Efficient Sequential Algorithms, Comp309

University of Liverpool

2010–2011

Module Organiser, Igor Potapov

Part 3. String Algorithms

References: T. H. Cormen, C. E. Leiserson, R. L. Rivest **Introduction to Algorithms**, Second Edition. MIT Press (2001). "Longest Common Subsequence" and "Huffman Codes"

Longest Common Subsequence

The Longest Common Subsequence problem is like the pattern matching problem, except that you are allowed to skip characters in the text. Also, the goal is to return just one match, which is as long as possible. Here is an example of two sequences, with a longest common subsequence marked in red.

Sequence 1: A C B C D

Sequence 2: A B C B D

Motivation

Here are some motivating applications from David Eppstein's page
`http://www.ics.uci.edu/~eppstein/161/960229.html`

Molecular biology. DNA sequences (genes) can be represented as sequences of four letters ACGT, corresponding to the four submolecules forming DNA. When biologists find a new sequences, they typically want to know what other sequences it is most similar to. One way of computing how similar two sequences are is to find the length of their longest common subsequence.

File comparison. The Unix program "diff" is used to compare two different versions of the same file, to determine what changes have been made to the file. It works by finding a longest common subsequence of the lines of the two files; any line in the subsequence has not been changed, so what it displays is the remaining set of lines that have changed. In this instance of the problem we should think of each line of a file as being a single complicated character in a string.

Definitions

Given a sequence (i.e. a string of characters) $X = x_1 x_2 \ldots x_m$, another sequence $Z = z_1 z_2 \ldots z_k$ is a subsequence of X if there exists a (strictly increasing) list of indices of X $i_1, i_2, \ldots, i_k$ such that for all $j \in \{1, 2, \ldots, k\}$, we have $x_{i_j} = z_j$.

Examples. The sequence $Z =$ BCDB is a subsequence of $X =$ ABCBDAB with corresponding index list: $2, 3, 5, 7$.

The sequence $Z = 1011$ is a subsequence of $X = 1011011$ with corresponding index list: $1, 2, 3, 4$.

The empty sequence is a subsequence of all sequences.

Given two sequences X and Y, we say that a sequence Z is a common subsequence of X and Y if Z is a subsequence of both X and Y.

Examples. The sequence ABC is a common subsequence of both $X = \text{ACBCD}$ and $Y = \text{ABCBD}$.

The sequence ABCD is a longer (indeed the longest) common subsequence of X and Y (and so is ACBD).

In the longest common subsequence problem we are given two sequences X and Y and wish to find a maximum-length common subsequence (or LCS) of both X and Y.

Characterising a longest common subsequence

Optimal solutions can be obtained from optimal solutions of sub-problems.

Let $X = x_1 x_2 \dots x_m$, and $Y = y_1 y_2 \dots y_n$ be sequences. and let $Z = z_1 z_2 \dots z_k$ be any LCS of X and Y.

1. If $x_m = y_n$, then

   $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.

2. If $x_m \neq y_n$, then

   if $z_k \neq x_m$ then Z is an LCS of X_{m-1} and Y;
   if $z_k \neq y_n$ then Z is an LCS of X and Y_{n-1}.

Characterising a longest common subsequence

Optimal solutions can be obtained from optimal solutions of sub-problems.

Let $X = x_1 x_2 \ldots x_m$, and $Y = y_1 y_2 \ldots y_n$ be sequences. and let $Z = z_1 z_2 \ldots z_k$ be any LCS of X and Y.

1. If $x_m = y_n$, then

   $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.

2. If $x_m \neq y_n$, then

   if $z_k \neq x_m$ then Z is an LCS of X_{m-1} and Y;
   if $z_k \neq y_n$ then Z is an LCS of X and Y_{n-1}.

Characterising a longest common subsequence

Optimal solutions can be obtained from optimal solutions of sub-problems.

Let $X = x_1 x_2 \ldots x_m$, and $Y = y_1 y_2 \ldots y_n$ be sequences. and let $Z = z_1 z_2 \ldots z_k$ be any LCS of X and Y.

1. If $x_m = y_n$, then

   $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.

2. If $x_m \neq y_n$, then

   if $z_k \neq x_m$ then Z is an LCS of X_{m-1} and Y;

   if $z_k \neq y_n$ then Z is an LCS of X and Y_{n-1}.

Characterising a longest common subsequence

Optimal solutions can be obtained from optimal solutions of sub-problems.

Let $X = x_1 x_2 \ldots x_m$, and $Y = y_1 y_2 \ldots y_n$ be sequences. and let $Z = z_1 z_2 \ldots z_k$ be any LCS of X and Y.

1. If $x_m = y_n$, then

   $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.

2. If $x_m \neq y_n$, then

   if $z_k \neq x_m$ then Z is an LCS of X_{m-1} and Y;

   if $z_k \neq y_n$ then Z is an LCS of X and Y_{n-1}.

Characterising a longest common subsequence

Optimal solutions can be obtained from optimal solutions of sub-problems.

Let $X = x_1 x_2 \ldots x_m$, and $Y = y_1 y_2 \ldots y_n$ be sequences. and let $Z = z_1 z_2 \ldots z_k$ be any LCS of X and Y.

1. If $x_m = y_n$, then

   $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.

2. If $x_m \neq y_n$, then

   if $z_k \neq x_m$ then Z is an LCS of X_{m-1} and Y;

   if $z_k \neq y_n$ then Z is an LCS of X and Y_{n-1}.

Proof.

Let $X = x_1 x_2 \dots x_m$, and $Y = y_1 y_2 \dots y_n$ be sequences. and let $Z = z_1 z_2 \dots z_k$ be any LCS of X and Y.

1. If $x_m = y_n$ then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.

If $z_k \neq x_m$ we could append x_m to the end of Z, getting a longer common subsequence of X and Y. So Z_{k-1} is a common subsequence of X_{m-1} and Y_{n-1}.

Suppose for contradiction there is a common subsequence W of X_{m-1} and Y_{m-1} with length greater than $k-1$. Then adding x_m and y_n gives a common subsequence of X and Y with length greater than k.

Proof.

Let $X = x_1 x_2 \ldots x_m$, and $Y = y_1 y_2 \ldots y_n$ be sequences. and let $Z = z_1 z_2 \ldots z_k$ be any LCS of X and Y.

1. If $x_m = y_n$ then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.

If $z_k \neq x_m$ we could append x_m to the end of Z, getting a longer common subsequence of X and Y. So Z_{k-1} is a common subsequence of X_{m-1} and Y_{n-1}.

Suppose for contradiction there is a common subsequence W of X_{m-1} and Y_{m-1} with length greater than $k-1$. Then adding x_m and y_n gives a common subsequence of X and Y with length greater than k.

Proof.

Let $X = x_1 x_2 \dots x_m$, and $Y = y_1 y_2 \dots y_n$ be sequences. and let $Z = z_1 z_2 \dots z_k$ be any LCS of X and Y.

1. If $x_m = y_n$ then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.

If $z_k \neq x_m$ we could append x_m to the end of Z, getting a longer common subsequence of X and Y. So Z_{k-1} is a common subsequence of X_{m-1} and Y_{n-1}.

Suppose for contradiction there is a common subsequence W of X_{m-1} and Y_{m-1} with length greater than $k-1$. Then adding x_m and y_n gives a common subsequence of X and Y with length greater than k.

Proof.

Let $X = x_1 x_2 \ldots x_m$, and $Y = y_1 y_2 \ldots y_n$ be sequences. and let $Z = z_1 z_2 \ldots z_k$ be any LCS of X and Y.

2. If $x_m \neq y_n$ and $z_k \neq x_m$ then Z is an LCS of X_{m-1} and Y.

If $z_k \neq x_m$ then Z is a common subsequence of X_{m-1} and Y. If there were a common sub sequence W of X_{m-1} and Y with length greater than k then W would also be a common subsequence of X and Y, contradicting the assumption that Z is an LCS of X and Y.

3. If $x_m \neq y_n$ and $z_k \neq y_n$ then Z is an LCS of X and Y_{n-1}.
symmetric proof.

Proof.

Let $X = x_1 x_2 \ldots x_m$, and $Y = y_1 y_2 \ldots y_n$ be sequences. and let $Z = z_1 z_2 \ldots z_k$ be any LCS of X and Y.

