Greedy Algorithms

The Greedy strategy is (just like D&C or DP) a *design paradigm*.

General idea: Greedy algorithms always make choices that "look best at the moment."

They do not always yield optimal results, but in many cases they do (and if not, then often "pretty close" to optimal).

Huffman codes

Used for compressing data (savings 20% to 90%).
Data is considered to be a sequence of characters.

Huffman's greedy algorithm

- computes frequency of occurrence of characters, and
- assigns binary strings to characters: the more frequent a character, the shorter the string

Results in binary character code ("code").

Consider file of length $100,000$, containing only characters a,b,c,d,e,f, and the following frequencies (in thousands).

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

With **fixed length codes**, exact code of each character does not matter (w.r.t. length). For six characters, we need three bits per character, a total of $300,000$ bits.

With **variable length codes**, assignment *does* matter. Consider following code.

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

Resulting length (in bits) now is

$$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1,000 = 224,000$$

Prefix codes

No codeword is prefix of some other codeword!

Encoding is simple: just concatenate codewords. Using

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

the code for "deaf" is 111110101100.

Prefix codes simplify **decoding**, they parse uniquely.

Binary Tree Representation

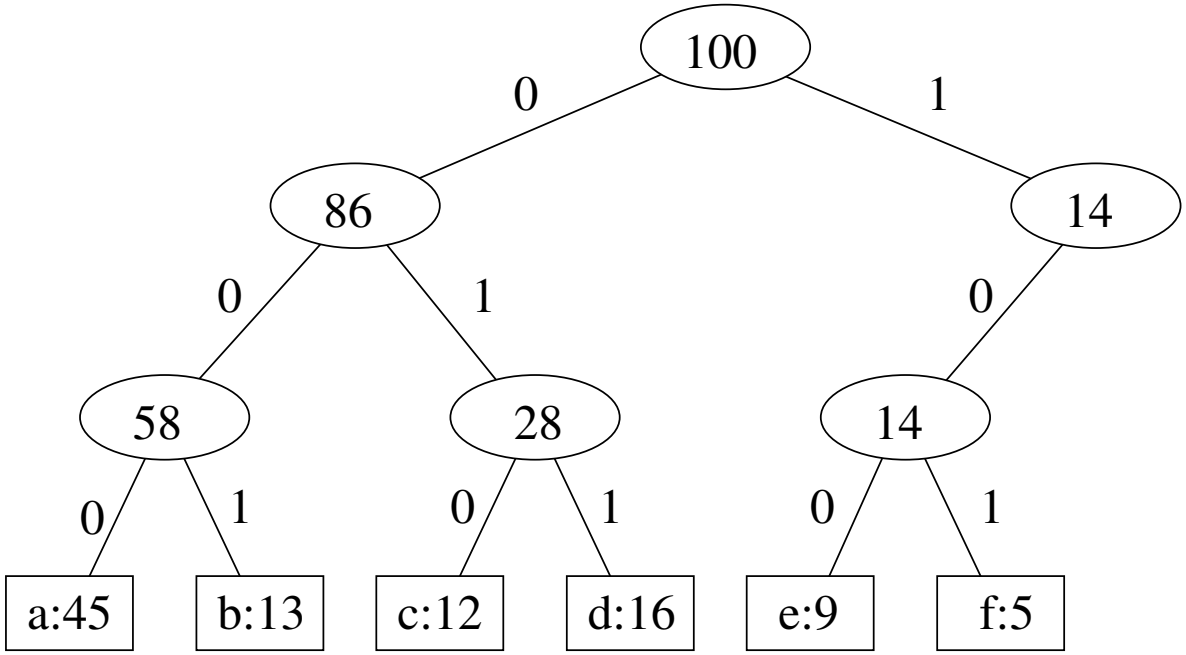
We need convenient representation for prefix codes.

Use **binary tree:**

- leaves represent characters,
- interpret binary codeword for a character as path from root to corresponding leaf; 0 means “left”, 1 means “right”.

