Greedy Algorithms

Recall that Dynamic Programming consists of two parts:

_____

_____

Greedy Programming also consists of two parts:

1. _____
2. _____ A globally-optimal solution can be obtained by making a locally-optimal (greedy) choice. In other words, the globally optimal solution does not depend on the solution to its subproblems.

A **Greedy Algorithm** starts with a locally-optimal choice, and continues making locally-optimal choices until a solution is found.

Proving Greedy Algorithms Optimal

Need to prove 1) optimal substructure and 2) greedy choice property.

The proof of 2 typically involves:

a. Consider globally-optimal solution.

b. Show greedy choice at first step reduces problem to the same but smaller problem.
Greedy choice must be

- Part of an optimal solution, and
- Can be made first

c. Use induction to show greedy choice is best at each step (i.e., optimal substructure)

Example: Fractional Knapsack Problem

A thief considers taking W pounds of loot. The loot is in the form of n items, each with weight w_i and value v_i. Any amount of an item can be put in the knapsack as long as the weight limit W is not exceeded.

Question:

How much of which items should the thief take to maximize the value of the loot?

Greedy Algorithm: Take as much of the item with the _____ value per pound (v_i/w_i) as you can. If you run out of that item, take from the _____ (v_i/w_i) item. Continue until knapsack is full.

The Fractional Knapsack Problem exhibits Optimal Substructure

Proof Sketch: Consider the most valuable load weighing W. If we remove weight w of item i from the loot, then the remaining load must con-

tain the optimal selection of the other n-1 items and the $w_i - w$ pounds of item i. If not, then the original load was not the most valuable.

Fractional Knapsack Problem exhibits Greedy Choice Property

Proof:

First show that as much as possible of the highest value/pound item must be included in the optimal solution.

Let w_h, v_h be the weight available and value of the item with the highest value/pound ratio (item h).

Let L(i) be the weight of item i contained in the thief's loot L.

$$\text{Total Value V} = \Sigma_{i=1}^{n} L(i)\frac{v_i}{w_i}$$

If some of item h is left, and L(j) $\neq 0$ for some j $\neq$ h, then replacing j with h will yield a higher value.

$$\text{L(j)}\ \frac{v_j}{w_j} \leq L(j)\frac{v_h}{w_h}$$

$$\frac{v_j}{w_j} \leq \frac{v_h}{w_h} \text{ true by definition of h}$$

Fractional Knapsack Problem

Now we need to show that an initial greedy choice of item h leads to an optimal solution. We can make the greedy choice first.

Case I: If W $< w_h$, then _____ and for all j $\neq$ h _____ is the only take.

We can fill the knapsack with the highest value-per-pound item.

Case II: There is room left after greedy choice. L(i), L(j) > 0, i $\neq$ j. Assume item i was the first choice and j = h a subsequent choice.

Thus, V = L(i) $\frac{v_i}{w_i}$ + L(h) $\frac{v_h}{w_h}$.

If we choose the same amount L(h) of item h first (we cannot choose more or we would have by the above result), then the value will be

$$V' = L(h)\frac{v_h}{w_h} + L(i)\frac{v_i}{w_i}$$
$$V' = V$$

Thus we can make the greedy choice first.

Activity-Selection (scheduling) problem

- S = {1, 2, .., n} proposed activities
- All want the same resource (classroom), used by one at a time
- Each activity i has start time s_i and finish time f_i, $s_i \leq f_i$, $[s_i, f_i)$
- Two activities i and j are compatible if they do not overlap ($s_i \geq f_j$ or $s_j \geq f_i$)

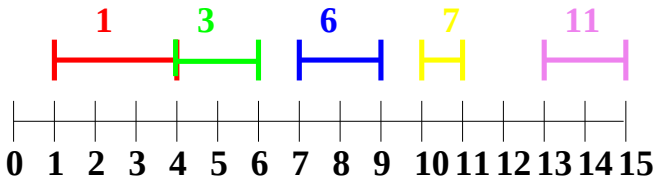
Problem: Select maximum-size set of compatible activities. Assume the input is sorted by $f_1 \leq f_2 \leq .. \leq f_n$ (O(nlgn)).