2. If $x_m \neq y_n$ and $z_k \neq x_m$ then Z is an LCS of X_{m-1} and Y.

If $z_k \neq x_m$ then Z is a common subsequence of X_{m-1} and Y. If there were a common sub sequence W of X_{m-1} and Y with length greater than k then W would also be a common subsequence of X and Y, contradicting the assumption that Z is an LCS of X and Y.

3. If $x_m \neq y_n$ and $z_k \neq y_n$ then Z is an LCS of X and Y_{n-1}.
symmetric proof.

Proof.

Let $X = x_1 x_2 \ldots x_m$, and $Y = y_1 y_2 \ldots y_n$ be sequences. and let $Z = z_1 z_2 \ldots z_k$ be any LCS of X and Y.

2. If $x_m \neq y_n$ and $z_k \neq x_m$ then Z is an LCS of X_{m-1} and Y.

If $z_k \neq x_m$ then Z is a common subsequence of X_{m-1} and Y. If there were a common sub sequence W of X_{m-1} and Y with length greater than k then W would also be a common subsequence of X and Y, contradicting the assumption that Z is an LCS of X and Y.

3. If $x_m \neq y_n$ and $z_k \neq y_n$ then Z is an LCS of X and Y_{n-1}.

symmetric proof.

Proof.

Let $X = x_1 x_2 \ldots x_m$, and $Y = y_1 y_2 \ldots y_n$ be sequences. and let $Z = z_1 z_2 \ldots z_k$ be any LCS of X and Y.

2. If $x_m \neq y_n$ and $z_k \neq x_m$ then Z is an LCS of X_{m-1} and Y.

If $z_k \neq x_m$ then Z is a common subsequence of X_{m-1} and Y. If there were a common sub sequence W of X_{m-1} and Y with length greater than k then W would also be a common subsequence of X and Y, contradicting the assumption that Z is an LCS of X and Y.

3. If $x_m \neq y_n$ and $z_k \neq y_n$ then Z is an LCS of X and Y_{n-1}.
symmetric proof.

A recursive solution to subproblems

A recursive solution to the LCS problem also has the overlapping-subproblems property. The analysis so far implies that to find the LCS of X and Y we either have $x_m = y_n$, in which case the problem is reduced to compute the LCS of X_{m-1} and Y_{n-1} or else we find an LCS in X_{m-1} and Y or X and Y_{n-1}. Each of these subproblems has LCS(X_{m-1}, Y_{n-1}) as common sub-problem.

The recursive definition of the optimal cost is readily completed. If $lcs[i, j]$ is the longest common subsequence of X_i and Y_j, then

$$lcs[i,j] = \begin{cases} 0 & i = 0 \ \vee \ j = 0 \\ lcs[i-1, j-1] + 1 & i, j > 0 \ \wedge \ x_i = y_j \\ \max\{lcs[i, j-1], lcs[i-1, j]\} & i, j > 0 \ \wedge \ x_i \neq y_j \end{cases}$$

A recursive solution to subproblems

A recursive solution to the LCS problem also has the overlapping-subproblems property. The analysis so far implies that to find the LCS of X and Y we either have $x_m = y_n$, in which case the problem is reduced to compute the LCS of X_{m-1} and Y_{n-1} or else we find an LCS in X_{m-1} and Y or X and Y_{n-1}. Each of these subproblems has LCS(X_{m-1}, Y_{n-1}) as common sub-problem.

The recursive definition of the optimal cost is readily completed. If $lcs[i, j]$ is the longest common subsequence of X_i and Y_j, then

$$lcs[i,j] = \begin{cases} 0 & i = 0 \ \vee \ j = 0 \\ lcs[i-1, j-1] + 1 & i, j > 0 \ \wedge \ x_i = y_j \\ \max\{lcs[i, j-1], lcs[i-1, j]\} & i, j > 0 \ \wedge \ x_i \neq y_j \end{cases}$$

Computing the length of an LCS

One can easily write an exponential time recursive algorithm to compute *lcs*. However, since there are only $\Theta(nm)$ distinct subproblems, we can also use dynamic programming to compute the solutions bottom up.

The following procedure takes two sequences $X = x_1 x_2 \ldots x_m$ and $Y = y_1 y_2 \ldots y_n$ and stores the output in a two dimensional array *lcs*. The first row is filled from left to right, then the second row, and so on. On completion $lcs[m, n]$ contains the size of an LCS between X and Y.

The table $b[1..m, 1..n]$ will be used later to simplify the construction of a longest common subsequence from *lcs*.

The overall running time of the procedure is $O(nm)$ since each table entry takes $O(1)$ time to compute.

Computing the length of an LCS

One can easily write an exponential time recursive algorithm to compute *lcs*. However, since there are only $\Theta(nm)$ distinct subproblems, we can also use dynamic programming to compute the solutions bottom up.

The following procedure takes two sequences $X = x_1 x_2 \ldots x_m$ and $Y = y_1 y_2 \ldots y_n$ and stores the output in a two dimensional array *lcs*. The first row is filled from left to right, then the second row, and so on. On completion $lcs[m, n]$ contains the size of an LCS between X and Y.

The table $b[1..m, 1..n]$ will be used later to simplify the construction of a longest common subsequence from *lcs*.

The overall running time of the procedure is $O(nm)$ since each table entry takes $O(1)$ time to compute.

Computing the length of an LCS

One can easily write an exponential time recursive algorithm to compute *lcs*. However, since there are only $\Theta(nm)$ distinct subproblems, we can also use dynamic programming to compute the solutions bottom up.

The following procedure takes two sequences $X = x_1 x_2 \ldots x_m$ and $Y = y_1 y_2 \ldots y_n$ and stores the output in a two dimensional array *lcs*. The first row is filled from left to right, then the second row, and so on. On completion $lcs[m, n]$ contains the size of an LCS between X and Y.

The table $b[1..m, 1..n]$ will be used later to simplify the construction of a longest common subsequence from *lcs*.

The overall running time of the procedure is $O(nm)$ since each table entry takes $O(1)$ time to compute.

Computing the length of an LCS

One can easily write an exponential time recursive algorithm to compute *lcs*. However, since there are only $\Theta(nm)$ distinct subproblems, we can also use dynamic programming to compute the solutions bottom up.

The following procedure takes two sequences $X = x_1 x_2 \ldots x_m$ and $Y = y_1 y_2 \ldots y_n$ and stores the output in a two dimensional array *lcs*. The first row is filled from left to right, then the second row, and so on. On completion $lcs[m, n]$ contains the size of an LCS between X and Y.

The table $b[1..m, 1..n]$ will be used later to simplify the construction of a longest common subsequence from *lcs*.

The overall running time of the procedure is $O(nm)$ since each table entry takes $O(1)$ time to compute.

```
LCS (X, Y)
        m ← length(X)
        n ← length(Y)
        for i ← 1 to m lcs[i, 0] ← 0
        for j ← 0 to n lcs[0, j] ← 0
        for i ← 1 to m
                for j ← 1 to n
                        if x_i = y_j
                                lcs[i, j] ← lcs[i − 1, j − 1] + 1
                                b[i, j] ← "↖"
                        else if lcs[i − 1, j] ≥ lcs[i, j − 1]
                                lcs[i, j] ← lcs[i − 1, j]
                                b[i, j] ← "↑"
                        else
                                lcs[i, j] ← lcs[i, j − 1]
                                b[i, j] ← "←"
        return lcs and b
```

Constructing an LCS

The b table can be used to construct a LCS for the given instance. We simply begin at $b[m, n]$ and trace through the table "following directions": the "$\nwarrow$" symbol implies that $x_i = y_j$ is an element of the LCS.

```
PRINT-LCS (b, X, i, j)
    if i = 0 ∨ j = 0 return
    if b[i, j] = "↖"
        PRINT-LCS (b, X, i − 1, j − 1)
        print x_i
    else if b[i, j] = "↑"
        PRINT-LCS (b, X, i − 1, j)
    else
        PRINT-LCS (b, X, i, j − 1)
```

The procedure takes $O(n + m)$ time, since at least one of i and j is decremented in each stage of the recursion.