Example: fixed length code

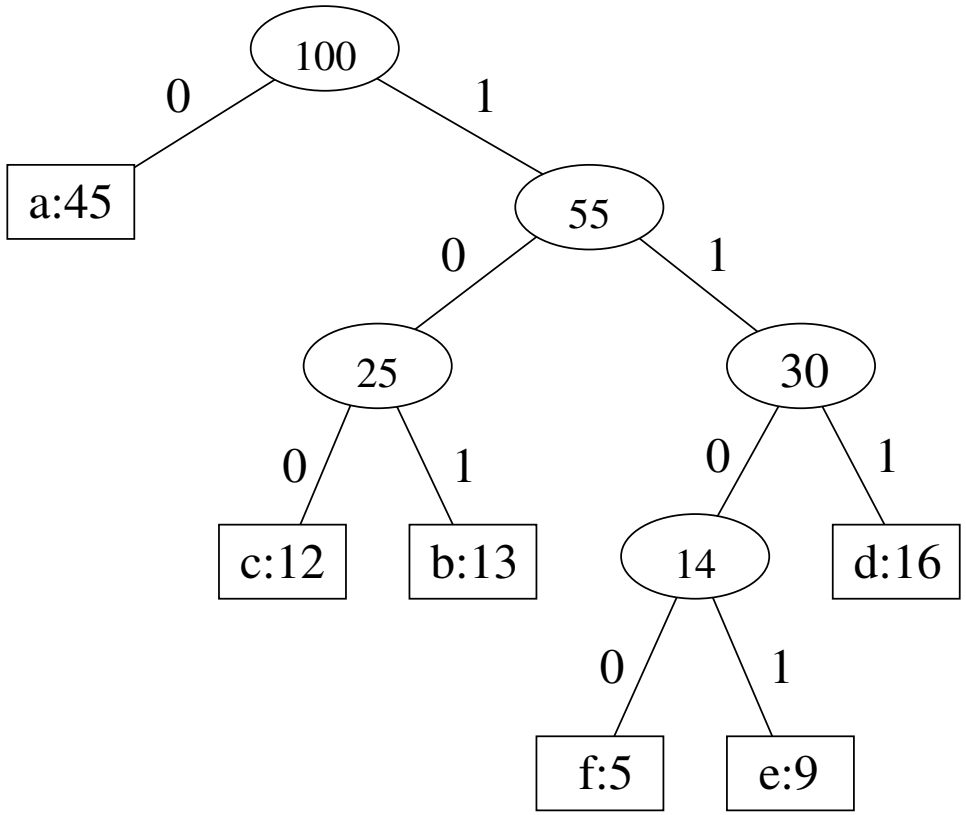
character	a	b	c	d	e	f
frequency	45	13	12	16	9	5
codeword	000	001	010	011	100	101



Leaves are labeled with character and frequency, interior vertices with sum of frequencies of leaves in sub-tree.

Example: variable length code

character	a	b	c	d	e	f
frequency	45	13	12	16	9	5
codeword	0	101	100	111	1101	1100



Optimal Codes

Optimal codesare always represented by a **full binary tree**, where every non-leaf vertex has two children (exercise).

- Fixed-length example therefore non-optimal (obviously, we already have seen better one).

- Full binary trees $\Longrightarrow$ if C is our alphabet, then

$$|C| \text{ leaves and } |C| - 1 \text{ internal vertices}$$

(exercise).

Cost Model

Given tree T corresponding to a prefix code, we can compute number of bits to encode a file.

For $c \in C$, $f(c)$ denotes frequency, and $d_T(c)$ denotes depth of c's leaf.

Then, cost of T is

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

Note: $d_T(c)$ is also length of c's codeword!

Excursion: Min-priority queues

Huffman's algorithm uses a **min-priority queue** (a heap with certain properties). Relevant operations:

`Build-Min-Heap`: constructs the heap; takes $O(n)$ for n items.

`Extract-Min`: finds the minimal item and removes it from heap; takes $O(\log n)$ per operation.

`Insert`: inserts new items into queue; takes $O(\log n)$.

Idea of the Algorithm

The idea of Huffman's algorithm is as follows.

- Tree is built **bottom-up**.
- Begin with $|C|$ leaves, then do $|C| - 1$ **merging operations** to create final tree.
- In each merger,
  - extract two **least-frequent** objects to merge;

  Result: new object whose frequency is sum of frequencies of two merged objects.