Greedy algorithm:

```
Greedy-AS(s, f)
1    n = length(s)
2    A = {1}              ; Initialize A
3    j = 1
4    for i = 2 to n
5        if s_i ≥ f_j     ; Compatible
6        then A = A ∪ {i}
7             j = i
8    return A
```

i	si	fi
1	1	4
2	3	5
3	4	6
4	5	7
5	3	8
6	7	9
7	10	11
8	8	12
9	8	13
10	2	14
11	13	15

A = {1}, fj = 4
A = {1,3}, fj = 6
A = {1,3,6}, fj = 9
A = {1,3,6,7}, fj = 11
A = {1,3,6,7,11}, fj = 15



Activity Selection Problem

- Note that f_j always has the maximum finish time of any activity in A
- Greedy-AS takes $\Theta(n)$ time
- This algorithm is greedy because it always picks the activity with the earliest compatible finish time (leave as much time as possible)
- Optimal? ____
- Proof
  - Note that if ordered by f_i, activity 1 has earliest finish
  - Show that there exists an optimal solution that begins with a greedy choice (activity 1).
  - Let A $\subseteq$ S be an optimal solution. If the first activity in A is $k \neq 1$ (not greedy), then there exists another optimal solution B that begins with 1.
  - B = A - {k} $\cup$ {1}

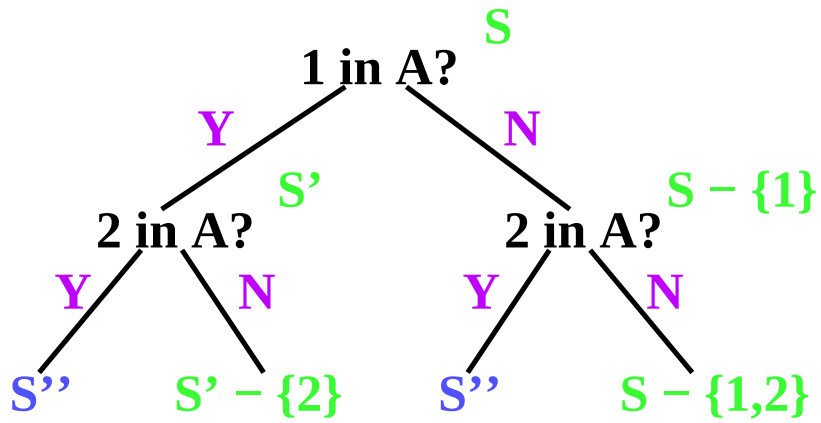
- Because $f_1 \leq f_k$, activity 1 is still compatible with A
- |B| = |A|, so B is also optimal. Thus there exists an optimal solution that **begins** with a greedy choice.
- Now prove optimal substructure. Once we make the first greedy choice, now the problem is $S' = \{i \in S : s_i \geq f_1\}$, with **optimal solution** A' = A - {1}.
- This solution must be optimal. If B' solves S' with more activities than A', adding activity 1 to B' makes it bigger than A, which is a contradiction.
- Therefore, greedy choice yields an optimal solution.

Dynamic Programming for Activity Selection

Exercise 17.1-1

1. Optimal substructure (already proven)
2. Recursive, overlapping problems (need to show this)
3. Bottom-up algorithm
4. Compute solution

Recursive Algorithm — find all possible compatible subsets, notice repeated subproblems.



S' = {i in S: si >= f1}
S'' = {i in S: si >= f2}

The number of distinct subproblems is exponential (the power set), so Dynamic Programming is not a good idea.

0-1 Knapsack Problem

- The thief wants to steal n items
- The ith item weighs w_i and has value v_i
- Take the most valuable load, limit of W pounds
- This is called the 0-1 version of the problem because there are no fractions.
  The thief must take the whole item or leave it behind.
- Both the 0-1 Knapsack Problem and the Fractional Knapsack Problem have optimal substructure. If we remove item j from the optimal load, the remaining problem must be optimal for the remaining W - w_j pounds for the overall solution to be optimal.

- The greedy solution for the Fractional Knapsack Problem _____ work here.

Example:

Item	Value	Weight	Val/Weight
1	60	10	6
2	100	20	5
3	120	30	4

W = 50

The greedy solution would select 1, leaving 40 pounds.

At this point if we pick items 1 and 2 the total value is 160.

If we pick items 1 and 3 the total value is 180.

The optimal solution is to pick items _____. The total value of this solution is ____.

