SFU CMPT-307 2008-2 Lecture: Week 12

Ján Maňuch

E-mail: jmanuch@sfu.ca

Lecture on July 29, 2008, 5.30pm-8.20pm

Greedy algorithms

The Greedy strategy is (just like Divide&Conquer or Dynamic Programming) a *design paradigm*.

Basic idea: Greedy algorithms always make choices that "look best at the moment" — *locally* optimal solutions.

This does not always yield a *globally* optimal solution, but in many cases it does (and if not, then we are often "pretty close" to optimal).

The running time is very good (often linear).

Activity selection problem

Consider:
A set of n **activities** $S = \{a_1, \ldots, a_n\}$ sharing a common resource. Each activity a_i has a **star time** s_i and a **finish time** f_i, where
$0 \leq s_i < f_i < \infty$.

Example. Activities = lectures. A common resource = lecture room.

Activities a_i and a_j are **compatible** if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap, i.e., if either $s_i \geq f_j$ or $s_j \geq f_i$.

Activity selection problem: select a maximum-size subset of S of mutually compatible activities.

We will assume that the activities are **sorted** in monotically increasing order of finish time:

$$f_1 \leq f_2 \leq \cdots \leq f_n$$

Example.

We have 6 lectures:

a_1 from 8 till 11	a_2 from 10 till 12
a_3 from 14 till 15	a_4 from 11 till 14
a_5 from 12 till 14	a_6 from 15 till 16

But only one lecture room. We want to choose as many lectures which do not overlap as possible.

Dynamic programming approach.

1. The optimal substructure

subproblems:

$$S_{ij} = \{a_k \in S; \quad f_i \leq s_k < f_k \leq s_j\}$$

i.e., S_{ij} contains all activities which are compatible with

- all activities that finish no later than a_i ($a_1, \ldots a_i$)
- all activities that start no earlier than a_j

subproblem $\mathcal{S}_{ij}$ is to find a maximal-size subset of pairwise compatible activities of S_{ij}

add activities a_0 and a_{n+1} such that $f_0 = 0$ and $s_{n+1} = \infty$; then the main problem is equal to $\mathcal{S}_{0,n+1}$.

Property. if $i \geq j$, then $S_{ij} = \emptyset$

Proof: Suppose that $i \geq j$ and that there exists $a_k \in S_{ij}$.
Since $i \geq j$, $f_i \geq f_j$.
Since $a_k \in S_{ij}$, $f_i \leq s_k < f_k \leq s_j < f_j$.
A contradiction.

Hence, our **space of subproblems** contains $\mathcal{S}_{ij}$ for all $0 \leq i < j \leq n+1$.

optimal substructure:

Let $A_{ij} \subset S_{ij}$ is an optimal solution for subproblem $\mathcal{S}_{ij}$. Assume that $a_k \in A_{ij}$. Then A_{ij} contains solutions $A_{ik} = A_{ij} \cap S_{ik}$ and $A_{kj} = A_{ij} \cap S_{kj}$ to $\mathcal{S}_{ik}$ and $\mathcal{S}_{kj}$, respectively.

These subsolutions are *optimal*. Again: use "cut-and-paste" argument.
For instance, if A_{ik} has a better solution A'_{ik} for $\mathcal{S}_{ik}$, then we can replace A_{ik} with A'_{ik} in A_{ij} forming a better solution for $\mathcal{S}_{ij}$.

Hence, if we know that a solution A_{ij} of $\mathcal{S}_{ij}$ contains a_k, then A_{ij} can be obtained as

$$A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$$

2. A recursive solution

Consider an $a_k \in A_{ij}$, where A_{ij} is an optimal solution of $\mathcal{S}_{ij}$. Then $f_i < f_k < f_j$, i.e., $k = i+1, \ldots, j-1$.

We don't know which a_k is in A_{ij}, so we have to consider all possibilities.