Huffman's greedy algorithm

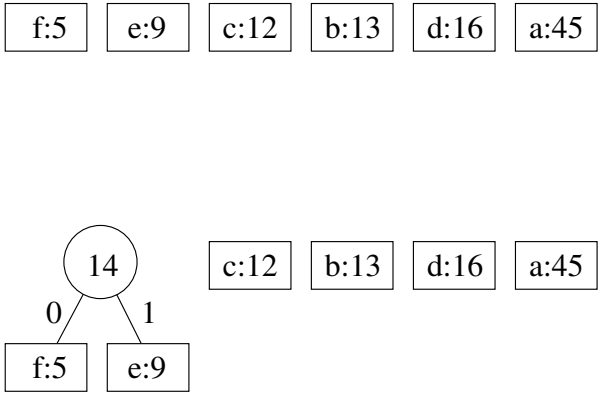
```
1: n ← |C|
2: Q ← C        {Build-Min-Heap}
3: for i ← 1 to n − 1 do
4:     allocate new object z
5:     left[z] ← x ← Extract-Min(Q)
6:     right[z] ← y ← Extract-Min(Q)
7:     f[z] ← f[x] + f[y]
8:     Insert(Q, z)
9: end for
```

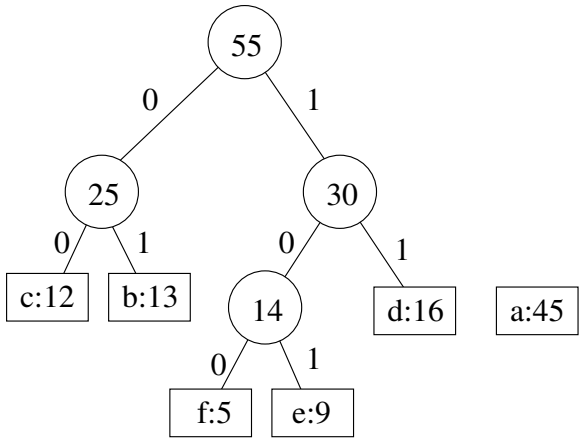
Running time

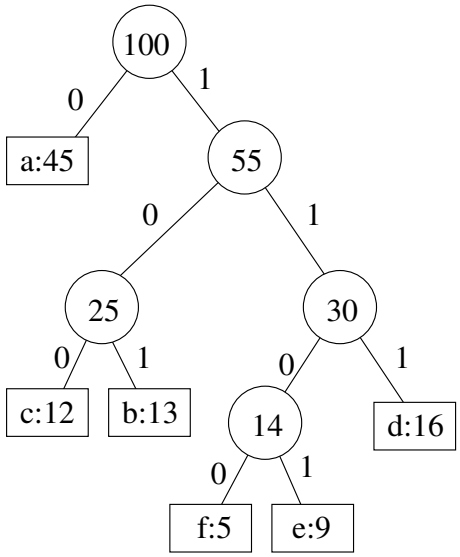
Initialization takes $O(n)$ and each heap operation in loop takes $O(\log n)$.

Total running time is therefore $O(n \log n)$.

Example


f:5
e:9
c:12
b:13
d:16
a:45
14
0
1
f:5
e:9
c:12
b:13
d:16
a:45
14
0
1
f:5
e:9
25
0
1
c:12
b:13
d:16
a:45
30
0
1
14
d:16
0
1
f:5
e:9
25
0
1
c:12
b:13
a:45


55
0
1
25
30
0
1
0
1
c:12
b:13
14
d:16
a:45
0
1
f:5
e:9


100
0
1
a:45
55
0
1
25
30
0
1
0
1
c:12
b:13
14
d:16
0
1
f:5
e:9

Correctness

Definition: Let C be alphabet, character $c \in C$ has frequency $f[c]$.

Lemma 1. Let x and y two characters in C with lowest frequency. Then there is optimal prefix code for C in which codewords for x and y have same length and differ in only one bit.