Dynamic Programming for 0-1 Knapsack Problem

Exercise 17.2-2

For this algorithm, let c[i,w] = value of solution for items 1..i and maximum weight w.

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

```
DP-01K(v, w, n, W)
1     for w = 0 to W
2         c[0,w] = 0
3     for i = 1 to n
4         c[i,0] = 0
```

	a	b	c	d	e	f	
Frequency	45k	13k	12k	16k	9k	5k	
Fixed-Length Code	000	001	010	011	100	101	300,000 bits
Variable-Length Code	O	101	100	111	1101	1100	224,000 bits (optimal)

```
5       for w = 1 to W
6           if w[i] ≤ w
7           then if v[i] + c[i-1,w-w[i]] > c[i-1,w]
8                then c[i,w] = v[i] + c[i-1,w-w[i]]
9                else c[i,w] = c[i-1,w]
10            else c[i,w] = c[i-1,w]
```

The run time performance of this algorithm is $\Theta(nW)$.

Huffman Codes

Data compression by assigning binary codes to characters.

For example, consider a 100,000 character file with only six different characters: a, b, c, d, e, f.

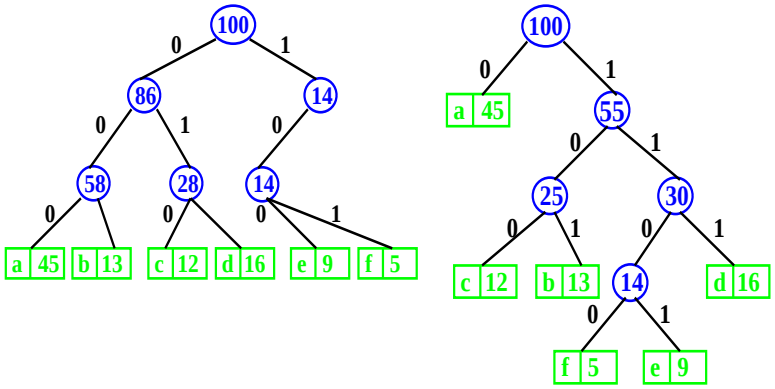
Huffman's algorithm determines optimal variable-length code (Huffman Codes).

Prefix Codes

In a **prefix code**, no code is a prefix of another code.

Why?

1. Any optimal character code has an equally optimal prefix code.
2. Easy encoding and decoding.
   "abc" $\longrightarrow$ 0.101.100
   Once a string of bits matches a character code, output that character with no ambiguity (no need to look ahead).
   Use a binary tree for decoding.



Prefix Codes

Note: An optimal code is always represented by a full binary tree in which every non-leaf node has exactly two children.

Given tree T, compute the number of bits to represent file.
Let C = set of unique characters in file.
Let f(c) = frequency of character c in file.
Let $d_T(c)$ = depth of c's leaf node in T

Then, the number of bits required to encode a file B(T) = $\Sigma_{c \in C} f(c) d_T(c)$.
B(T) = cost of tree T