Let $c[i,j]$ be the maximal number of pairwise compatible activities in S_{ij}. Then

$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset, \\ \max_{i<k<j}\{c[i,k] + 1 + c[k,j]\} & \text{otherwise.} \end{cases}$$

Now, it's easy to make Dynamic Programming algorithm working in $\mathcal{O}(n^3)$ time.

Greedy properties

Consider a nonempty S_{ij}. Let a_m be the first activity in S_{ij} in our order, i.e., the one with the earliest finish time:

$$f_m = \min\{f_k; \quad a_k \in S_{ij}\}.$$

Then

1. The set S_{im} is empty. So, the subproblems $\mathcal{S}_{im}$ is trivial.
2. There is a maximum-size solution to the subproblems $\mathcal{S}_{ij}$ containing a_m.

Proof.

1. Suppose that S_{im} is non-empty, i.e., there exists $a_k \in S_{im}$. Then $f_i \leq s_k < f_k \leq s_m$. By the definition of a_m, $f_m \leq f_k$. A contradiction: $f_k \leq s_m < f_m \leq f_k$.
2. Let A_{ij} be an optimal solution to $\mathcal{S}_{ij}$. If it contains a_m, we are done. Otherwise, let a_k be the first activity in A_{ij} (the one with the earliest finish time).

2. (continued)
   Now, let's replace a_k with a_m:

$$A'_{ij} = A_{ij} - \{a_k\} \cup \{a_m\}.$$

   We will prove that A'_{ij} is a solution to $\mathcal{S}_{ij}$ (and hence also an optimal solution we are looking for).
   It's enough to check that the activity a_m is compatible with other activities in A'_{ij}.
   Take $a_l \in A'_{ij} - \{a_m\} = A_{ij} - \{a_k\}$. Since, A_{ij} is a solution, a_l and a_k are compatible: either $f_l \leq s_k < f_k$ (not possible) or $f_k \leq s_l$ (ok). Then $f_m \leq f_k \leq s_l$, i.e., a_m and a_l are also compatible.

Consider the recursive solution:

$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset, \\ \max_{i<k<j}\{c[i,k] + 1 + c[k,j]\} & \text{otherwise.} \end{cases}$$

Now, we know that we can take $k = m$ (by 2.). And in such a case, the first subproblem $\mathcal{S}_{im}$ is trivial, i.e., $c[i,m] = 0$ (by 1.).

$j - i - 1$ choices $\longrightarrow$ 1 choice 2 subproblems $\longrightarrow$ 1 subproblem

The algorithm:

— choose the first a_m in S_{ij} (**locally optimal solution:** leave as much space for the remaining activities as possible)

— $A_{ij} = A_{mj} \cup \{a_m\}$ is an optimal solution to $\mathcal{S}_{ij}$, where A_{mj} is an optimal solution to $\mathcal{S}_{mj}$.

— $c[i, j] = c[m, j] + 1$.

Recursive algorithm:

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

```
1: m ← i + 1
2: /* find the first activity in S_ij */
3: while m < j and s_m < f_i do
4:     m ← m + 1
5: end while
6: if m < j then
7:     return {a_m} ∪ RAS(s, f, m, j)
8: else
9:     return ∅
10: end if
```

Note. Index j never changes. Hence, if we start with the problems $\mathcal{S}_{0,n+1}$ in the beginning, and try to solve the problem recursively (top-to-bottom fashion), then we only encounter the subproblems $\mathcal{S}_{ij}$ with $j = n + 1$.

We can write **iterative** version of the algorithm working in linear time.

Greedy-Activity-Selector(s, f)

1: $n \leftarrow \text{length}(s)$
2: $A \leftarrow \{a_1\}$
3: $i \leftarrow 1$
4: **for** $m \leftarrow 2$ **to** n **do**
5: **if** $s_m \geq f_i$ **then**
6: $A \leftarrow A \cup \{a_m\}$
7: $i \leftarrow m$
8: **end if**
9: **end for**
10: **return** A

Explanations:

– the first activity in $S_{0,n+1}$ is a_1

– $i =$ index of latest addition to A, i.e., we are solving the subproblem $\mathcal{S}_{i,n+1}$

– the loop finds the first a_m in $S_{i,n+1}$

– and adds it to A

Assignment Problem 12.1. (deadline: August 5, 5:30pm)
Suppose that instead of always selecting the first activity to finish, we select the last activity to start from S_{ij}. Describe the greedy properties of this approach, and prove that it yields an optimal solution. Write an iterative version of the new algorithm.