Example

Let $X = 10010101$ and $Y = 010110110$.

i	j	0	1	2	3	4	5	6	7	8	9
		y_i	0	1	0	1	1	0	1	1	0
0	x_i										
		0	0	0	0	0	0	0	0	0	0
1	1										
		0									
2	0										
		0									
3	0										
		0									
4	1										
		0									
5	0										
		0									
6	1										
		0									
7	0										
		0									
8	1										
		0									

$i = 1, j = 1$: $x_i \neq y_j$. Also $lcs[i-1,j] \geq lcs[i,j-1]$ so we do

$lcs[i,j] \leftarrow lcs[i-1,j]$ and $b[i,j]$ becomes $\uparrow$

i	j	0	**1**	2	3	4	5	6	7	8	9
		y_i	0	1	0	1	1	0	1	1	0
0	x_i										
		0	0	0	0	0	0	0	0	0	0
1	1		$\uparrow$								
		0	0								
2	0										
		0									
3	0										
		0									
4	1										
		0									
5	0										
		0									
6	1										
		0									
7	0										
		0									
8	1										
		0									

$i = 1, j = 2$: $x_i = y_j$. So we do $lcs[i,j] \leftarrow lcs[i-1, j-1] + 1$ and $b[i,j]$ becomes $\nwarrow$

	j	0	1	**2**	3	4	5	6	7	8	9
i		y_i	0	1	0	1	1	0	1	1	0
0	x_i										
		0	0	0	0	0	0	0	0	0	0
1	1		↑	$\nwarrow$							
		0	0	1							
2	0										
		0									
3	0										
		0									
4	1										
		0									
5	0										
		0									
6	1										
		0									
7	0										
		0									
8	1										
		0									

$i = 1, j = 3$: $x_i \neq y_j$. Also $lcs[i-1, j] < lcs[i, j-1]$ so we do $lcs[i, j] \leftarrow lcs[i, j-1]$ and $b[i, j]$ becomes $\leftarrow$

i	j	0	1	2	**3**	4	5	6	7	8	9
		y_i	0	1	0	1	1	0	1	1	0
0	x_i										
		0	0	0	0	0	0	0	0	0	0
1	1		↑	↖							
		0	0	1	← 1						
2	0										
		0									
3	0										
		0									
4	1										
		0									
5	0										
		0									
6	1										
		0									
7	0										
		0									
8	1										
		0									

$i = 1, j = 4$: $x_i = y_j$. So we do $lcs[i,j] \leftarrow lcs[i-1, j-1] + 1$ and $b[i,j]$ becomes $\nwarrow$

i	j	0	1	2	3	**4**	5	6	7	8	9
		y_i	0	1	0	1	1	0	1	1	0
0	x_i										
		0	0	0	0	0	0	0	0	0	0
1	1		$\uparrow$	$\nwarrow$							
		0	0	1	$\leftarrow$ 1	1					
2	0										
		0									
3	0										
		0									
4	1										
		0									
5	0										
		0									
6	1										
		0									
7	0										
		0									
8	1										
		0									

The process should be clear now. Here is the final table.

i	j	0	1	2	3	4	5	6	7	8	9
		y_i	0	1	0	1	1	0	1	1	0
0	x_i										
		0	0	0	0	0	0	0	0	0	0
1	1		↑	↖		↖	↖		↖	↖	
		0	0	1	← 1	1	1	← 1	1	1	← 1
2	0		↖	↑	↖			↖			↖
		0	1	1	2	← 2	← 2	2	← 2	← 2	2
3	0		↖	↑	↖	↑	↑	↖			↖
		0	1	1	2	2	2	3	← 3	← 3	3
4	1		↑	↖	↑	↖	↖	↑	↖	↖	
		0	1	2	2	3	3	3	4	4	← 4
5	0		↖	↑	↖	↑	↑	↖	↑	↑	↖
		0	1	2	3	3	3	4	4	4	5
6	1		↑	↖	↑	↖	↖	↑	↖	↖	↑
		0	1	2	3	4	4	4	5	5	5
7	0		↖	↑	↖	↑	↑	↖	↑	↑	↖
		0	1	2	3	4	4	5	5	5	6
8	1		↑	↖	↑	↖	↖	↑	↖	↖	↑
		0	1	2	3	4	5	5	6	6	6

And here is the common subsequence: print x_i when we leave row i with ↖

i	j	0	1	2	3	4	5	6	7	8	9
		y_i	0	1	0	1	1	0	1	1	0
0	x_i	0	0	0	0	0	0	0	0	0	0
1	1	0	0	1 ←	1	1	1 ←	1	1	1	1 ←
2	0	0	1	1	2 ←	2 ←	2	2	2 ←	2 ←	2
3	0	0	1	1	2	2	2	3 ←	3 ←	3	3
4	1	0	1	2	2	3	3	3	4	4	4 ←
5	0	0	1	2	3	3	3	4	4	4	5
6	1	0	1	2	3	4	4	4	5	5	5
7	0	0	1	2	3	4	4	5	5	5	6
8	1	0	1	2	3	4	5	5	6	6	6

Improvements

- It is possible to eliminate the b table altogether. Each entry $lcs[i, j]$ depends on only three other values: $lcs[i-1, j-1]$, $lcs[i-1, j]$, and $lcs[i, j-1]$. Given $lcs[i, j]$ we can determine in $O(1)$ time which of the three values was used to compute it. Thus, we can reconstruct an LCS in $O(n+m)$ time using a procedure similar to PRINT-LCS (exercise!). Although we save $\Theta(mn)$ space by this method, the auxiliary space requirement for computing an LCS does not asymptotically decrease, since we still need $\Theta(mn)$ space for the table lcs.
- If we care only about the length of the longest common subsequence and we don't want to construct it, we can reduce the asymptotic space requirements by keeping only two rows of lcs, the current one and the previous one.

Improvements

- It is possible to eliminate the b table altogether. Each entry $lcs[i, j]$ depends on only three other values: $lcs[i-1, j-1]$, $lcs[i-1, j]$, and $lcs[i, j-1]$. Given $lcs[i, j]$ we can determine in $O(1)$ time which of the three values was used to compute it. Thus, we can reconstruct an LCS in $O(n+m)$ time using a procedure similar to PRINT-LCS (exercise!). Although we save $\Theta(mn)$ space by this method, the auxiliary space requirement for computing an LCS does not asymptotically decrease, since we still need $\Theta(mn)$ space for the table lcs.
- If we care only about the length of the longest common subsequence and we don't want to construct it, we can reduce the asymptotic space requirements by keeping only two rows of lcs, the current one and the previous one.

Currently David Eppstein, Zvi Galil, Raffaele Giancarlo and Giuseppe Italiano hold the record for the fastest LCS algorithm:

> *"Actually, if you look at the matrix above, you can tell that it has a lot of structure – the numbers in the matrix form large blocks in which the value is constant, with only a small number of* corners *at which the value changes. It turns out that one can take advantage of these corners to speed up the computation. The current (theoretically) fastest algorithm for longest common subsequences (due to myself and co-authors) runs in time* $O(n \log s + c \log \log \min(c, mn/c))$ *where c is the number of these corners, and s is the number of characters appearing in the two strings."*

Text Compression

Suppose we have a text (a sequence of characters) that we wish to store on a computer. Information is stored as a binary sequence. The simplest option is to use the extended ASCII code, like Java. Encode each character using two bytes. Clearly, this method uses 200,000 bytes to store a 100,000-character text. We can do much better in practice.

Lossless Compression

Lossless compression algorithms are algorithms that allow the exact original data to be reconstructed from the compressed data.

A lossless compression algorithm cannot make every file shorter.

If some file gets shorter, some other file must get longer.

To see this, use the pigeonhole principle.

Lossless Compression

Lossless compression algorithms are algorithms that allow the exact original data to be reconstructed from the compressed data.

A lossless compression algorithm cannot make every file shorter.

If some file gets shorter, some other file must get longer.

To see this, use the pigeonhole principle.

Lossless Compression

Lossless compression algorithms are algorithms that allow the exact original data to be reconstructed from the compressed data.

A lossless compression algorithm cannot make every file shorter.

If some file gets shorter, some other file must get longer.

To see this, use the pigeonhole principle.

Let $\ell(f)$ denote the length of a file f and let $c(f)$ denote the length of the compressed version of f.

