Topic 12: Greedy Algorithms

(CLRS 16.0–16.3)

CPS 230, Fall 2001

1 Greedy Algorithms

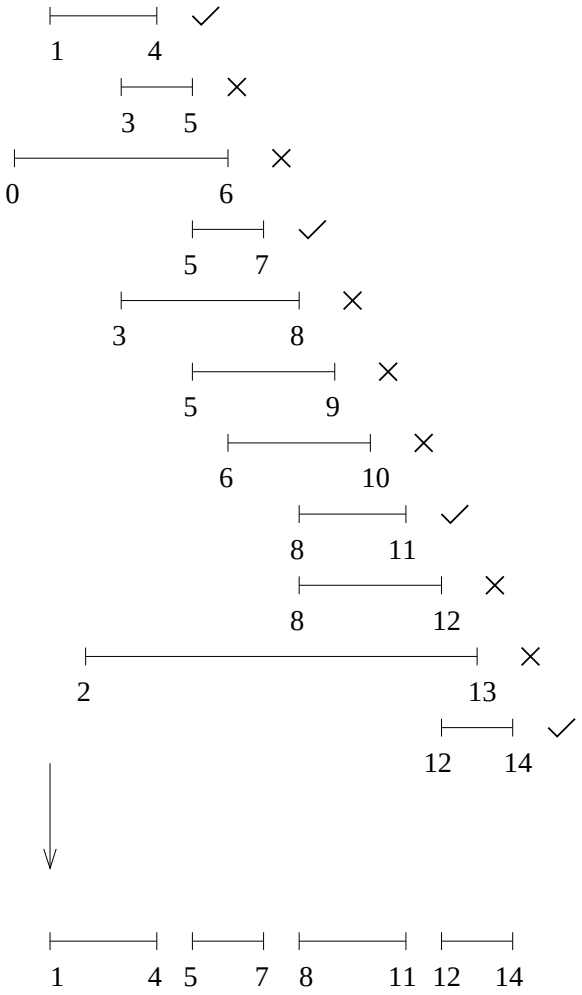
- In divide-and-conquer (and thus dynamic programming) we used the fact that the solution to a problems depends on solutions to smaller subproblems.
- A simpler but often less powerful (and less well-defined) technique that uses the same feature is *greediness*
- Like in the case of dynamic programming, we will introduce greedy algorithms via an example.

1.1 Activity Selection

- Problem: Given a set $A = \{A_1, A_2, \cdots, A_n\}$ of n activities with start and finish times (s_i, f_i), $1 \leq i \leq n$, select maximum set S of "nonoverlapping" activities.
  - We can think of the problem as corresponding to scheduling the maximum number of classes (given their start and finish times) in one classroom.
- Solution:
  - Sort activity by finish time (let $A_1, A_2, \cdots, A_n$ denote sorted sequence)
  - Pick first activity A_1
  - Remove all activities with start time before finish time of A_1
  - Recursively solve problem on remaining activities.
- Program:

```
Sort A by finish time
S = {A_1}
j = 1
FOR i = 2 to n DO
    IF s_i ≥ f_j THEN
        S = S ∪ {A_i}
        j = i
    FI
OD
```

- Example:
  - 11 activities sorted by finish time: $(1,4), (3,5), (0,6), (5,7), (3,8), (5,9), (6,10), (8,11), (8,12), (2,13), (12,14)$



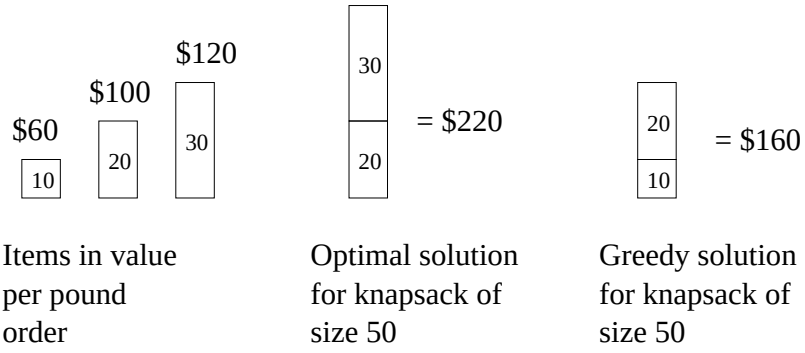
- Running time is obviously $O(n \log n)$.
- Is algorithm correct?
  - Output is set of nonoverlapping activities, but is it the largest possible?
- Proof of correctness:
  - Given activities $A = \{A_1, A_2, \cdots, A_n\}$ ordered by finish time, there is an optimum solution containing A_1:
    - Suppose $S \subseteq A$ is an optimum solution
    - If $A_1 \in S$, we are done
    - If $A_1 \notin S$:
      - Let first activity in S be A_k
      - Make new solution $S' = S \setminus \{A_k\} \cup \{A_1\}$ by removing A_k and using A_1 instead
      - S' is valid solution ($f_1 < f_k$) of maximum size ($|S'| = |S|$)
  - S is an optimum solution for A containing A_1
    $\Longrightarrow$ $S' = S \setminus \{A_1\}$ optimal solution for $A' = \{A_i \in A : s_j \geq f_1\}$.

