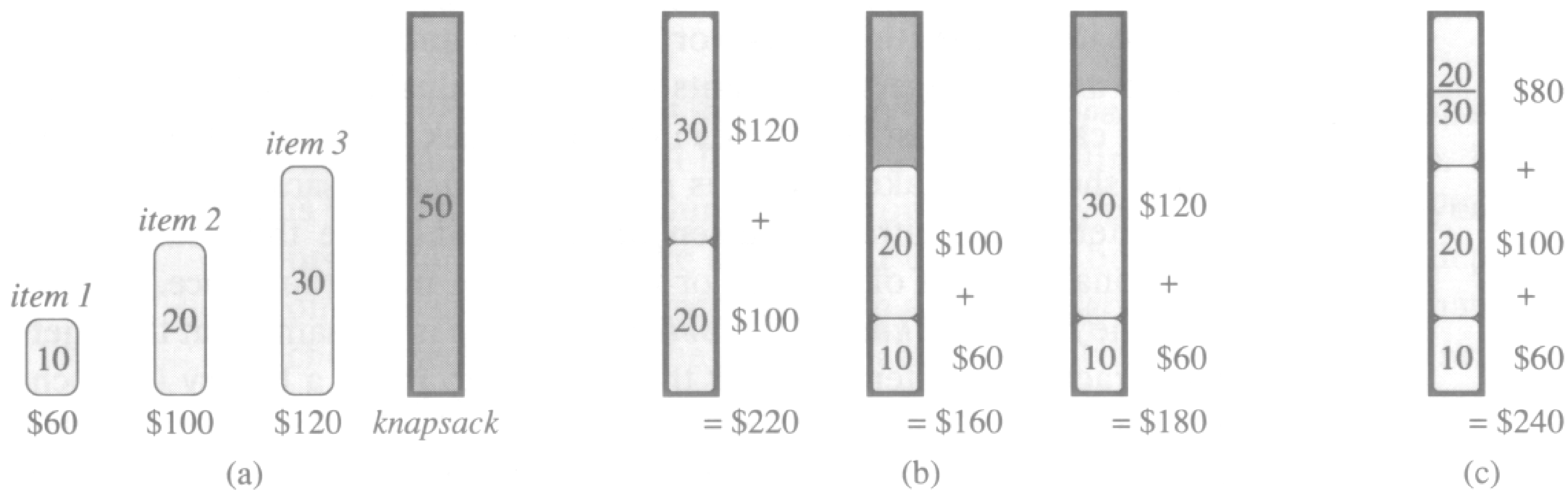
- Consider optimal (most valuable) load of weight at most W
- Remove weight w from one item j from optimal load,
- Remaining load must be the most valuable load weighing $W - w$ that can be taken from $n - 1$ original items <u>plus</u> $w_j - w$ pounds of item j

Greedy algorithm for Fractional knapsack problem

- Compute value per pound for each item: v_i/w_i
- Thief should take as much as possible of item with greatest value per pound
- When supply is exhausted and thief can still carry more, he should take as much as possible of the next greatest value per pound, etc.
- Sorting items by value/pound, complexity: $O(n \lg n)$

