SS 2022

# Algorithms

# Chapter 5.2

# Huffman encoding

Text: Martin Dietzfelbinger  Presentation: Arindam Biswas

July 2022

# Explanation

"Greedy" algorithms may be used in
**construction tasks** for **finding an optimal structure**.

They find a solution structure step by step, without ever undoing any step.
In each step, they make a choice that looks good in the present situation.
They never construct more intermediate solutions than absolutely necessary.

Previous example: Dijkstra's algorithm.
Coming examples: Algorithms by Jarník/Prim and Kruskal for minimum spanning trees.

## 5.2.1 Huffman Codes

**Given:** **Alphabet $A$** with $2 \leq |A| < \infty$ and "probabilities" or "relative frequencies" $p(a) \in [0,1]$ for each $a \in A$.
Hence: $\sum_{a \in A} p(a) = 1$.

*Example* with $|A| = 10$:

| | A | B | C | D | E | F | G | H | I | K |
|---|---|---|---|---|---|---|---|---|---|---|
| $p(a)$ | 0.15 | 0.08 | 0.07 | 0.10 | 0.21 | 0.08 | 0.07 | 0.09 | 0.06 | 0.09 |

**Origin** of the probabilites:
(1) Statistical frequency of letters in natural language (English, German, . . . ) *or*
(2) empirical relative frequencies in a given text $w = a_1 \ldots a_n$:
Fraction of letter $a$ in $w$ is $(p(a) \cdot 100)$ percent.

**Note:** In the book absolute frequencies instead of our percentages are used. There is no conceptual difference. You can use the book notation if you wish.

We seek: **"good" binary prefix code** for $(A, p)$. — *Example*:

| A | B | C | D | E | F | G | H | I | K |
|---|---|---|---|---|---|---|---|---|---|
| 1100 | 0110 | 000 | 111 | 10 | 0011 | 010 | 0010 | 0111 | 1101 |

**Definition 5.2.1 prefix code:**

Each $a \in A$ is associated with a "codeword" $c(a) \in \{0,1\}^*$ (set of all binary strings), with property **prefix free**:

> For $a, b \in A$, $a \neq b$, codeword $c(a)$ is not a prefix of codeword $c(b)$.

Coding of words (strings):

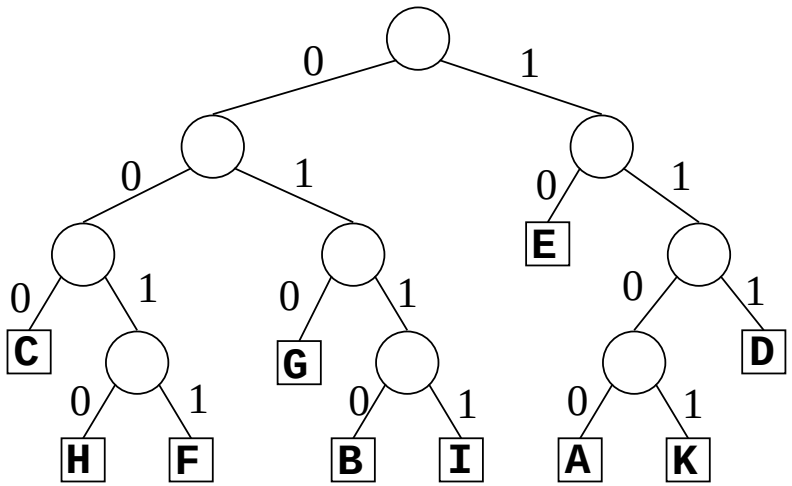
$c(a_1 \dots a_n) = c(a_1) \cdots c(a_n) \in \{0,1\}^*$.

For **coding** use the table directly.

*Example*: $c(\texttt{C A C H E}) = 0001100000001010$.

Leave out the gaps.

Compact representation of the code as binary tree:




Leaves are marked with letters; edges are marked with bits; edge markings on the path from the root to the leaf gives the codeword (left: $0$, right: $1$).

**Decoding:** Start at root, walk path in tree, directed by the bits in the codeword, until leaf is reached. Repeat with remainder of the word, until nothing is left.
Prefix property $\Rightarrow$ we do not need gaps between the codes for the single letters.

*Example*: $0110110011101010$ gives "`BADGE`".

**First idea:** All code words $c(a)$ have same length;
best possible: length of $\lceil \log_2 |A| \rceil$ bits.

$\Rightarrow c(a_1 \dots a_n)$ has length $\lceil \log_2 |A| \rceil \cdot n$.