Suppose for contradiction that $\forall f, c(f) \leq \ell(f)$ and $\exists f, c(f) < \ell(f)$.

Let L be the smallest integer such that there exists an f with $L = \ell(f) > c(f)$. Pick some such f and let $C = c(f) < L$.

Now, how many different files f have $c(f) = C$?

At least $2^C + 1$, including 2^C files f with $\ell(f) = C$ and at least one file with $\ell(f) = L$.

By the pigeonhole principle, at least two of these have the same compressed text, so the compression is not lossless.

Let $\ell(f)$ denote the length of a file f and let $c(f)$ denote the length of the compressed version of f.

Suppose for contradiction that $\forall f, c(f) \leq \ell(f)$ and $\exists f, c(f) < \ell(f)$.

Let L be the smallest integer such that there exists an f with $L = \ell(f) > c(f)$. Pick some such f and let $C = c(f) < L$.

Now, how many different files f have $c(f) = C$?

At least $2^C + 1$, including 2^C files f with $\ell(f) = C$ and at least one file with $\ell(f) = L$.

By the pigeonhole principle, at least two of these have the same compressed text, so the compression is not lossless.

Let $\ell(f)$ denote the length of a file f and let $c(f)$ denote the length of the compressed version of f.

Suppose for contradiction that $\forall f, c(f) \leq \ell(f)$ and $\exists f, c(f) < \ell(f)$.

Let L be the smallest integer such that there exists an f with $L = \ell(f) > c(f)$. Pick some such f and let $C = c(f) < L$.

Now, how many different files f have $c(f) = C$?

At least $2^C + 1$, including 2^C files f with $\ell(f) = C$ and at least one file with $\ell(f) = L$.

By the pigeonhole principle, at least two of these have the same compressed text, so the compression is not lossless.

Let $\ell(f)$ denote the length of a file f and let $c(f)$ denote the length of the compressed version of f.

Suppose for contradiction that $\forall f, c(f) \leq \ell(f)$ and $\exists f, c(f) < \ell(f)$.

Let L be the smallest integer such that there exists an f with $L = \ell(f) > c(f)$. Pick some such f and let $C = c(f) < L$.

Now, how many different files f have $c(f) = C$?

At least $2^C + 1$, including 2^C files f with $\ell(f) = C$ and at least one file with $\ell(f) = L$.

By the pigeonhole principle, at least two of these have the same compressed text, so the compression is not lossless.

Let $\ell(f)$ denote the length of a file f and let $c(f)$ denote the length of the compressed version of f.

Suppose for contradiction that $\forall f, c(f) \leq \ell(f)$ and $\exists f, c(f) < \ell(f)$.

Let L be the smallest integer such that there exists an f with $L = \ell(f) > c(f)$. Pick some such f and let $C = c(f) < L$.

Now, how many different files f have $c(f) = C$?

At least $2^C + 1$, including 2^C files f with $\ell(f) = C$ and at least one file with $\ell(f) = L$.

By the pigeonhole principle, at least two of these have the same compressed text, so the compression is not lossless.

So choosing the compression algorithm, is essentially choosing which files will get shorter.

Text Compression through Huffman Coding

In a code, each character in the text is represented by a unique binary string.

Fixed-length codes (for example, the extended ASCII code in which each character is stored in two bytes) can be inefficient. We can do better with a variable length code. Give frequent characters short codewords.

Huffman codes represent a very effective technique for compressing data; they usually produce savings between 20% and 90%.

Text Compression through Huffman Coding

In a code, each character in the text is represented by a unique binary string.

Fixed-length codes (for example, the extended ASCII code in which each character is stored in two bytes) can be inefficient. We can do better with a variable length code. Give frequent characters short codewords.

Huffman codes represent a very effective technique for compressing data; they usually produce savings between 20% and 90%.

Text Compression through Huffman Coding

In a code, each character in the text is represented by a unique binary string.

Fixed-length codes (for example, the extended ASCII code in which each character is stored in two bytes) can be inefficient. We can do better with a variable length code. Give frequent characters short codewords.

Huffman codes represent a very effective technique for compressing data; they usually produce savings between 20% and 90%.

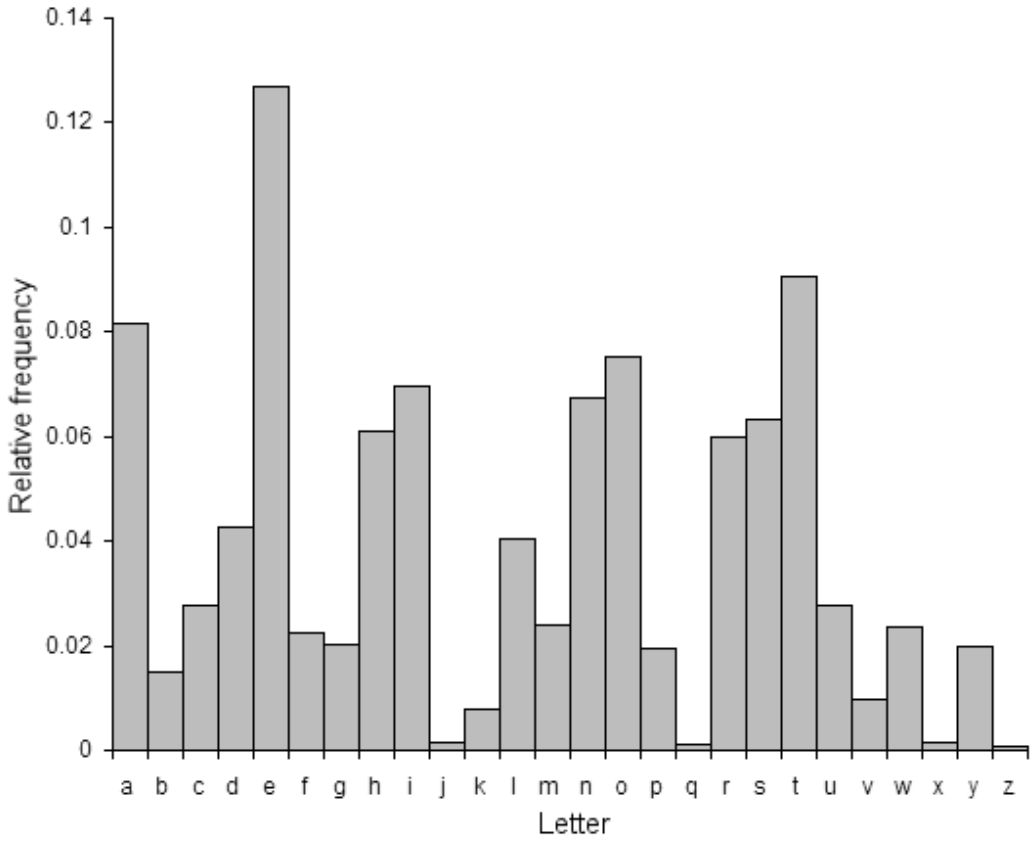
Text Compression through Huffman Coding

In a code, each character in the text is represented by a unique binary string.

Fixed-length codes (for example, the extended ASCII code in which each character is stored in two bytes) can be inefficient. We can do better with a variable length code. Give frequent characters short codewords.

Huffman codes represent a very effective technique for compressing data; they usually produce savings between 20% and 90%.

Which characters are frequent?



English character frequencies, from Wikipedia.

Prefix Codes

A prefix code is a code in which no codword is a prefix of some other codeword.

symbol	codeword
a	0
b	10
c	110

encoding is easy. Given the source text T, simply concatenate the codewords for the characters in T:

abababa becomes 0100100100

(the colour is to help you see what is going on — it is not part of the code).

Prefix Codes

A prefix code is a code in which no codword is a prefix of some other codeword.

symbol	codeword
a	0
b	10
c	110

encoding is easy. Given the source text T, simply concatenate the codewords for the characters in T:

abababa becomes 0100100100

(the colour is to help you see what is going on — it is not part of the code).

symbol	codeword
a	0
b	10
c	110

decoding is easy too.

the code 0100100100 corresponds to *a* followed by text for 100100100

which is *ab* followed by text for 0100100

which is *aba* followed by text for 100100

...

which is *abababa*.

Exercise

Write a decoder for this code.

symbol	codeword
a	0
b	10
c	110

What is its time complexity?