Greedy algorithm does <u>not</u> work for 0-1 knapsack problem

Item 1: \$6/pound, Item 2: \$5/pound, Item 3: \$4/pound,

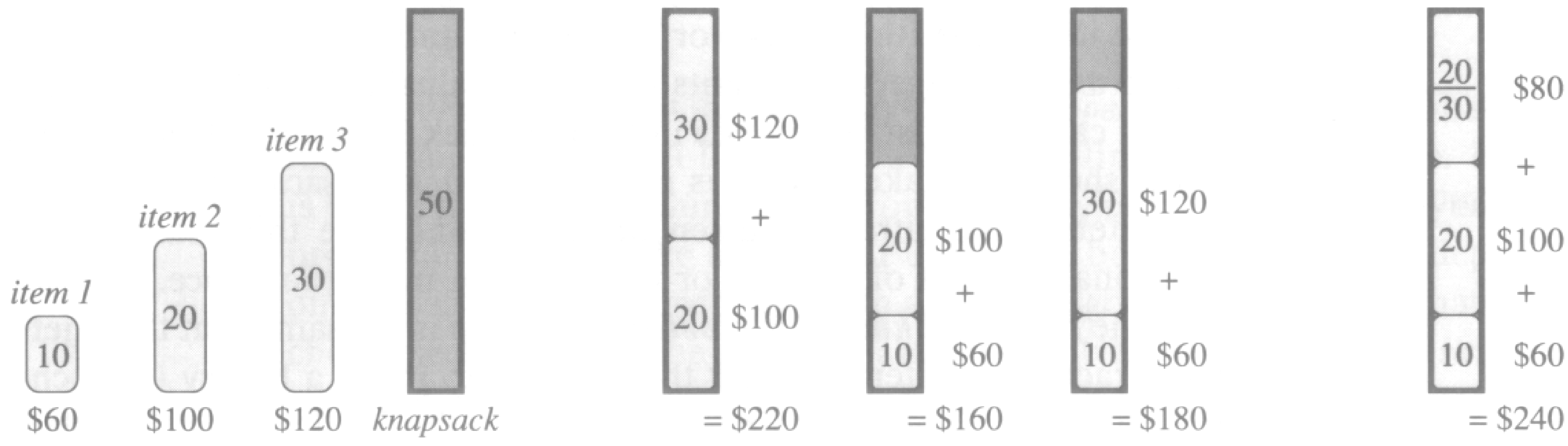


Greedy: takes item 1 first, but sack is not filled to capacity

Optimal: takes item 2 and item 3, leaves item 1 :—(

Greedy algorithm works for fractional knapsack problem

Item 1: \$6/pound, Item 2: \$5/pound, Item 3: \$4/pound,



(a) (b) (c)

Greedy: takes 10 pounds of item 1, 20 pounds of item 2 and 20 pounds of item 3

Optimal

Why Greedy algorithm does <u>not</u> work for 0-1 knapsack problem

- When considering an item for inclusion in knapsack
  Compare solution of subproblem in which item is included to solution to subproblem in which item is excluded, before making a choice
- gives rise to many overlapping subproblems
- a hallmark of dynamic programming

Huffman codes

- Used to compress data (savings 20% to 90%)
- Widely used, very effective
- Effectiveness depends on characteristics of file
- Uses a table of frequencies of occurrence of each character
  → too restrictive in a real-world setting (e.g., LempelZivWelsh LZW)
- Builds up an optimal way to represent each character as a binary string

Data encoding

Given:

- a data file with 100’000 characters
- list of characters that appear in file: $\{a, b, c, d, e, f\}$
- Frequency of appearance of characters:
  $(a, 45000), (b, 13000), (c, 12000), (d, 16000), (e, 9000), (f, 5000)$
- Characters represented by a binary character code (code)
  Each character is represented by a binary string

Goal: find a binary code for each character

Fixed length encoding: principle

Given the length:

- All characters are binary strings of length n (n-bit codeword)
- Number of possible characters is 2^n
- A data file 100 characters requires $100n$ bits of storage after encoding

Given the number of characters to encode:

- There are k characters to encode (k-bit codeword)
- Number of necessary bits: $\lceil \lg k \rceil$

Example:

- 6 characters $\Rightarrow n = 3$ (3-bit codeword)
- $a = 000, b = 001, c = 010, d = 011, e = 100, f = 101$
- A file of 100,000 characters can be encoded in 300,000 bits

From fixed to variable length encoding

Characters have fixed length:

Frequent characters and rare characters use same codeword length

Idea:

If we encode **frequent** characters with **shorter** codewords
we will need need **longer** codewords for **rarely occurring** characters
we may save on size of encoded file

Example:

Frequency of appearance of characters:

$(a, 45000), (b, 13000), (c, 12000), (d, 16000), (e, 9000), (f, 5000)$

Encoding: $(a, 0), (b, 101), (c, 100), (d, 111), (e, 1101), (f, 1100)$

Requires: $(45 \cdot 1) + (13 \cdot 3) + (12 \cdot 3) + (16 \cdot 3) + (9 \cdot 4) + (5 \cdot 4) \cdots 10^3$
$=$ 224,000 bits

Fixed length encoding: 300,000 bits

Variable length encoding: 224,000 bits

Savings: 25% approx.