    - Suppose we have solution S'' to A' such that $|S''| > |S'| = |S| - 1$
    - $S''' = S'' \cup \{A_1\}$ would be solution to A
    - Contradiction since we would have $|S'''| > |S|$
  - That is, after choosing A_1, the problem reduces to finding an optimum solution for activities not overlapping with A_1.
  - Correctness follows formally by induction on the size of S:
    - The algorithm trivially finds an optimum S for the base case when the optimum size is $|S| = 1$.
    - Suppose inductively that our algorithm finds an optimum S whenever $|S| = k$. Now suppose that the optimum S for A has size $|S| = k + 1$. We know from what we just proved above that there is an optimum S that includes A_1. The other activities in S cannot conflict with A_1 and therefore must be in the set $A' = \{A_i \in A : s_j \geq f_1\}$. Any optimum solution S' to A' has size $k + 1 - 1 = k$, and therefore by the induction hypothesis, our algorithm can find an optimum solution to A' of size k.
    - The solution S that our algorithm constructs for A consists of A_1 and the solution to A', yielding a solution of size $k + 1$. Therefore, S is an optimum solution.

- Comparison of greedy algorithm technique with dynamic programming (divide-and-conquer):
  - In greedy algorithm we choose what looks like best solution at any given moment and recurse (choice does not depend on solution to subproblems).
  - In dynamic programming, solution depends on solution to subproblems.
  - Both techniques use an optimum solution to subproblems. The optimum solution "contains optimum solutions for subproblems within it".

- It is often hard to figure out when being greedy works!

  Example:
  - 0–1 KNAPSACK PROBLEM: Given n items, where item i is worth $\$v_i$ and weighs w_i pounds, fill a knapsack of capacity w pounds with the maximum possible value.
  - FRACTIONAL KNAPSACK PROBLEM: As in the 0–1 KNAPSACK PROBLEM but we can divide items into fractional units.
  - Problems appear very similar, but only FRACTIONAL KNAPSACK PROBLEM can be solved greedily:
    - Compute value per pound v_i/w_i for each item
    - Sort items by value per pound.
    - Fill knapsack greedily (take objects in order)
      $\Longrightarrow$ $O(n \log n)$ time, which is easy to show is optimal time.
  - Example that 0–1 KNAPSACK PROBLEM cannot be solved greedily:



Note: In FRACTIONAL KNAPSACK PROBLEM we can take $\frac{2}{3}$ of the \$120 object (as well as all of the first two objects) and get a \$240 solution.

- 0–1 KNAPSACK PROBLEM can be solved in time $O(n \cdot w)$ using dynamic-programming (homework).

1.2 Huffman codes

- Another example of a greedy algorithm is the algorithm for computing *Huffman codes*—used to compress data.
- Consider the following example:
  - We are given a $100,000$-character data file where only the characters a, b, c, d, e, f appear, and they appear with the following frequencies (number of characters per 100 characters):

	a	b	c	d	e	f
Frequency	45	13	12	16	9	5

  - We store the data file using the following *fixed-length* binary code:

	a	b	c	d	e	f
code	000	001	010	011	100	101

    The resulting file has size $300,000$ bits.
  - We could also store the data using the following *variable-length* binary code:

	a	b	c	d	e	f
code	0	101	100	111	1101	1100

    The idea is to use short codes for frequent characters and long codes for infrequent characters. With this code the file has size $(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1000 = 224,000$
    $\Longrightarrow$ Savings of $\approx 25\%$
- In general we will always be interested in *prefix codes* where no codeword is a prefix of another codeword.
  - We can prove that best compression achievable by any character code can always be achieved by a prefix code.
  - Allows us to efficiently encode data: Just concatenate the codes for each character in the file.

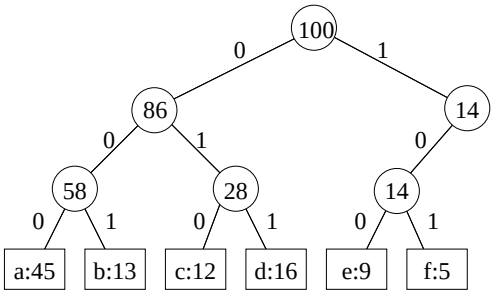
- Prefix property allows us to uniquely decode file: Just identify codeword at beginning of file, remove and translate, and iterate.