It has been shown that the optimal data compression achievable by any code can always be achieved with a prefix code. Therefore we will stick to prefix codes.

Data Structures

We will use a priority queue, which you learned about in Comp202. A *priority queue* is a data structure for maintaining a set Q of elements, each with an associated value (or *key*). A priority queue supports the following operations:

INSERT(Q,x) inserts the element x into Q.

MIN(Q) returns the element of Q with minimal key.

EXTRACT-MIN(Q) removes and returns the element of Q with minimal key.

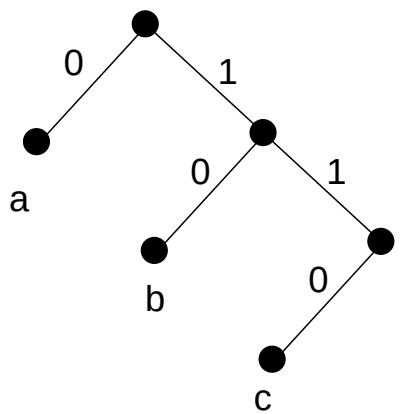
You learned an implementation in which each operation takes $O(\log n)$ time, where n is the maximum size of Q.

Data Structures

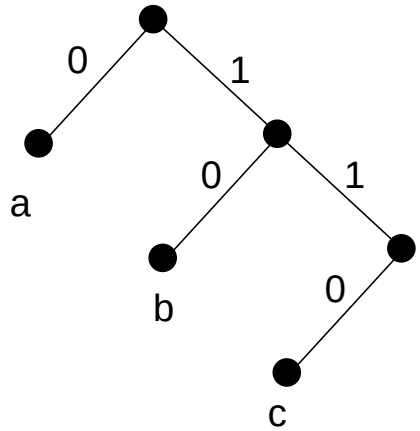
We represent a codeword as a binary tree. We store the characters at the leaves. The binary codeword for a character is interpreted as a path from the root to the character, where 0 means “go left” and 1 means “go right”. So the code

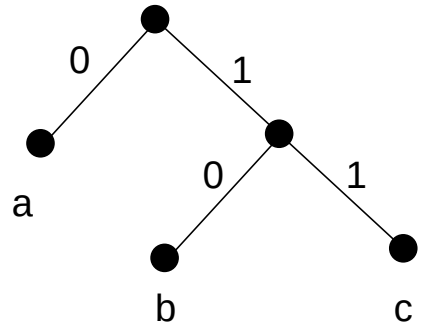
symbol	codeword
a	0
b	10
c	110

is depicted like this.



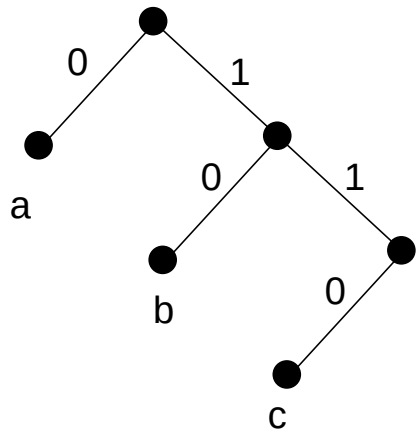
An optimal code is full, meaning that every non-leaf node has two children.

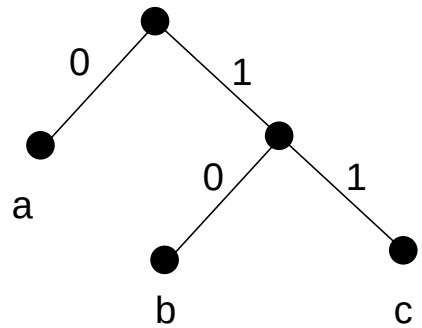




A full binary tree with n leaves has $n - 1$ internal nodes, so So, if Σ is the text alphabet, then the tree for an optimal prefix code has exactly $|\Sigma|$ leaves, one for each letter in Σ, and $|\Sigma| - 1$ internal nodes.

An optimal code is full, meaning that every non-leaf node has two children.





A full binary tree with n leaves has $n - 1$ internal nodes, so So, if Σ is the text alphabet, then the tree for an optimal prefix code has exactly $|\Sigma|$ leaves, one for each letter in Σ, and $|\Sigma| - 1$ internal nodes.

Cost of a tree

Suppose that we are given relative character frequencies for the characters in the alphabet where $f[c]$ denotes the relative frequency of character c.

Given a tree T corresponding to a prefix code, the cost of T is defined to be the average code-word length, which is denoted $B(T)$ and is computed as follows.

Let $d_T(c)$ denote the depth of c's leaf in T (note that $d_T(c)$ is also the length of the codeword for c).

$$B(T) = \sum_{c \in \Sigma} f[c] d_T(c).$$

Cost of a tree

Suppose that we are given relative character frequencies for the characters in the alphabet where $f[c]$ denotes the relative frequency of character c.

Given a tree T corresponding to a prefix code, the cost of T is defined to be the average code-word length, which is denoted $B(T)$ and is computed as follows.

Let $d_T(c)$ denote the depth of c's leaf in T (note that $d_T(c)$ is also the length of the codeword for c).

$$B(T) = \sum_{c \in \Sigma} f[c] d_T(c).$$

Cost of a tree

Suppose that we are given relative character frequencies for the characters in the alphabet where $f[c]$ denotes the relative frequency of character c.

Given a tree T corresponding to a prefix code, the cost of T is defined to be the average code-word length, which is denoted $B(T)$ and is computed as follows.

Let $d_T(c)$ denote the depth of c's leaf in T (note that $d_T(c)$ is also the length of the codeword for c).

$$B(T) = \sum_{c \in \Sigma} f[c] d_T(c).$$

Cost of a tree

Suppose that we are given relative character frequencies for the characters in the alphabet where $f[c]$ denotes the relative frequency of character c.

Given a tree T corresponding to a prefix code, the cost of T is defined to be the average code-word length, which is denoted $B(T)$ and is computed as follows.

Let $d_T(c)$ denote the depth of c's leaf in T (note that $d_T(c)$ is also the length of the codeword for c).

$$B(T) = \sum_{c \in \Sigma} f[c] d_T(c).$$

Constructing a Huffman code

Huffman invented a greedy algorithm that constructs an optimal prefix code, called a Huffman code.

The algorithm builds the tree T corresponding to an optimal code in a bottom-up manner. It begins with a set of $|\Sigma|$ leaves and performs a sequence of $|\Sigma| - 1$ "merging" operations to create the final tree.

For each $c \in \Sigma$, $f[c]$ denotes the (given) frequency of c.

The nodes of T are stored in a priority queue Q. Initially, Q contains an object corresponding to each leaf c and the object has key $f[c]$.

The two least frequent leaves are then selected and merged together. The leaves are merged by adding a new root which points to both of them. The key of the new object is the sum of the frequencies of the two objects that were merged.

The generic node z of T is an object containing two pointers "left" and "right" to the left and right child of z in T.

Constructing a Huffman code

Huffman invented a greedy algorithm that constructs an optimal prefix code, called a Huffman code.

The algorithm builds the tree T corresponding to an optimal code in a bottom-up manner. It begins with a set of $|\Sigma|$ leaves and performs a sequence of $|\Sigma| - 1$ "merging" operations to create the final tree.

For each $c \in \Sigma$, $f[c]$ denotes the (given) frequency of c.

The nodes of T are stored in a priority queue Q. Initially, Q contains an object corresponding to each leaf c and the object has key $f[c]$.

The two least frequent leaves are then selected and merged together. The leaves are merged by adding a new root which points to both of them. The key of the new object is the sum of the frequencies of the two objects that were merged.

The generic node z of T is an object containing two pointers "left" and "right" to the left and right child of z in T.

Constructing a Huffman code

Huffman invented a greedy algorithm that constructs an optimal prefix code, called a Huffman code.

The algorithm builds the tree T corresponding to an optimal code in a bottom-up manner. It begins with a set of $|\Sigma|$ leaves and performs a sequence of $|\Sigma| - 1$ "merging" operations to create the final tree.

For each $c \in \Sigma$, $f[c]$ denotes the (given) frequency of c.

The nodes of T are stored in a priority queue Q. Initially, Q contains an object corresponding to each leaf c and the object has key $f[c]$.