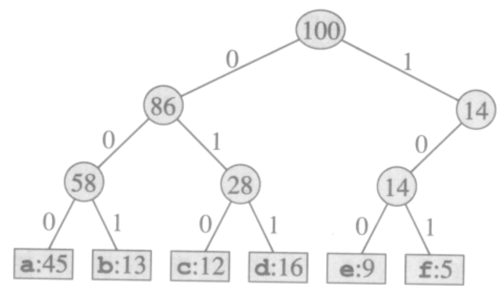
Note: this is an **optimal** character code for this file

Prefix code (read: prefix-free code)

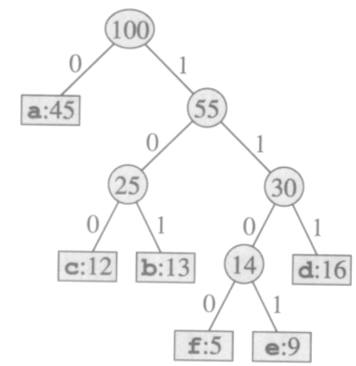
- No codeword is a prefix of another codeword
- Prefix code are desirable: encoding are decoding are simple
- Encoding is done by simple **concatenation** of codewords representing characters
  Example: `abc` → `01011100`
- Decoding: codewords are unambiguous as no codeword is a prefix of any other
  Identify initial codeword, translate it back, remove it from encoded file, and repeat..
  Example: `001011101` → `aabe`
- **Side note:** it can be shown that the optimal data compression achievable by a character code can always be achieved with a prefix code.
  ⇒ restriction to prefix code causes no loss of generality

Binary tree representation of a prefix code

Useful for quick decoding: codeword can be easily picked off



(a)



(b)

Binary tree:

— Leaves represent the characters to encode

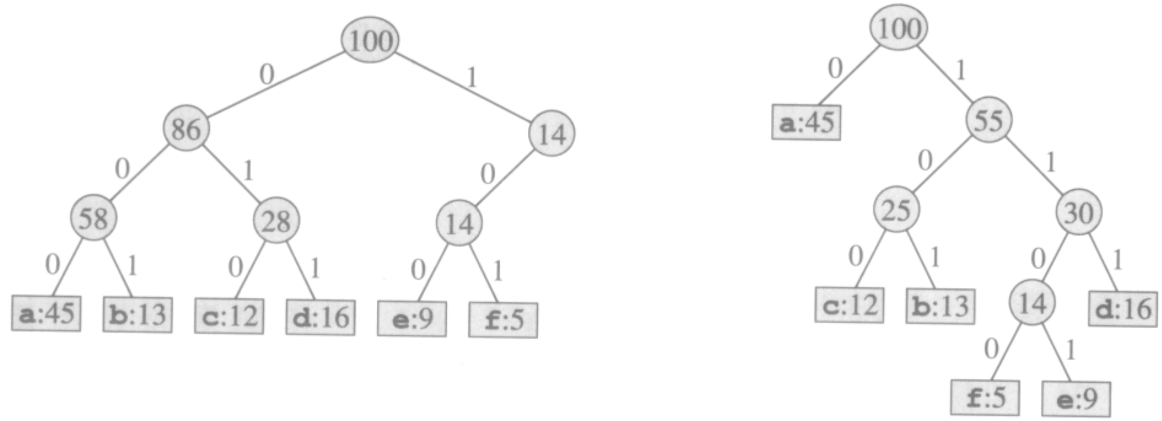
— Path from root to a leaf interpreted as binary codeword:

left child means 0 , right child means 1

— Binary trees, not binary search trees: internal nodes do not contain character keys, leaf keys do not necessarily by binary search tree property

Optimal prefix-code representation: full binary tree

An **optimal** code is always represented by a **full** binary tree (every nonleaf node has two children)

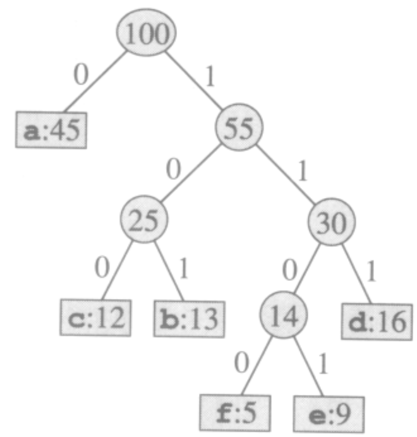


(a) (b)