Assignment Problem 12.2. (deadline: August 5, 5:30pm)
Show on an example, that the greedy approach of selecting the activity of least duration from S_{ij} does not work.

Designing a greedy algorithm

- Find a recursive description of the problem in which after making a choice we have only 1 subproblem to solve.
- Prove that there is an optimal solution that makes the greedy choice (a choice that seems locally optimal, e.g., selecting interval with the earliest end; or selecting the shortest interval).
- Prove that an optimal solution to subproblem combined with the greedy choice gives an optimal solution to the original problem ("optimal substructure").

Huffman codes

Used for compressing data (typically savings of 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 a binary character code ("code").

Comparing **variable length code** *with a* **fixed length 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

We only consider **prefix codes**: no codeword is a 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**. No codeword is prefix of any other, so codewords are unambiguous (they parse uniquely).

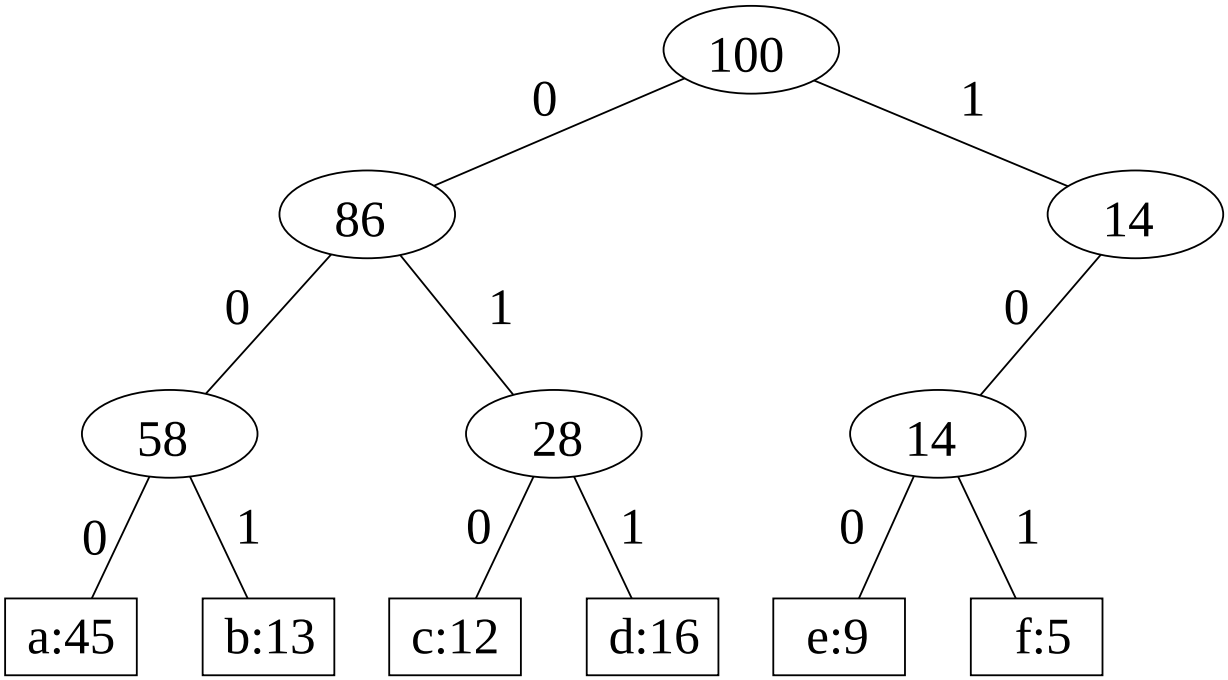
We need convenient representation for prefix codes.