The two least frequent leaves are then selected and merged together. The leaves are merged by adding a new root which points to both of them. The key of the new object is the sum of the frequencies of the two objects that were merged.

The generic node z of T is an object containing two pointers "left" and "right" to the left and right child of z in T.

Constructing a Huffman code

Huffman invented a greedy algorithm that constructs an optimal prefix code, called a Huffman code.

The algorithm builds the tree T corresponding to an optimal code in a bottom-up manner. It begins with a set of $|\Sigma|$ leaves and performs a sequence of $|\Sigma| - 1$ "merging" operations to create the final tree.

For each $c \in \Sigma$, $f[c]$ denotes the (given) frequency of c.

The nodes of T are stored in a priority queue Q. Initially, Q contains an object corresponding to each leaf c and the object has key $f[c]$.

The two least frequent leaves are then selected and merged together. The leaves are merged by adding a new root which points to both of them. The key of the new object is the sum of the frequencies of the two objects that were merged.

The generic node z of T is an object containing two pointers "left" and "right" to the left and right child of z in T.

Constructing a Huffman code

Huffman invented a greedy algorithm that constructs an optimal prefix code, called a Huffman code.

The algorithm builds the tree T corresponding to an optimal code in a bottom-up manner. It begins with a set of $|\Sigma|$ leaves and performs a sequence of $|\Sigma| - 1$ "merging" operations to create the final tree.

For each $c \in \Sigma$, $f[c]$ denotes the (given) frequency of c.

The nodes of T are stored in a priority queue Q. Initially, Q contains an object corresponding to each leaf c and the object has key $f[c]$.

The two least frequent leaves are then selected and merged together. The leaves are merged by adding a new root which points to both of them. The key of the new object is the sum of the frequencies of the two objects that were merged.

The generic node z of T is an object containing two pointers "left" and "right" to the left and right child of z in T.

Constructing a Huffman code

Huffman invented a greedy algorithm that constructs an optimal prefix code, called a Huffman code.

The algorithm builds the tree T corresponding to an optimal code in a bottom-up manner. It begins with a set of $|\Sigma|$ leaves and performs a sequence of $|\Sigma| - 1$ "merging" operations to create the final tree.

For each $c \in \Sigma$, $f[c]$ denotes the (given) frequency of c.

The nodes of T are stored in a priority queue Q. Initially, Q contains an object corresponding to each leaf c and the object has key $f[c]$.

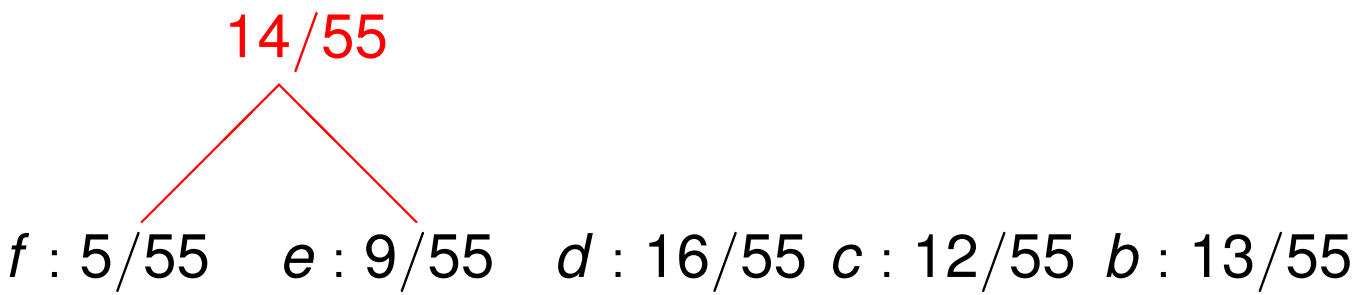
The two least frequent leaves are then selected and merged together. The leaves are merged by adding a new root which points to both of them. The key of the new object is the sum of the frequencies of the two objects that were merged.

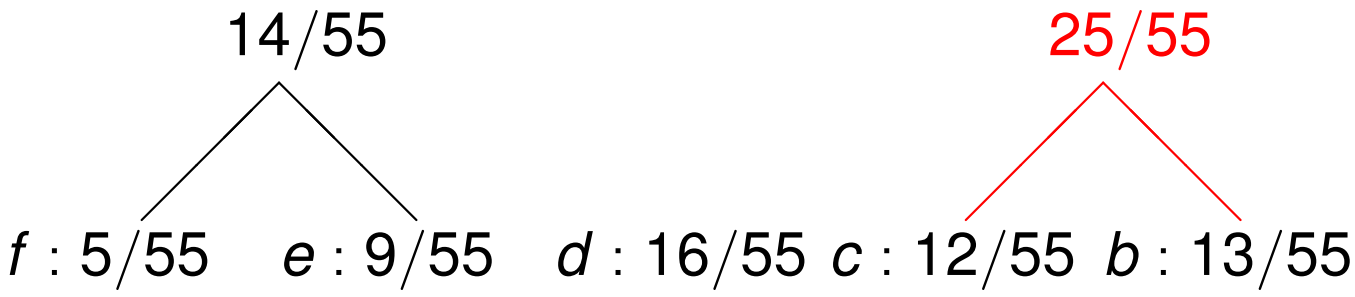
The generic node z of T is an object containing two pointers "left" and "right" to the left and right child of z in T.

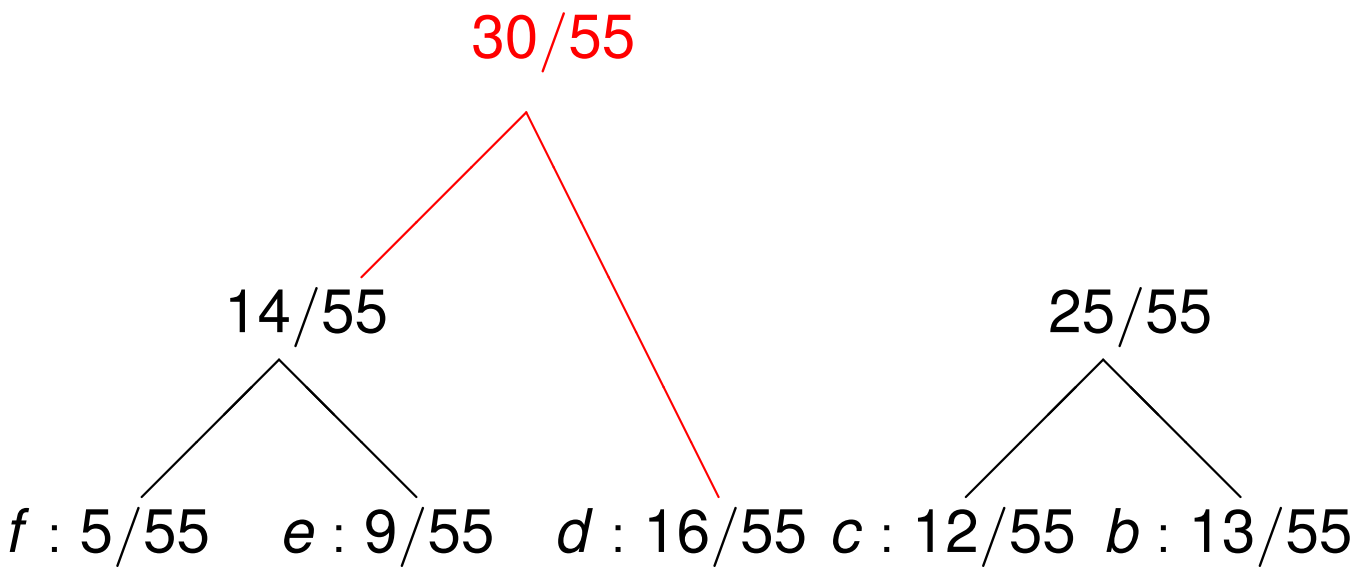
```
HUFFMAN (Σ)
    n ← |Σ|
    Q ← Σ
    for i ← 1 to n − 1
        z ← ALLOCATE-NODE()
        left[z] ← EXTRACT-MIN(Q)
        x ← left[z]
        right[z] ← EXTRACT-MIN(Q)
        y ← right[z]
        f[z] ← f[x] + f[y]
        INSERT(Q, z)
    return EXTRACT-MIN(Q)
```