Fixed-length code: not a full binary tree, thus not optimal

∃ codewords beginning with 10 but ¬∃ beginning with 11

Optimal prefix-code representation: full binary tree



(b)

If C is the alphabet, then the tree for an optimal prefix code has:

- $\|C\|$ leaves
- $\|C\| - 1$ internal nodes

Optimal prefix-code representation: full binary tree

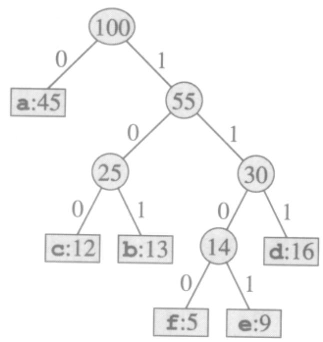
f C is the alphabet, then the tree for an optimal prefix code has:

- $\|C\|$ leaves
- $\|C\| - 1$ internal nodes

Proof: Suppose it has x internal nodes aside from the root:
number of nodes in tree $= x + \|C\| + 1$
number of edges in tree = number of nodes -1 $= x + c$
Degree of nodes: leaf node $= 1$, root node $= 2$, otherwise $= 3$
Sum (degree of nodes) $= 2$ number of edges in tree (handshaking lemma)
$c + 3x + 2 = 2(x + c) \Rightarrow x = c - 2$
number of internal nodes $= x$ +1 (root) $= c - 1$

Optimal prefix-code representation:

number of bits required



(b)

Given Tree T, for each character $c \in C$:

- $f(c)$ denotes frequency in file
- $d_T(c)$ denotes depth of c's leaf in the tree
  $d_T(c)$ is also the length of codeword for c

Cost of T: The number of bits required to encode a file:

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

Can be obtained by summing values of internal nodes, including root

(Expression reminds us of Shannon entropy $-\sum p_i \lg(p_i)$) (double-check)