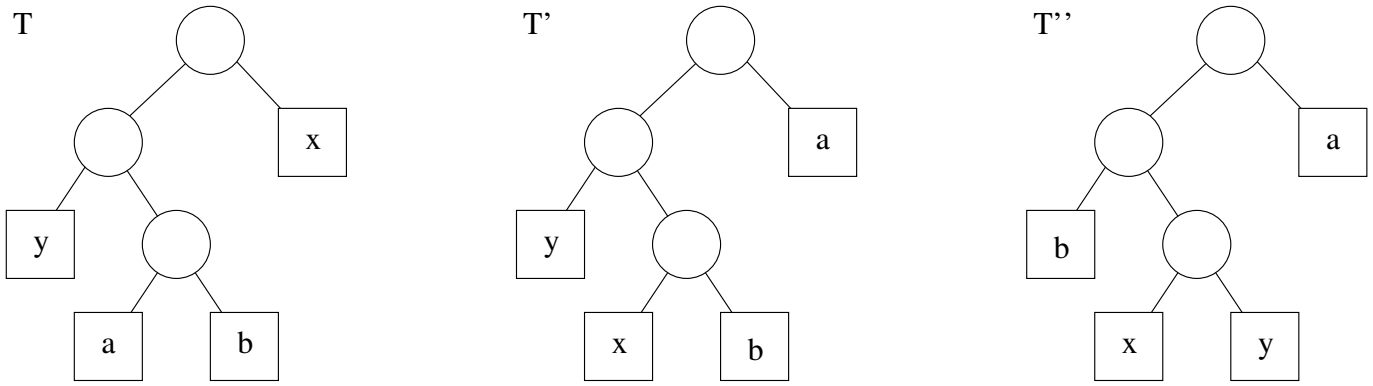
In words: building up tree can w.l.o.g. begin with greedy choice of merging lowest-frequency characters.

Proof.

- Let T be any optimal tree.
- Let a and b characters that are sibling leaves of maximum depth in T.
- W.l.o.g. $f[a] \leq f[b]$ and $f[x] \leq f[y]$.

Recall: $f[x]$ and $f[y]$ are the two lowest frequencies. Thus, $f[x] \leq f[a]$ and $f[y] \leq f[b]$.

Now exchange positions of a and x ($\rightarrow$ T') and then in T' exchange positions of b and y ($\rightarrow T''$).



$$
\begin{aligned}
& B(T) - B(T') \\
= & \sum_{c \in C} f(c) d_T(c) - \sum_{c \in C} f(c) d_{T'}(c) \\
= & f[x] d_T(x) + f[a] d_T[a] - f[x] d_{T'}(x) - f[a] d_{T'}(a) \\
= & f[x] d_T(x) + f[a] d_T[a] - f[x] d_T(a) - f[a] d_T(x) \\
= & (f[a] - f[x]) \cdot (d_T(a) - d_T(x)) \geq 0.
\end{aligned}
$$

Last inequality holds since $f[a] - f[x] \geq 0$ and $d_T(a) - d_T(x) \geq 0$.

Similarly, $B(T') - B(T'') \geq 0$.

Therefore, $B(T'') \leq B(T') \leq B(T)$.
T **is optimal:** $B(T) \leq B(T'')$.

Thus, $B(T) = B(T'')$ and T'' is optimal. Also: T'' has required form!

Lemma 2.

Let x, y be two characters in C with min. frequency.
Let $C' = C - \{x, y\} \cup \{z\}$ with $f[z] = f[x] + f[y]$
Let T' be any tree representing opt. prefix code for C'.

Then T, obtained from T' by replacing leaf z with internal vertex having x and y as children, represents optimal prefix code for C.

Proof. For each $c \in C - \{x, y\}$, $d_T(c) = d_{T'}(c)$, hence $f[c]d_T(c) = f[c]d_{T'}(c)$.

First we show: $B(T') = B(T) - f[x] - f[y]$

$d_T(x) = d_T(y) = d_{T'}(z) + 1$ (we have replaced leaf repr. z with internal vertex with x, y as children).

We have

$$\begin{aligned} & f[x]d_T(x) + f[y]d_T(y) \\ = \;& f[x] \cdot (d_{T'}(z) + 1) + f[y] \cdot (d_{T'}(z) + 1) \\ = \;& (f[x] + f[y]) \cdot (d_{T'}(z) + 1) \\ = \;& f[z] \cdot (d_{T'}(z) + 1) \\ = \;& f[z]d_{T'}(z) + f[z] \\ = \;& f[z]d_{T'}(z) + (f[x] + f[y]) \end{aligned}$$

With $f[x]d_T(x) + f[y]d_T(y) = f[z]d_{T'}(z) + (f[x] + f[y])$,

$$\begin{aligned} & B(T) = B(T') + f[x] + f[y] \\ \iff \;& B(T') = B(T) - f[x] - f[y] \end{aligned}$$

Rest of the proof of lemma **by contradiction.**

Suppose T does **not** represent optimal prefix code for C. Then $\exists T''$ with $B(T'') < B(T)$.