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

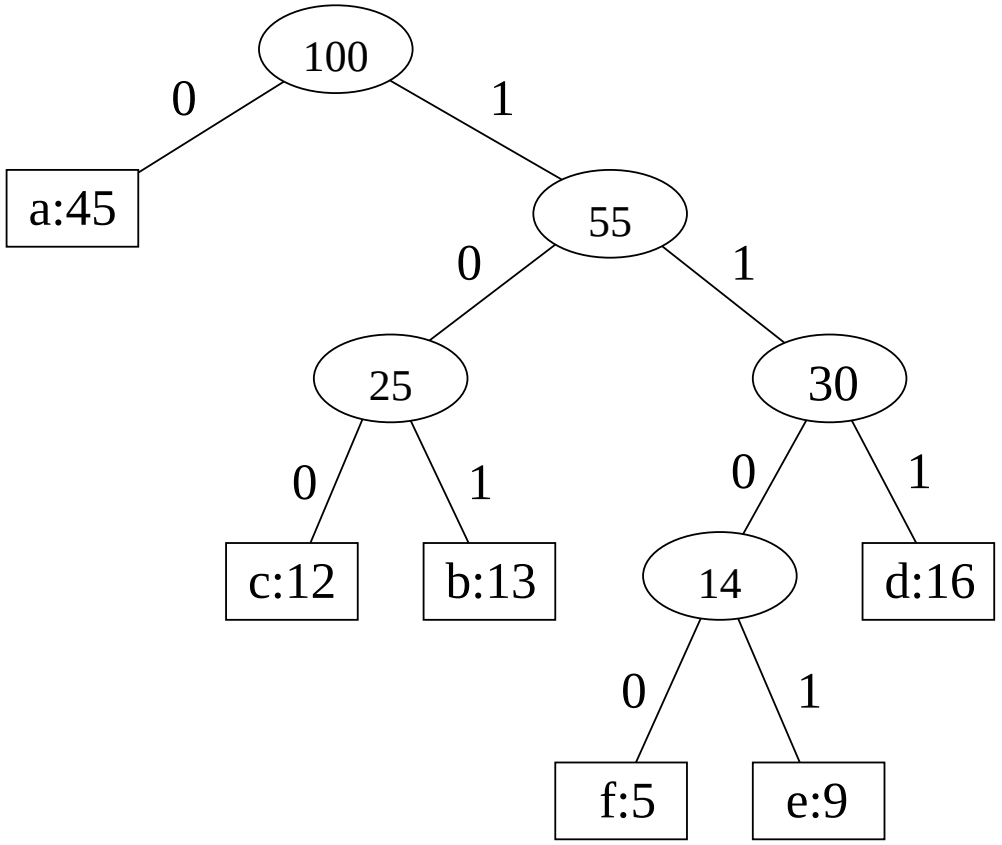
character	a	b	c	d	e	f
frequency	45	13	12	16	9	5
codeword	000	001	010	011	100	101



Leaves are labeled with its character and its frequency, internal vertices with the sum of frequencies of leaves in its 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



Given a tree T corresponding to a prefix code, we can compute the number of bits to encode a file as follows. For $c \in C$, $f(c)$ denotes its frequency, and $d_T(c)$ denotes its depth in T (hence, $d_T(c)$ is also the length of the code of c).
The cost of T is

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

Optimal code for a file is always represented by a **full binary tree**, every non-leaf node has two children.

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

Full binary trees $\Longrightarrow$ if C is our alphabet, then the tree has $|C|$ leaves and $|C| - 1$ internal vertices.

Assignment Problem 12.3. (deadline: August 5, 5:30pm)
Show that a full binary tree with n leaves has $n - 1$ internal vertices.

Hint: use induction.

Min-priority queues

Huffman's algorithm uses a min-priority queue (implemented as a heap). Recall the 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`: insert a new item into queue; takes $O(\log n)$.

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 a final tree.
- In each merger, extract two least-frequent objects to merge; result is a new object whose frequency is the sum of frequencies of two merged objects.