Huffman code

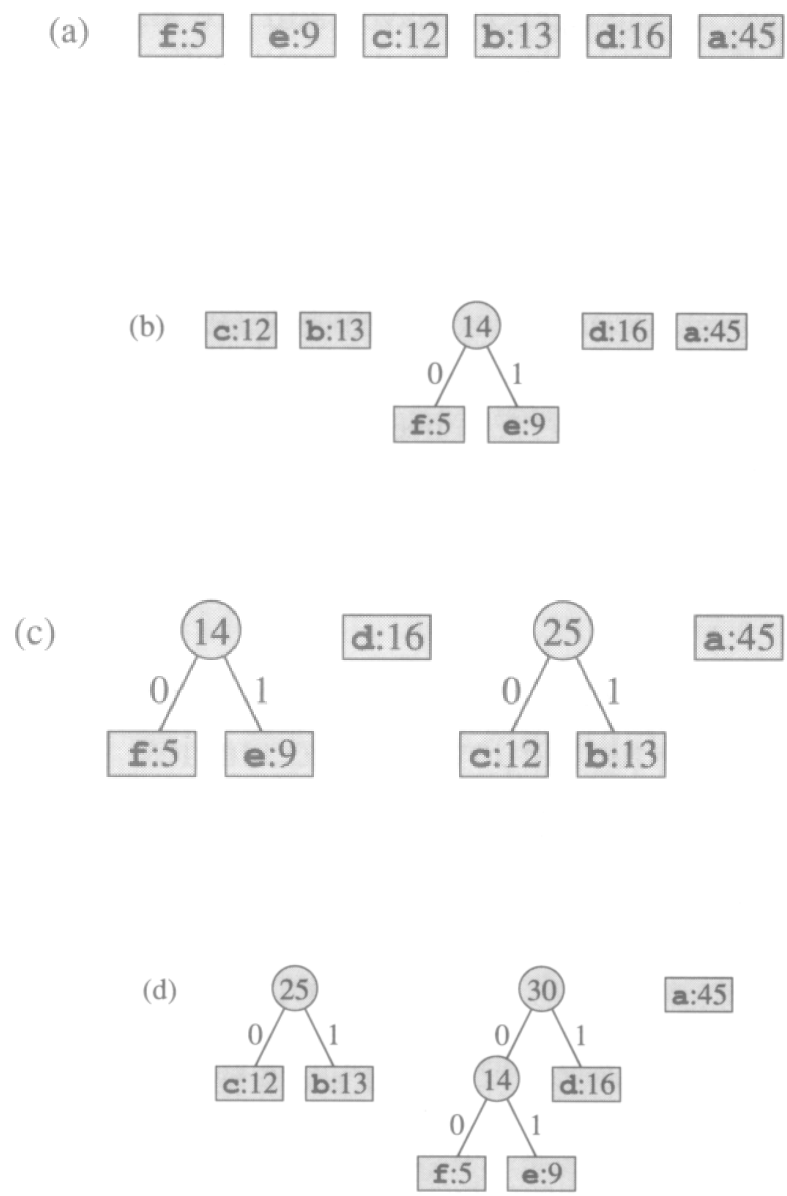
invented as course homework

Greedy algorithm for constructing optimal code, bottom-up manner.

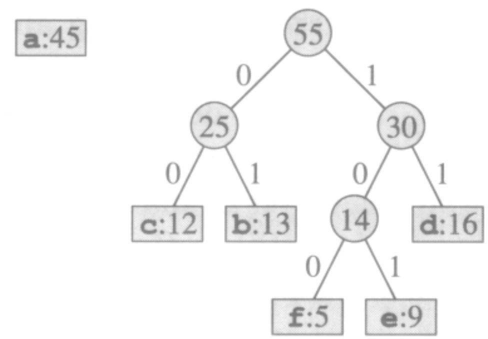
- Starts with $\|C\|$ leaves, performs $\|C\| - 1$ merging operations
- C set of n characters, for $c \in C, f[c]$
- Q priority queue, keyed on f, lists characters, objects, with increasing frequency
  how to implement Q? What needs to be changed?
- Consider the 2 least-frequent objects and merge them
  Replace them into one object with frequency $=$ sum of individual frequencies
  Link new and original nodes, label edges with 0 and 1
  Position new object in priority queue
- repeat until the list has only one node

Huffman code: Example

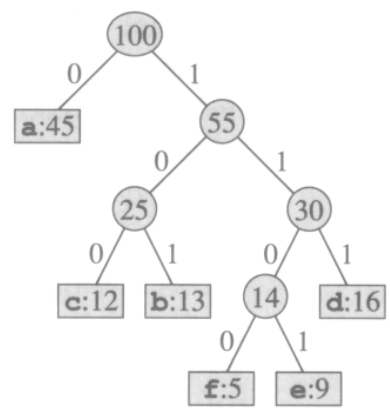
$n = 6$, 5 merge steps required



(e)



(f)



Final tree represents optimal prefix code

Codeword for a character is sequence of edge labels

Huffman code: Pseudocode

```
HUFFMAN(C)
n ← |C|
Q ← C
for i ← 1 to n − 1
      do z ← ALLOCATE-NODE()
         x ← left[z] ← EXTRACT-MIN(Q)
         y ← right[z] ← EXTRACT-MIN(Q)
         f[z] ← f[x] + f[y]
         INSERT(Q, z)
return EXTRACT-MIN(Q)
```

Huffman code: interpretation of Pseudocode

Line 2: initializes Q with characters in C

Lines 3–8: repeatedly extracts the 2 nodes of lowest frequency x and y, replaces them with a new node z, parent of x and y (left/right not important: yields different codes of same cost)
There will be $n-1$ merges

Line 9: returns root

Huffman code: Analysis

- Assuming Q implemented as a binary heap
- Line 2 requires `Build-Heap` $\rightarrow O(n)$
- Lines 3–8 called $n-1$ times
- Each `Extract-Min` requires $O(\lg n)$
- Lines 3—8 cost $O(n \lg n)$
- Total running time for n characters $O(n \lg n)$

Huffman code: Correctness

We need to prove that problem of determining optimal prefix code has:

1. Greedy choice property: Lemma 17.2
   If x and y have 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 in only the last bit

2. Optimal-substructure property: Lemma 17.3
   Let z be the parent of any two characters x and y siblings leaves in T, with $f[z] = f[x] + f[y]$, $T' = T - \{x, y\}$ is an optimal prefix code for $C' = C - \{x, y\} \cup \{z\}$