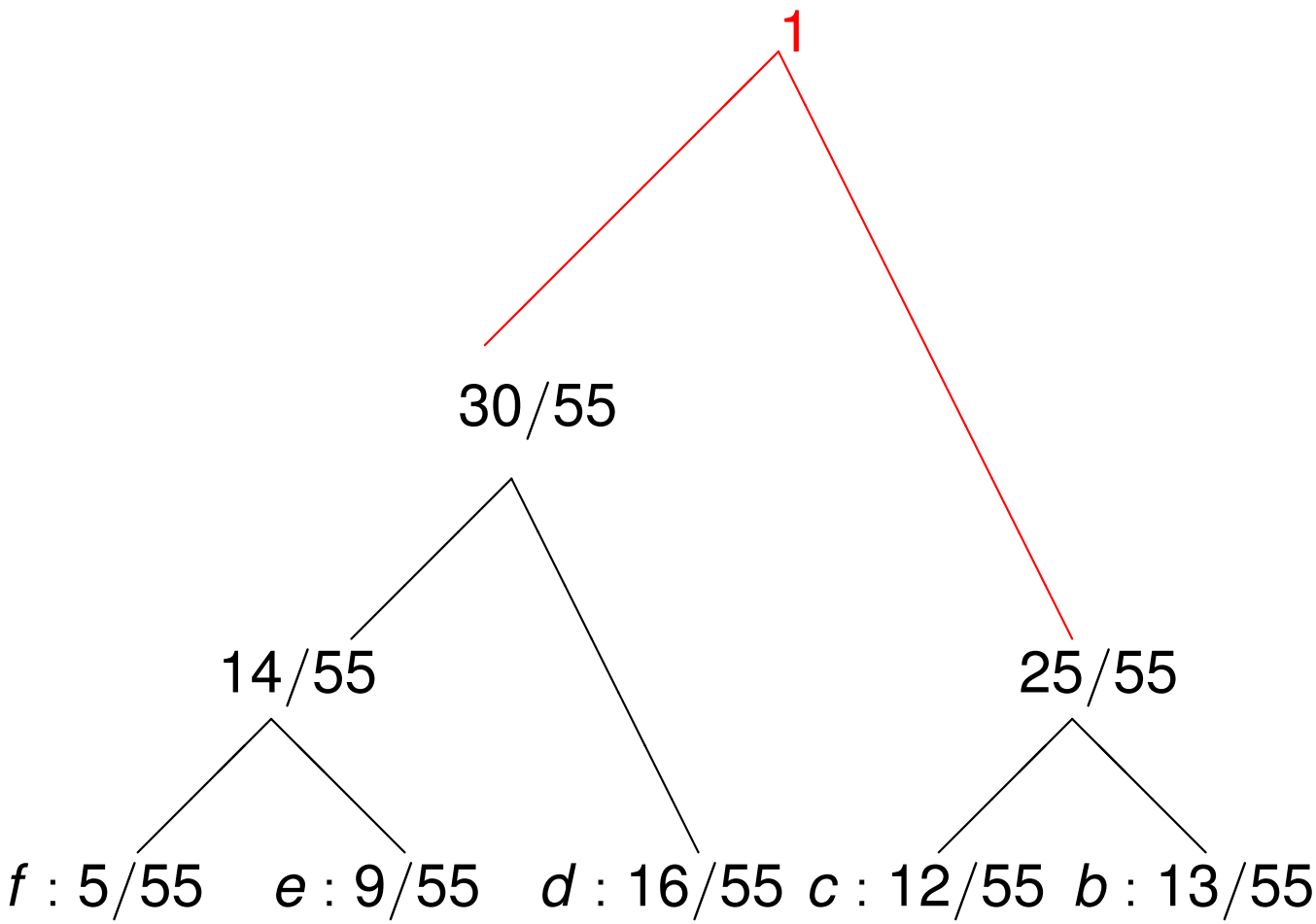
Example

$f : 5/55 \quad e : 9/55 \quad d : 16/55 \; c : 12/55 \; b : 13/55$


14/55
f : 5/55
e : 9/55
d : 16/55
c : 12/55
b : 13/55


14/55
f : 5/55
e : 9/55
d : 16/55
25/55
c : 12/55
b : 13/55


30/55
14/55
25/55
f : 5/55
e : 9/55
d : 16/55
c : 12/55
b : 13/55



So codeword for f is 000

Analysis

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

Each priority queue operation takes $O(\log n)$ so total is $O(n \log n)$.

Correctness

To prove the greedy algorithm HUFFMAN is correct, we must show that the problem exhibits these two properties:

Greedy-Choice property: For every instance, there is an optimal solution consistent with the first greedy choice. Let x and y be two characters in Σ having the lowest frequencies. Then there exists an optimal prefix code in which the codewords for x and y have the same length and differ only in the last bit.

Recursive property: For every instance Σ of the problem there is a smaller instance Σ' (namely the one in which x and y are replaced with z) such that, using any optimal solution to Σ', one obtains a best-possible solution for Σ amongst all solutions which are consistent with the first greedy choice.

Correctness

To prove the greedy algorithm HUFFMAN is correct, we must show that the problem exhibits these two properties:

Greedy-Choice property: For every instance, there is an optimal solution consistent with the first greedy choice. Let x and y be two characters in Σ having the lowest frequencies. Then there exists an optimal prefix code in which the codewords for x and y have the same length and differ only in the last bit.

Recursive property: For every instance Σ of the problem there is a smaller instance Σ' (namely the one in which x and y are replaced with z) such that, using any optimal solution to Σ', one obtains a best-possible solution for Σ amongst all solutions which are consistent with the first greedy choice.

Correctness

To prove the greedy algorithm HUFFMAN is correct, we must show that the problem exhibits these two properties:

Greedy-Choice property: For every instance, there is an optimal solution consistent with the first greedy choice. Let x and y be two characters in Σ having the lowest frequencies. Then there exists an optimal prefix code in which the codewords for x and y have the same length and differ only in the last bit.

Recursive property: For every instance Σ of the problem there is a smaller instance Σ' (namely the one in which x and y are replaced with z) such that, using any optimal solution to Σ', one obtains a best-possible solution for Σ amongst all solutions which are consistent with the first greedy choice.

Recursive property: For every instance Σ of the problem there is a smaller instance Σ' (namely the one in which x and y are replaced with z) such that, using any optimal solution to Σ', one obtains a best-possible solution for Σ amongst all solutions which are consistent with the first greedy choice.

Let T be a full binary tree representing an optimal prefix code for Σ. Consider any two characters x and y that appear as sibling leaves in T and let z be their parent. Construct Σ' from Σ by replacing x and y by a new character z with frequency $f[z] = f[x] + f[y]$. The induced tree $T' = T - \{x, y\}$ is an optimal prefix code for the alphabet $\Sigma' = \Sigma - \{x, y\} \cup \{z\}$

We must show that any tree for Σ' that is as good as T' can be extended by adding x and y below z to get a tree as good as T

Recursive property: For every instance Σ of the problem there is a smaller instance Σ' (namely the one in which x and y are replaced with z) such that, using any optimal solution to Σ', one obtains a best-possible solution for Σ amongst all solutions which are consistent with the first greedy choice.

Let T be a full binary tree representing an optimal prefix code for Σ. Consider any two characters x and y that appear as sibling leaves in T and let z be their parent. Construct Σ' from Σ by replacing x and y by a new character z with frequency $f[z] = f[x] + f[y]$. The induced tree $T' = T - \{x, y\}$ is an optimal prefix code for the alphabet $\Sigma' = \Sigma - \{x, y\} \cup \{z\}$

We must show that any tree for Σ' that is as good as T' can be extended by adding x and y below z to get a tree as good as T

Greedy-Choice

Let x and y be two characters in Σ having the lowest frequencies. Then there exists an optimal prefix code in which the codewords for x and y have the same length and differ only in the last bit.

Proof idea: Take any optimal T and modify it to make another optimal T'' in which x and y are sibling leaves of maximum depth.

Greedy-Choice

Let x and y be two characters in Σ having the lowest frequencies. Then there exists an optimal prefix code in which the codewords for x and y have the same length and differ only in the last bit.