Huffman's greedy algorithm

1: $n \leftarrow |C|$
2: $Q \leftarrow C$ `/*Build-Min-Heap*/`
3: **for** $i \leftarrow 1$ **to** $n - 1$ **do**
4: allocate a new node z
5: left$[z] \leftarrow x \leftarrow$ `Extract-Min`(Q)
6: right$[z] \leftarrow y \leftarrow$ `Extract-Min`(Q)
7: $f[z] \leftarrow f[x] + f[y]$
8: `Insert`(Q, z)
9: **end for**

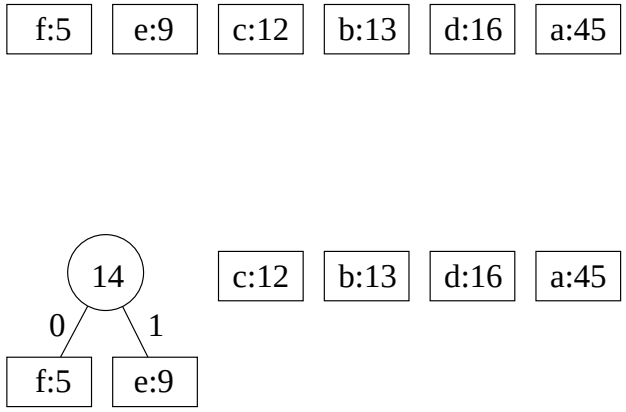
— the algorithm is building a tree

— z is a new internal node created by the merger of x and y, which are now its children

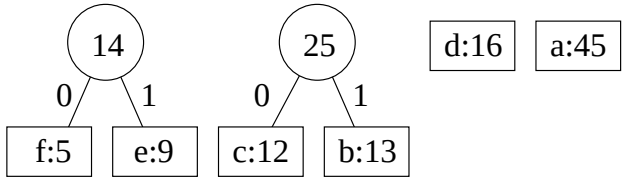
Running time:

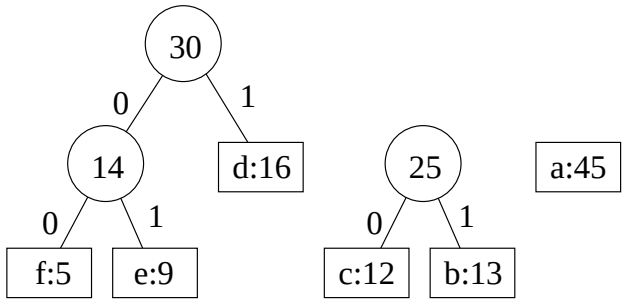
Initialization takes $O(n)$.
Each heap operation in loop takes $O(\log n)$.
Total running time is therefore $O(n \log n)$.