W.l.o.g. (by first lemma), T'' has x, y as siblings. Let T''' be T'' with common parent of x, y replaced by leaf z with $f[z] = f[x] + f[y]$. Then,

$$\begin{aligned} B(T''') &= B(T'') - f[x] - f[y] \\ &< B(T) - f[x] - f[y] \\ &= B(T') \end{aligned}$$

Contradiction since T' was assumed to be optimal!

Elements of the greedy strategy

To be taken with a grain of salt; this is **not** the holy grail.

From the book:

> How can one tell if a greedy algorithm will solve a particular optimization problem? **There is no way in general**, but the greedy-choice property and optimal substructure are the two key ingredients.

Other authors claim different things.

Greedy-choice property

A globally optimal solution can be arrived at by making a locally optimal (greedy) choice.

We make the choice that looks best without considering (or modifying) results from subproblems.

This (kinda) was our first lemma.

Optimal substructure

An optimal solution to the problem contains within it optimal solutions to subproblems.

This was our second lemma; optimal solution for C (with x, y) contained optimal solution for C' with z instead of x, y.

Another example: Scheduling

Given: n jobs $j_1, \ldots, j_n$, service times $t_1, \ldots, t_n$, and one machine.

Goal: minimize average time a job spends in system

Since n is fixed, problem is **equivalent** to minimizing

$$T = \sum_{i=1}^{n} (\text{time in system for customer } i),$$

what happens to be just n times the **average time in system**.

Example

Three customers, $t_1 = 5$, $t_2 = 10$, $t_3 = 3$. There are $3! = 6$ possible orders:

order	T	
1 2 3	$5 + (5 + 10) + (5 + 10 + 3) = 38$	
1 3 2	$5 + (5 + 3) + 5 + 3 + 10) = 31$	
2 1 3	$10 + (10 + 5) + (10 + 5 + 3) = 43$	
2 3 1	$10 + (10 + 3) + (10 + 3 + 5) = 41$	
3 1 2	$3 + (3 + 5) + (3 + 5 + 10) = 29$	opt
3 2 1	$3 + (3 + 10) + (3 + 10 + 5) = 34$	

Note: optimal solution when jobs sorted in order of increasing service times.

Idea of greedy algorithms is to do whatever seems best **at the moment**.

Suppose we already have scheduled the first ℓ jobs. What to do in order to have T as small as possible?

- Pick "cheapest" job available for the $(\ell + 1)$-st one!

Theorem. This greedy algorithm (at each stage pick job with shortest service time) is optimal.

Proof. Let $P = p_1 p_2 \cdots p_n$ be any permutation of $\{1, \ldots, n\}$, let $s_i = t_{p_i}$ (service time of i-th job w.r.t. P). Then

$$\begin{aligned} T(P) &= s_1 + (s_1 + s_2) + (s_1 + s_2 + s_3) + \cdots \\ &= n s_1 + (n-1) s_2 + (n-2) s_3 + \cdots \\ &= \sum_{k=1}^{n} (n - k + 1) \cdot s_k \end{aligned}$$

Suppose P does **not** arrange jobs in order of increasing service time. Then there must be a, b with $a < b$ and $s_a > s_b$ (a-th job is served before b-th although a-th needs more service time than b-th).

Now we swap positions of a-th and b-th jobs.

$$T(P') = (n-a+1)s_b + (n-b+1)s_a + \sum_{k \in \{1,n\}-\{a,b\}} (n-k+1)s_k$$

(job with s_b is in position a, and vice versa). Now

$$\begin{aligned} & T(P) - T(P') \\ &= (n-a+1)s_a + (n-b+1)s_b - \\ &\quad (n-a+1)s_b - (n-b+1)s_a \\ &= (n+1)(s_a + s_b - s_b - s_a) + a(s_b - s_a) + b(s_a - s_b) \\ &= b(s_a - s_b) - a(s_a - s_b) \\ &= (b-a) \cdot (s_a - s_b) \\ &> 0 \end{aligned}$$

since $b > a$ and $s_a > s_b$.

We can **improve** any schedule by swapping two jobs according to rule *shortest-service-time-first*. This proves the theorem.