Greedy Algorithm for Huffman Code

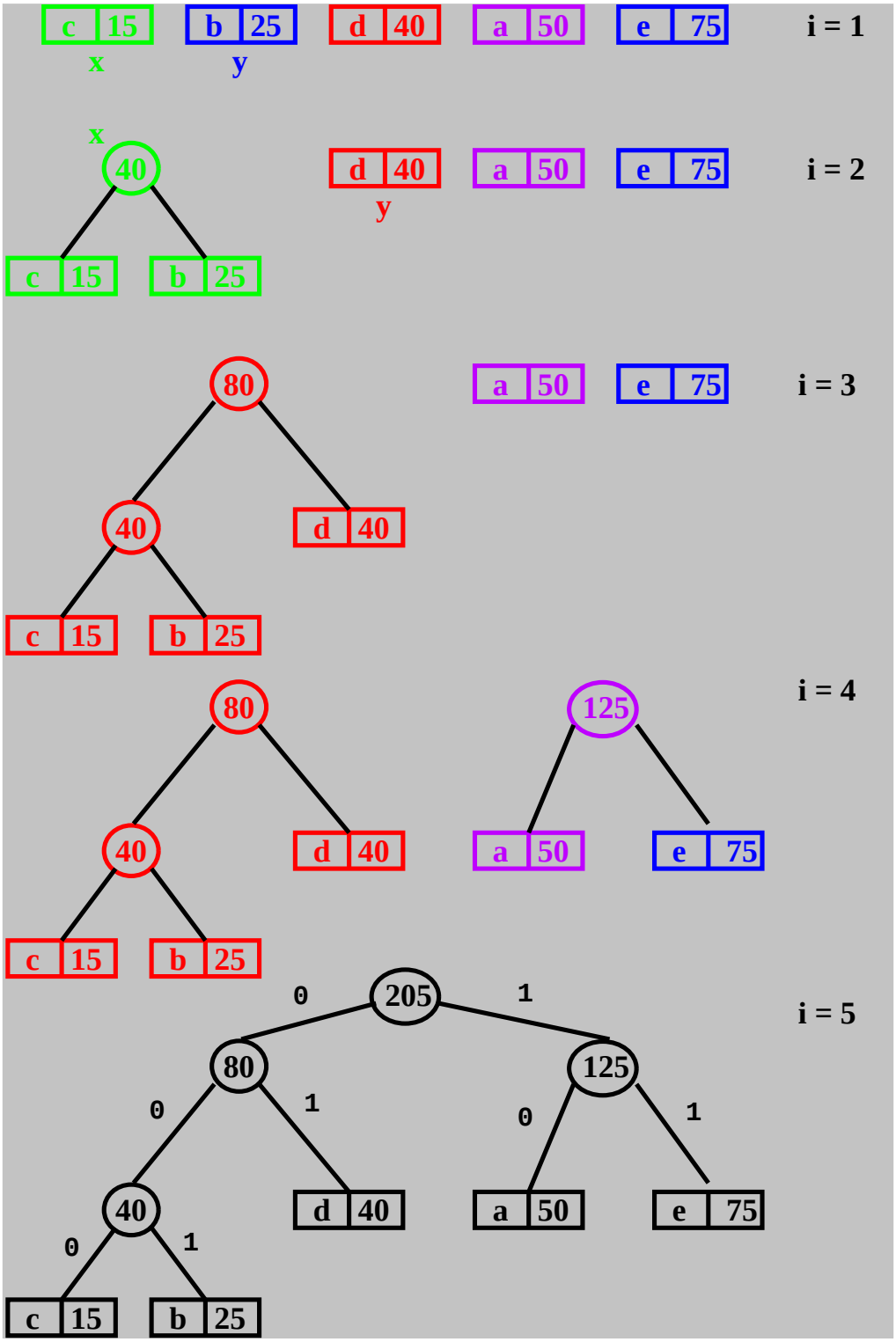
Idea: Merge the two _____ nodes (leaf or internal) into a new node until every leaf has been considered.

Use a priority queue Q to keep nodes ordered by frequency.

```
Huffman(c)                          ; Analysis
   n = |c|                          ; Q is a binary heap
   Q = c                            ; O(n) BuildHeap
   for i = 1 to n-1                 ; O(n)
      z = Allocate-Node()
      x = Extract-Min(Q)            ; O(lgn) O(n) times
      y = Extract-Min(Q)            ; O(lgn) O(n) times
      left(z) = x
      right(z) = y
      f(z) = f(x) + f(y)
      Insert(Q,z)                   ; O(lgn) O(n) times
   return Extract-Min(Q)            ; ----------------
                                    ; O(nlgn)
```

Example

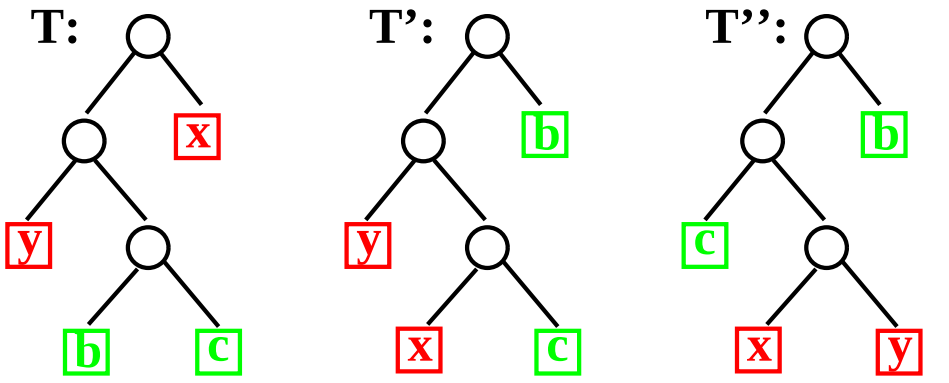
C = a, b, c, d, e
f(C) = 50, 25, 15, 40, 75
n = 5