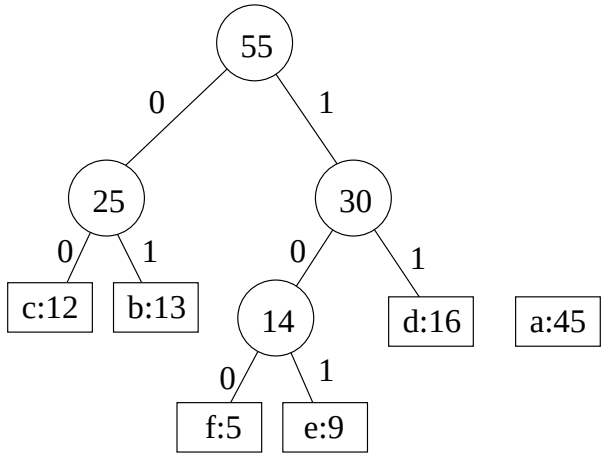
Example

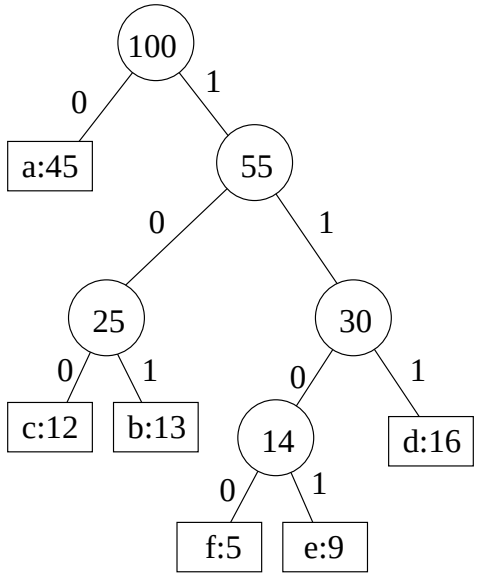

f:5
e:9
c:12
b:13
d:16
a:45

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

Assignment Problem 12.4. (deadline: August 5, 5:30pm)
Prove that the total cost $B(T)$ of a tree T can be computed as the sum of frequencies of all internal nodes (assigned by the algorithm).

Correctness

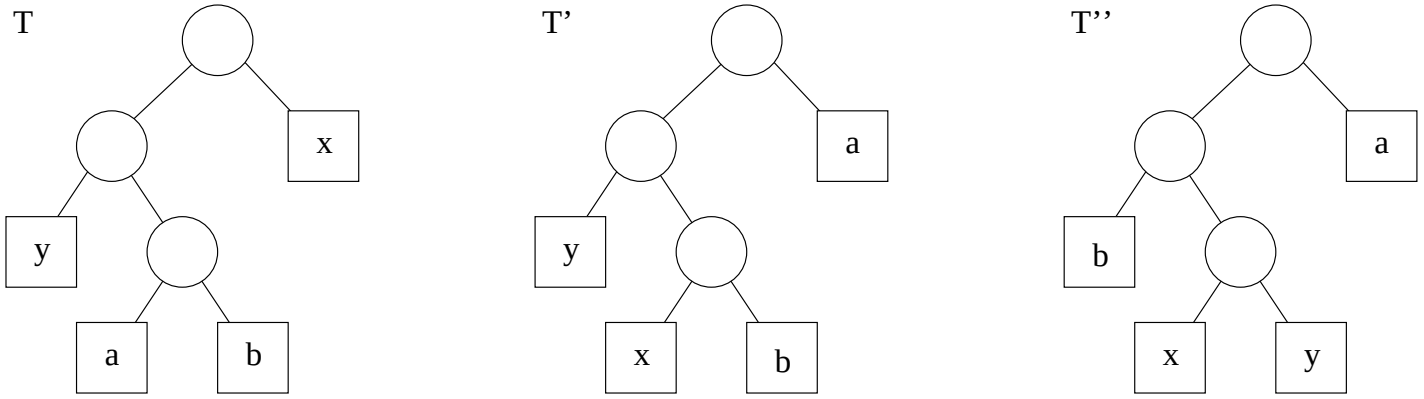
Lemma 1. *(Greedy choice property)*
Let C be alphabet, each character $c \in C$ has a frequency $f[c]$. Let x and y two characters in C with the lowest frequency. Then there is an optimal prefix code for C in which the codewords for x and y are the **sibling leaves** (have the same length and differ only in the last bit).

In other words: building up an optimal tree can, without loss of generality, begin with the greedy choice of merging two lowest-frequency characters.

Why is this a greedy choice? By Homework 12.4, the total cost equals to the sum of costs of mergers (cost = frequency f). Hence, we are choosing a merger with the minimal cost at each step.

Proof. Let T be any optimal tree. Let a and b be two characters that are sibling leaves of maximum depth in T. Without loss of generality, $f[a] \leq f[b]$ and $f[x] \leq f[y]$. Recall that $f[x]$ and $f[y]$ are two lowest frequencies (in order). $f[a]$ and $f[b]$ are arbitrary frequencies (also in order). 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 ($\rightarrow T''$).