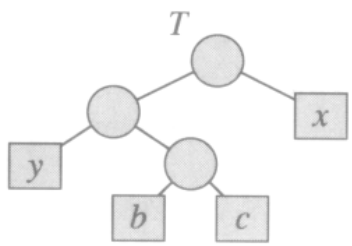
Huffman code: Proof of Greedy choice property (I)

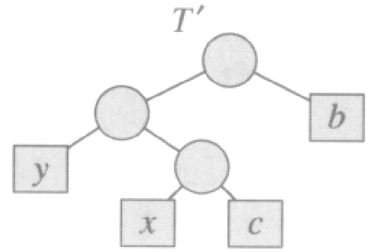
Informally: *If x and y have the lowest frequencies, then there exists an optimal prefix code for C in which the codewords for x and y have the same length and differ in only the last bit*

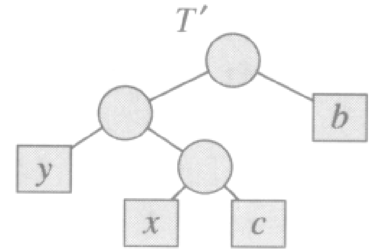
Idea:

- Take T, tree representing an arbitrary optimal prefix code
- Modify T to generate T', a tree representing another optimal prefix code such that x and y appear as sibling leaves in the new tree
- If we can do that, then their codewords will have same length and differ only in the last bit

Huffman code: Proof of Greedy choice property (II)







- Let b, c sibling leaves of maximum depth in T, assume $f[b] \leq f[c]$ and $f[x] \leq f[y]$
- x, y lowest frequency leaf frequencies, then $f[x] \leq f[y] \leq f[b] \leq f[c]$ and, a fortiori, $f[x] \leq f[b]$ and $f[y] \leq f[c]$
- Exchange $b \leftrightarrow x$ in T, yielding T'
- Exchange $c \leftrightarrow y$ in T', yielding T''
- Compute $B(T) - B(T')$, and show $B(T) - B(T') \geq 0$
  Similarly, $B(T') - B(T'') \geq 0$
  Therefore $B(T) \geq B(T') \geq B(T'')$
- Since T is optimal then $B(T) = B(T') = B(T'')$, and T'' is optimal

Greedy choice property: Lesson

Lemma 17.2 establishes that to build up an optimal tree by mergers, we can, without loss of generality begin with the greedy choice of *merging together those two characters of lowest frequency*

Why is this a greedy choice:

- **Cost** of a single merger is sum of frequencies of 2 objects being merged
- Huffman code chooses the merger that incurs the least cost
- Incidentally, the sum of all mergers (cost of root) is the cost of the code

Huffman code: Proof of optimal substructure property

Informally: *Let* z *be the parent of any two characters* x *and* y *siblings leaves in* T, *with* $f[z] = f[x] + f[y]$, $T' = T - \{x, y\}$ *is an optimal prefix code for* $C' = C - \{x, y\} \cup \{z\}$

Idea:

- We show that the cost $B(T)$ can be expressed in terms of $B(T')$
- $\forall\, c \in C - \{x, y\}$, $d_T(c) = d_{T'}(c)$
  $\Rightarrow f[c]d_T(c) = f[c]d_{T'}(c)$
  Since We have $d_T(x) = d_T(y) = d_T(z) + 1 = d_{T'}(z) + 1$
  Therefore, $f[x]d_T(x) + f(y)d_T(y) = f[z]d_{T'}(z) + (f[x] + f[y])$
  $\Rightarrow B(T) = B(T') + f[x] + f[y]$
- Suppose T' represents a <u>nonoptimal</u> prefix code for $C' \Rightarrow \exists\, T''$ with $B(T'') < B(T')$
- $z \in C' \Rightarrow z$ is a leaf in T''

- Adding x and y as children of z in T'' yields a prefix code for C with cost $B(T'') + f[x] + f[y]$
  Since $B(T'') < B(T') \Rightarrow B(T'') + f[x] + f[y] < B(T') + f[x] + f[y] = B(T)$
  So we have a prefix code for C cheaper than $B(T)$
- Contradiction with optimality of T and T' must be optimal for C'