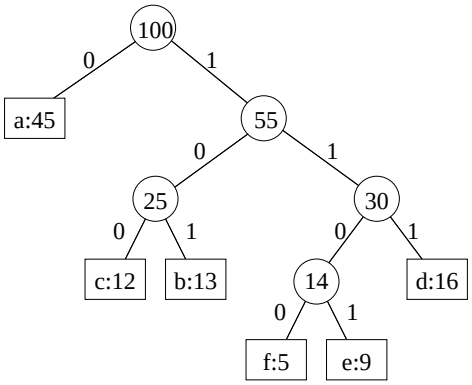
- We need convenient representation of prefix code to store with compressed file.

$\Longrightarrow$ We represent code with binary tree:

- Characters in leaves
- Branches marked with 0 or 1
- Code for character in leaf ℓ consists of the labels on the path from the root to ℓ

Examples: a fixed-length code tree and an optimal variable-length code tree:





- Note: An optimal code is always represented by a *full* tree (or technically a binary tree, in that every internal node has two children, each of which can be either an internal node or a leaf).
- The prefix property is equivalent to the each character being represented by a leaf: The node for a character cannot be an ancestor (prefix) for the node for another character.
- If we have a set of characters C, the tree has $|C|$ leaves and $|C|-1$ internal nodes.

- Given a code tree T, we can compute the *cost* of T (i.e., the number of bits (divided by 100) used to encode a file) as

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

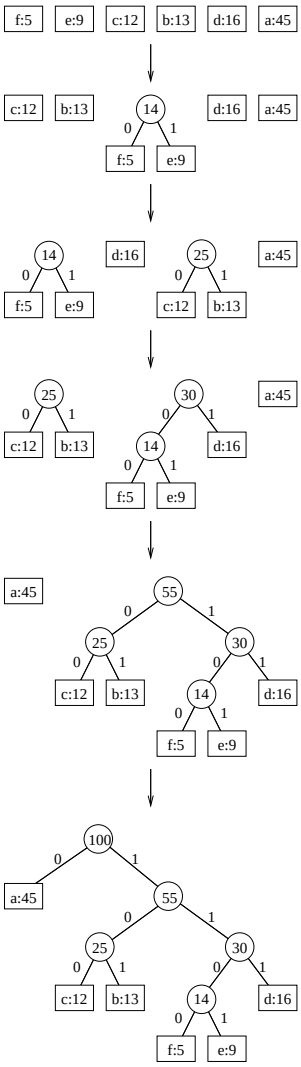
where $f(c)$ is frequency of character c (per 100 characters), and $d_T(c)$ is the depth in T of the leaf containing c.

- We want to compute the *Huffman code*: the code (tree T) that minimizes $B(T)$, which gives the optimum code and therefore optimum compression:
  - Note: Better compression schemes do exist that use the frequency of sequences of characters (a.k.a. words).
  - Examples: Ziv-Lempel (as in UNIX compress and gzip), and PPM (Markov model) algorithms.
  - You can think of such word-based schemes as using higher-order statistics. That is, the code used for the character "e" depends upon its context consisting of the previous characters in the file. We could keep statistics based upon the context to determine the likelihood of an "e" occurring next.

- Greedy algorithm for finding the Huffman code. (Let Q represent a priority queue):

```
FOR each c ∈ C DO
    Construct leaf for c
    Insert(Q, c)            */ Key of c is f(c)*/
OD
REPEAT |C| − 1 times DO
    x=DeleteMin(Q)
    y=DeleteMin(Q)
    Construct node z with nodes x and y as children
    f(z) = f(x) + f(y)
    Insert(Q, z)
OD
root=DeleteMin(Q)
```

Example:



- Running time: $O(|C| \log |C|)$

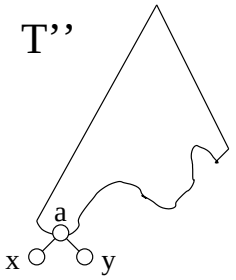
- Note: Algorithm is greedy:
  - Let cost of producing node z be $f(x) + f(y)$
  - We can show that the cost $B(T)$ of T is sum of cost of producing each node

    $\Longrightarrow$ We always make greedy choice (produce cheapest node)

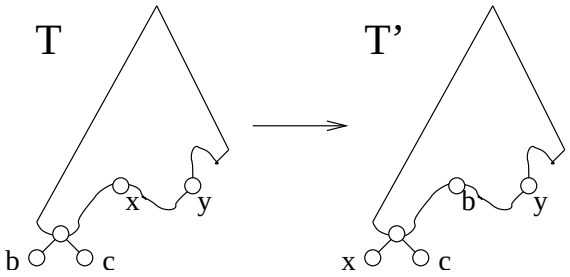
1.3 Proof of correctness (optimality) for Huffman codes

The proof that our Huffman algorithm produces an optimal code follows from the following two lemmas, using induction on the size of C.