Proof idea: Take any optimal T and modify it to make another optimal T'' in which x and y are sibling leaves of maximum depth.

Details

Take any optimal T and modify it to make another optimal T'' in which x and y are sibling leaves of maximum depth.

Let b and c be sibling leaves of maximum depth with $f[b] \leq f[c], f[x] \leq f[y]$

Since x and y have lowest frequencies, $f[x] \leq f[b]$ and $f[y] \leq f[c]$

Produce T' from T by exchanging x and b. Produce T'' from T' by exchanging y and c.

Details

Take any optimal T and modify it to make another optimal T'' in which x and y are sibling leaves of maximum depth.

Let b and c be sibling leaves of maximum depth with $f[b] \leq f[c], f[x] \leq f[y]$

Since x and y have lowest frequencies, $f[x] \leq f[b]$ and $f[y] \leq f[c]$

Produce T' from T by exchanging x and b. Produce T'' from T' by exchanging y and c.

Details

Take any optimal T and modify it to make another optimal T'' in which x and y are sibling leaves of maximum depth.

Let b and c be sibling leaves of maximum depth with $f[b] \leq f[c], f[x] \leq f[y]$

Since x and y have lowest frequencies, $f[x] \leq f[b]$ and $f[y] \leq f[c]$

Produce T' from T by exchanging x and b. Produce T'' from T' by exchanging y and c.

Details

Take any optimal T and modify it to make another optimal T'' in which x and y are sibling leaves of maximum depth.

Let b and c be sibling leaves of maximum depth with $f[b] \leq f[c], f[x] \leq f[y]$

Since x and y have lowest frequencies, $f[x] \leq f[b]$ and $f[y] \leq f[c]$

Produce T' from T by exchanging x and b. Produce T'' from T' by exchanging y and c.

Details

Take any optimal T and modify it to make another optimal T'' in which x and y are sibling leaves of maximum depth.

Let b and c be sibling leaves of maximum depth with $f[b] \leq f[c], f[x] \leq f[y]$

Since x and y have lowest frequencies, $f[x] \leq f[b]$ and $f[y] \leq f[c]$

Produce T' from T by exchanging x and b. Produce T'' from T' by exchanging y and c.

$$
\begin{aligned}
B(T) - B(T') &= \sum_{c \in \Sigma} f[c] d_T(c) - \sum_{c \in \Sigma} f[c] d_{T'}(c) \\
&= f[x] d_T(x) + f[b] d_T(b) - f[x] d_{T'}(x) - f[b] d_{T'}(b) \\
&= f[x] d_T(x) + f[b] d_T(b) - f[x] d_T(b) - f[b] d_T(x) \\
&= (f[b] - f[x])(d_T(b) - d_T(x)) \\
&\geq 0 \text{ since both factors are non-negative}
\end{aligned}
$$

Similarly, $B(T') \geq B(T'')$

So $B(T'') \leq B(T)$ and since T is optimal, so is T''.

$$
\begin{aligned}
B(T) - B(T') &= \sum_{c \in \Sigma} f[c] d_T(c) - \sum_{c \in \Sigma} f[c] d_{T'}(c) \\
&= f[x] d_T(x) + f[b] d_T(b) - f[x] d_{T'}(x) - f[b] d_{T'}(b) \\
&= f[x] d_T(x) + f[b] d_T(b) - f[x] d_T(b) - f[b] d_T(x) \\
&= (f[b] - f[x])(d_T(b) - d_T(x)) \\
&\geq 0 \text{ since both factors are non-negative}
\end{aligned}
$$

Similarly, $B(T') \geq B(T'')$

So $B(T'') \leq B(T)$ and since T is optimal, so is T''.

$$
\begin{aligned}
B(T) - B(T') &= \sum_{c \in \Sigma} f[c] d_T(c) - \sum_{c \in \Sigma} f[c] d_{T'}(c) \\
&= f[x] d_T(x) + f[b] d_T(b) - f[x] d_{T'}(x) - f[b] d_{T'}(b) \\
&= f[x] d_T(x) + f[b] d_T(b) - f[x] d_T(b) - f[b] d_T(x) \\
&= (f[b] - f[x])(d_T(b) - d_T(x)) \\
&\geq 0 \text{ since both factors are non-negative}
\end{aligned}
$$

Similarly, $B(T') \geq B(T'')$

So $B(T'') \leq B(T)$ and since T is optimal, so is T''.

Recursive property

Let T be a full binary tree representing an optimal prefix code. Consider any two characters x and y that appear as sibling leaves in T and let z be their parent. Let's first check that, considering z as a character with frequency $f[z] = f[x] + f[y]$, the induced tree $T' = T - \{x, y\}$ is an optimal prefix code for the alphabet $\Sigma' = \Sigma - \{x, y\} \cup \{z\}$

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

so $B(T) = B(T') + f[x] + f[y]$

Suppose for contradiction that T' is not optimal so there is a T'' for $\Sigma - \{x, y\} \cup \{z\}$ with $B(T'') < B(T')$

Then from T'' we can obtain a tree for Σ with cost $B(T'') + f[x] + f[y] < B(T)$ contradicting optimality of T.

Recursive property

Let T be a full binary tree representing an optimal prefix code. Consider any two characters x and y that appear as sibling leaves in T and let z be their parent. Let's first check that, considering z as a character with frequency $f[z] = f[x] + f[y]$, the induced tree $T' = T - \{x, y\}$ is an optimal prefix code for the alphabet $\Sigma' = \Sigma - \{x, y\} \cup \{z\}$

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

so $B(T) = B(T') + f[x] + f[y]$

Suppose for contradiction that T' is not optimal so there is a T'' for $\Sigma - \{x, y\} \cup \{z\}$ with $B(T'') < B(T')$

Then from T'' we can obtain a tree for Σ with cost $B(T'') + f[x] + f[y] < B(T)$ contradicting optimality of T.

Recursive property

Let T be a full binary tree representing an optimal prefix code. Consider any two characters x and y that appear as sibling leaves in T and let z be their parent. Let's first check that, considering z as a character with frequency $f[z] = f[x] + f[y]$, the induced tree $T' = T - \{x, y\}$ is an optimal prefix code for the alphabet $\Sigma' = \Sigma - \{x, y\} \cup \{z\}$

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

so $B(T) = B(T') + f[x] + f[y]$

Suppose for contradiction that T' is not optimal so there is a T'' for $\Sigma - \{x, y\} \cup \{z\}$ with $B(T'') < B(T')$

Then from T'' we can obtain a tree for Σ with cost $B(T'') + f[x] + f[y] < B(T)$ contradicting optimality of T.

Recursive property

Let T be a full binary tree representing an optimal prefix code. Consider any two characters x and y that appear as sibling leaves in T and let z be their parent. Let's first check that, considering z as a character with frequency $f[z] = f[x] + f[y]$, the induced tree $T' = T - \{x, y\}$ is an optimal prefix code for the alphabet $\Sigma' = \Sigma - \{x, y\} \cup \{z\}$

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

so $B(T) = B(T') + f[x] + f[y]$

Suppose for contradiction that T' is not optimal so there is a T'' for $\Sigma - \{x, y\} \cup \{z\}$ with $B(T'') < B(T')$

Then from T'' we can obtain a tree for Σ with cost $B(T'') + f[x] + f[y] < B(T)$ contradicting optimality of T.

Recursive property

Let T be a full binary tree representing an optimal prefix code. Consider any two characters x and y that appear as sibling leaves in T and let z be their parent. Let's first check that, considering z as a character with frequency $f[z] = f[x] + f[y]$, the induced tree $T' = T - \{x, y\}$ is an optimal prefix code for the alphabet $\Sigma' = \Sigma - \{x, y\} \cup \{z\}$

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

so $B(T) = B(T') + f[x] + f[y]$

Suppose for contradiction that T' is not optimal so there is a T'' for $\Sigma - \{x, y\} \cup \{z\}$ with $B(T'') < B(T')$

Then from T'' we can obtain a tree for Σ with cost $B(T'') + f[x] + f[y] < B(T)$ contradicting optimality of T.

We must show that any tree for Σ' that is as good as T' can be extended by adding x and y below z to get a tree as good as T.

Any such tree would have cost $B(T')$ and addding x and y gives $B(T') + f[x] + f[y] = B(T)$.