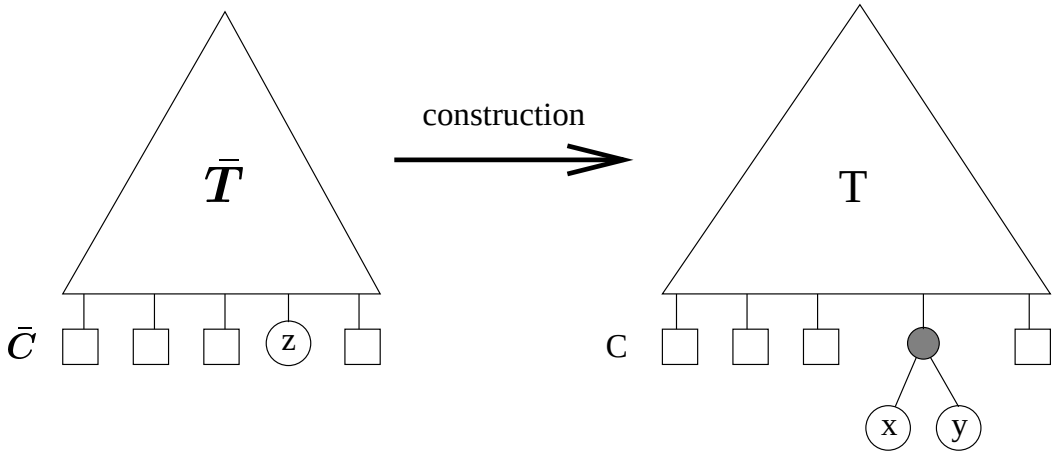
T'' is a tree we are looking for, codewords for x and y have the same length and differ only in the last bit. But we have to prove that it's an optimal tree.

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

because $f[a] - f[x] \geq 0$ and $d_T(a) - d_T(x) \geq 0$.

Similarly, $B(T') - B(T'') \geq 0$. Therefore, $B(T'') \leq B(T') \leq B(T)$ and (T is optimal) $B(T) \leq B(T'')$.
Thus, $B(T) = B(T'')$, and T'' is optimal, with x and y as sibling leaves of maximum depth.
We are done.

Lemma 2. *(optimal substructure property)*
Given an alphabet C with frequencies $f[c]$ for $c \in C$. Let x, y be two characters in C with minimal frequencies. Let $\bar{C} = C - \{x, y\} \cup \{z\}$ be a new alphabet with the same frequencies as C except $f[z] = f[x] + f[y]$.
Let $\bar{T}$ be any tree representing an optimal prefix code for $\bar{C}$.
Then T, obtained from $\bar{T}$ by replacing leaf z with the internal vertex having x and y as children, represents an optimal prefix code for C.



The lemma shows the optimality of Huffman's algorithm, by showing that each step of the algorithm is optimal:
– merge two characters with lowest frequencies (greedy choice)
– compute optimal solution for $\bar{C}$
– combine the above

Proof of Lemma 2.

Let's compare the cost of T for C and the cost of $\bar{T}$ for $\bar{C}$. For each character $c \in C - \{x, y\}$, $d_T(c) = d_{\bar{T}}(c)$, hence $f[c]d_T(c) = f[c]d_{\bar{T}}(c)$. Hence we need only compare what x, y in T and z in $\bar{T}$ contribute to the cost.

Since $d_T(x) = d_T(y) = d_{\bar{T}}(z) + 1$ (we have replaced a leaf z with an internal vertex with x and y as its children). Hence,

$$\begin{aligned} & f[x]d_T(x) + f[y]d_T(y) \\ = \quad & f[x] \cdot (d_{\bar{T}}(z) + 1) + f[y] \cdot (d_{\bar{T}}(z) + 1) \\ = \quad & (f[x] + f[y]) \cdot (d_{\bar{T}}(z) + 1) \\ = \quad & (f[x] + f[y]) \cdot d_{\bar{T}}(z) + f[x] + f[y] \\ = \quad & f[z]d_{\bar{T}}(z) + f[x] + f[y] \end{aligned}$$