Lemma 1 *Let x and y be the two minimum frequency characters in C. Then there exists an optimum prefix code T'' where the codes for x and y have same length and differ only in the last bit.*



- Consider optimum tree T. We need to transform T to T''
- Let b and c be two sibling leaves in T at maximum depth and assume $f(x) \leq f(b)$ and $f(y) \leq f(c)$
- Produce T' by exchanging x and b



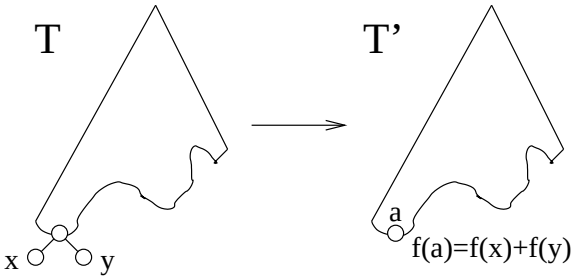
We have:

$$\begin{aligned} B(T) - B(T') &= \sum f(z)d_T(z) - \sum f(z)d_{T'}(z) \\ &= f(x)d_T(x) + f(b)d_T(b) - f(x)d_{T'}(x) - f(b)d_{T'}(b) \\ &= f(x)d_T(x) + f(b)d_T(b) - f(x)d_T(b) - f(b)d_T(x) \\ &= (f(b) - f(x))(d_T(b) - d_T(x)) \\ &\geq 0 \end{aligned}$$

- We produce T'' from T' by exchanging y and c. We can prove in a similar way that $B(T') - B(T'') \geq 0$

  $\Longrightarrow \quad B(T) \geq B(T') \geq B(T'')$
- But since T is optimum, we have $B(T) \leq B(T'')$, and therefore $B(T) = B(T'')$.

Lemma 2 *For alphabet C, let x and y be the two minimum-frequency characters in C. Let C' be the alphabet C with characters x and y removed and replaced by a; that is $C' = C - \{x, y\} \cup \{a\}$. Let T' be a tree representing an optimum prefix code for C'. Then we can get an optimal tree T for C by replacing the leaf node for a in T' by an internal node having leaf children x and y.*



- We have:

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

$$\Longrightarrow \quad B(T) = B(T') + f(x) + f(y)$$

- If there exists other tree T'' for C with $B(T'') < B(T)$, then by the previous lemma, T'' has x and y as leaf siblings. We could therefore produce a tree for C' with cost $B(T'') - f(x) - f(y) < B(T) - f(x) - f(y) = B(T')$, which is a contradiction, since T' is optimum for C'. Therefore, T' must be optimum for C.

1.4 Dynamic Huffman Trees

There's a near article (that I couldn't have written better myself!) available at

http://www.cs.duke.edu/~jsv/Papers/Vit87.jacm.pdf
http://www.cs.duke.edu/~jsv/Papers/Vit87.jacm.ps.gz

on the data structures problem of maintaining a dynamic Huffman tree. That is, at each time step a new character appears, which increments the weight of the corresponding leaf in the tree, and the problem is to readjust the tree to maintain the Huffman optimality, in time proportional to the encoding length of the new character (namely, the length of the root-to-leaf path to the character's leaf in the old tree).

Here's an excerpt from the intro of the paper: One motivation for dynamic Huffman coding in the context of data compression is that the encoding and gathering of statistics are done online in a single pass. The standard Huffman method makes two passes over the data: one pass to collect frequency counts of the letters in the message, followed by the construction of a Huffman tree and transmission of the tree to the receiver; and a second pass to encode and transmit the letters themselves, based upon the static tree structure. This two-pass nature causes delay when used for network communication, and in file compression applications the extra disk accesses can slow down the algorithm. Faller [1973] and Gallager [1978] independently proposed a one-pass scheme, later improved substantially by Knuth [1985], for constructing dynamic Huffman codes. The binary tree that the sender uses to encode the

$(t+1)$st letter in the message (and that the receiver uses to reconstruct the $(t+1)$st letter) is a Huffman tree for the first t letters of the message. Both sender and receiver start with the same initial tree and thereafter stay synchronized; they use the same algorithm to modify the tree after each letter is processed. Thus there is never need for the sender to transmit the tree to the receiver, unlike the case of the two-pass method. The processing time required to encode and decode a letter is proportional to the length of the letter's encoding, so the processing can be done in real time.

Of course, one-pass methods are not very interesting if the number of bits transmitted is significantly greater than with Huffman's two-pass method. The paper I mentioned gives the first analytical study of the efficiency of dynamic Huffman codes.