c 15
b 25
d 40
a 50
e 75
i = 1
x
y
x
40
c 15
b 25
d 40
a 50
e 75
i = 2
y
80
40
d 40
c 15
b 25
a 50
e 75
i = 3
80
40
d 40
c 15
b 25
125
a 50
e 75
i = 4
0
205
1
80
125
0
1
0
1
40
d 40
a 50
e 75
0
1
c 15
b 25
i = 5

Lemma 17.2

Huffman exhibits the greedy choice property.

Proof:



B(T') <= B(T) **B(T'') <= B(T)**

- Assume x and y have lowest frequencies.
- If x and y have lowest frequencies, then there exists an optimal code in which x and y are at the maximum depth (greedy choice).
- Greedy choice would put them where _ and ___ are in T.
- Assume f(x) $\leq$ f(y) and f(b) $\leq$ f(c). We know f(x) $\leq$ f(b) and f(y) $\leq$ f(c).
- B(T) - B(T') $= \Sigma_{c \in C} f(c) d_T(c) - \Sigma_{c \in C} f(c) d_{T'}(c)$

$$= f[x]d_T(x) + f[b]d_T(b) - f[x]d_{T'}(x) - f[b]d_{T'}(b)$$

$$= f[x]d_T(x) + f[b]d_T(b) - f[x]d_T(b) - f[b]d_T(x)$$

$$= (f[b] - f[x])(d_T(b) - d_T(x))$$

$$\geq 0$$

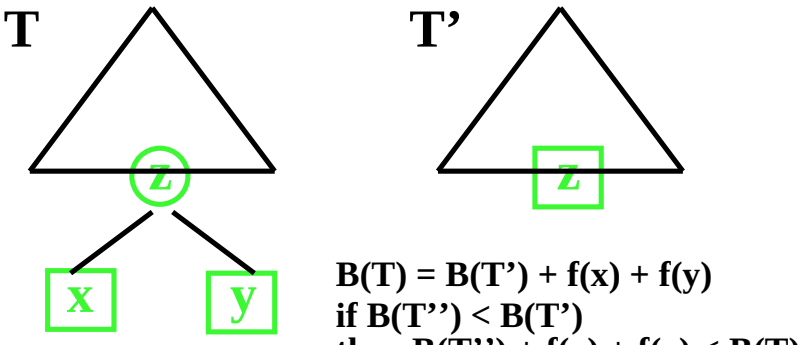
- Thus, moving x to the bottom (similarly, y to the bottom) yields a better (optimal) solution.

Lemma 17.3

Huffman exhibits optimal substructure.

Proof: Consider the optimal tree T for characters C.

- Let x and y be lowest frequency characters in C.
- Consider the optimal tree T' for C' = C - {x,y} ∪ {z}, where f(z) = f(x) + f(y).
- If there is a better tree for C', call it T", then we could use T" to build a better original tree by adding in x and y under z.
- The original tree T is optimal, so this is a contradiction.
- Thus T' is the optimal tree for C'.
- Huffman exhibits optimal substructure.



Theorem 17.4

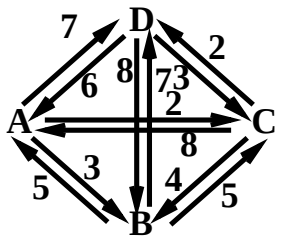
Huffman produces optimal prefix codes.

Example

Not all divide-and-conquer problems exhibit optimal substructure.

Consider the problem of visiting all vertices of a graph in a simple cycle, starting from a chosen vertex. The problem is to find the cycle that yields the smallest total edge cost. We will break our problem down by finding the last vertex to visit in our tour. The subproblem is then to find the optimal arrangement of vertices to visit before the chosen last vertex.

Consider this example. The optimal tour is $A \to B \to C \to D \to A$, but when the last vertex D is removed, the optimal tour for the subproblem is $A \to C \to B \to A$, which is not part of the globally optimal solution.



Applications