Hence,

$$
\begin{aligned}
B(T) &= \sum_{c \in C} f[c] d_T(c) \\
&= \sum_{c \in C - \{x,y\}} f[c] d_T(c) + f[x] d_T(x) + f[y] d_T(y) \\
&= \sum_{c \in \bar{C} - \{z\}} f[c] d_{\bar{T}}(c) + f[z] d_{\bar{T}}(z) + f[x] + f[y] \\
&= \sum_{c \in \bar{C}} f[c] d_{\bar{T}}(c) + f[x] + f[y] \\
&= B(\bar{T}) + f[x] + f[y]
\end{aligned}
$$

We have

$$\begin{aligned} & B(T) = B(\bar{T}) + f[x] + f[y] \\ \iff \quad & B(\bar{T}) = B(T) - f[x] - f[y] \end{aligned}$$

Now, it's easy to prove Lemma 2 by contradiction.

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

By **Lemma 1**, we can assume that T' has x, y as *sibling leaves*. Let $\bar{T}'$ be T' with the common parent of x, y replaced by a leaf z with $f[z] = f[x] + f[y]$. Then,

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

Note that $\bar{T}'$ is a tree for $\bar{C}$, i.e., we have a *contradiction* since $\bar{T}$ was assumed to be optimal!

Assignment Problem 12.5. (deadline: August 5, 5:30pm)
Suppose a data file contains a sequence of 8-bit characters such that all 256 characters are about as common: the maximum character frequency is less than twice the minimum character frequency. Prove that Huffman coding in this case is no more efficient than using an ordinary 8-bit fixed-length code.

Another example: Scheduling

- We want to optimally schedule jobs on a single machine
- Comes down to defining the **order** in which jobs are processed
- Different objectives possible
- In this case, we want to *minimize the average time jobs spend in the system*

Given: n jobs $j_1, \ldots, j_n$ (arriving at the same time), with service times $t_1, \ldots, t_n$.

Note that the total time is fixed ($\sum t_i$), but not average system time.

Since n is fixed, problem is equivalent to minimizing

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

which is equal to "n times the average system (waiting) time".

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: in optimal solution, jobs are sorted in order of increasing service times.

The 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 a "cheapest" job available for the $(\ell + 1)$-st one, so that other jobs don't have to wait much.

Theorem. Greedy algorithm (choosing at each stage a job with the shortest service time) is optimal.

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

$$\begin{aligned} T(P) &= s_1 + (s_1 + s_2) + (s_1 + s_2 + s_3) + \cdots \\ &= ns_1 + (n-1)s_2 + (n-2)s_3 + \cdots \\ &= \sum_{k=1}^{n} (n - k + 1) \cdot s_k \end{aligned}$$

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

Recall $T(P) = \sum_{k=1}^{n}(n-k+1)\cdot s_k$.

Now we swap positions of a-th and b-th jobs, and obtain a new permutation P' (which is same as P but with p_a and p_b interchanged).

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

(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 $s_a > s_b$.

Conclusion: any schedule P which doesn't arrange jobs in order of increasing service time, is not optimal ($T(P') < T(P)$). Hence, any optimal schedule arranges jobs in *shortest-service-time-first* order. Such schedules have the same cost, and so they are all optimal. Done. □

Extra Assignment Problem 3. (2% added to the overall performance if solved completely)

Deadline: The final exam. (Note: You get extra points only if your solution is completely correct. You can submit the solution several times. If it's not correct, I will point out the problem(s) in your solution and you can try again.)

Consider the following scheduling problem. We have one computer and n jobs $j_1, \ldots, j_n$. Job j_i has a *service time* t_i and a *release time* r_i. We can start processing a job only after its release time. The jobs can be suspended and restarted at a later time! The *completion time* of an job j_i is the time elapsed from the release of the job until it's completely processed by the computer.

Give an algorithm that schedules the jobs so as to minimize the average completion time. Prove that your algorithm minimizes the average completion time, and state the running time of your algorithm.

You can assume that all times t_i and r_i are non-negative integers.