(Examples: $52$ uppercase and lowercase letters plus punctuation marks plus space:
$\log 64 = 6$ bits per codeword. Latin-1-Code: $8$ bits per codeword.)

**Second idea:** Can we save something by coding frequent letters with shorter codewords than rare letters?

A first attempt at **data compression**
(store files in less storage space, save time when transmitting data)!

Here: **"lossless compression"** – full information is preserved in the codeword.

**Opposite:** e.g. MP3: loses information when compressing.

| | A | B | C | D | E | F | G | H | I | K |
|---|---|---|---|---|---|---|---|---|---|---|
| $p(a)$ | 0.15 | 0.08 | 0.07 | 0.10 | 0.21 | 0.08 | 0.07 | 0.09 | 0.06 | 0.09 |
| $c_1$ | 0000 | 0001 | 0010 | 0011 | 0100 | 0101 | 0110 | 0111 | 1000 | 1001 |
| $c_2$ | 1100 | 0110 | 000 | 111 | 10 | 0011 | 010 | 0010 | 0111 | 1101 |

We code a file $T$ with $100000$ letters from $A$, where the relative frequency of $a \in A$ is given by $p(a)$.

With $c_1$ (fixed code word length): $400000$ bits.

With $c_2$ (variable code word length):
$(4 \cdot (0.15 + 0.08 + 0.08 + 0.09 + 0.06 + 0.09) + 3 \cdot (0.07 + 0.10 + 0.07)$
$+ 2 \cdot 0.21) \cdot 100000 \ = 334000$ bits.

For long files or when transmission is expensive it pays to count the letters and to determine the relative frequencies $p(a)$, and to find a good code with variable code word length.

## Definition 5.2.2

A **coding tree** for $A$ is a binary tree $T$ in which

- the edge out of an inner node to its left resp. right child is marked by $0$ resp. $1$;
- each letter $a \in A$ is assigned to one and only one leaf of $T$, and each leaf is assigned a letter.

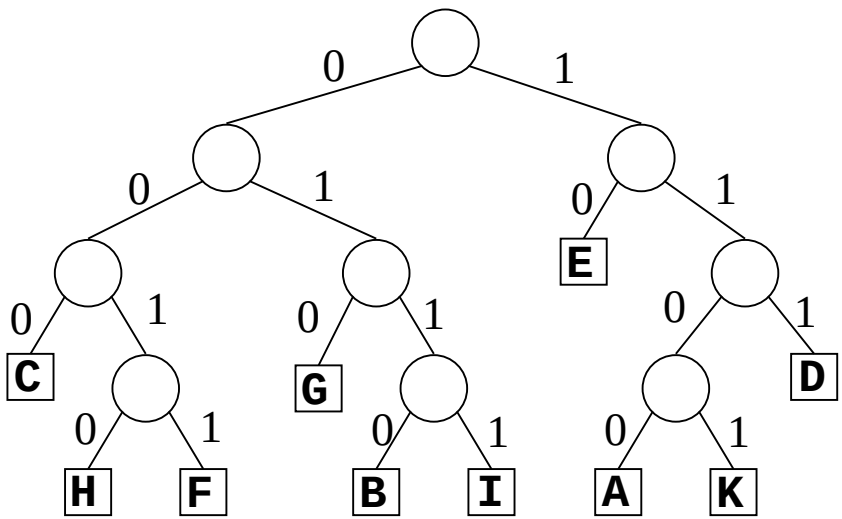
The code $c_T(a)$ for $a$ is the inscription on the edges of the path from the root to the leaf with inscription $a$.

The **cost** of $T$ under $p$ are defined as

$$\boldsymbol{B(T,p)} = \sum\nolimits_{a \in A} p(a) \cdot d_T(a),$$

where $d_T(a) = |c_T(a)|$ is the depth of the $a$-leaf in $T$.

*Example*: If $T$ is our example tree and $p$ is the sample distribution from above, then $B(T,p) = 3.34$.




Easy to see: $B(T,p) = |c_T(a_1 \ldots a_n)|/n$,
if the relative frequency of $a$ in $w = a_1 \ldots a_n$ is given by $p(a)$,

**or** $B(T,p) =$ the **expected number of bits** per letter,
if $p(a)$ is the **probability of letter** $a \in A$.

## Definition 5.2.3

A coding tree $T$ for $A$ is **optimal** (or **redundancy minimal** for $p$) if $B(T,p) \leq B(T',p)$ for all coding trees $T'$ for $A$.

**Task:**
Given a probability distribution $p\colon A \to [0,1]$, find an optimal coding tree $T$.

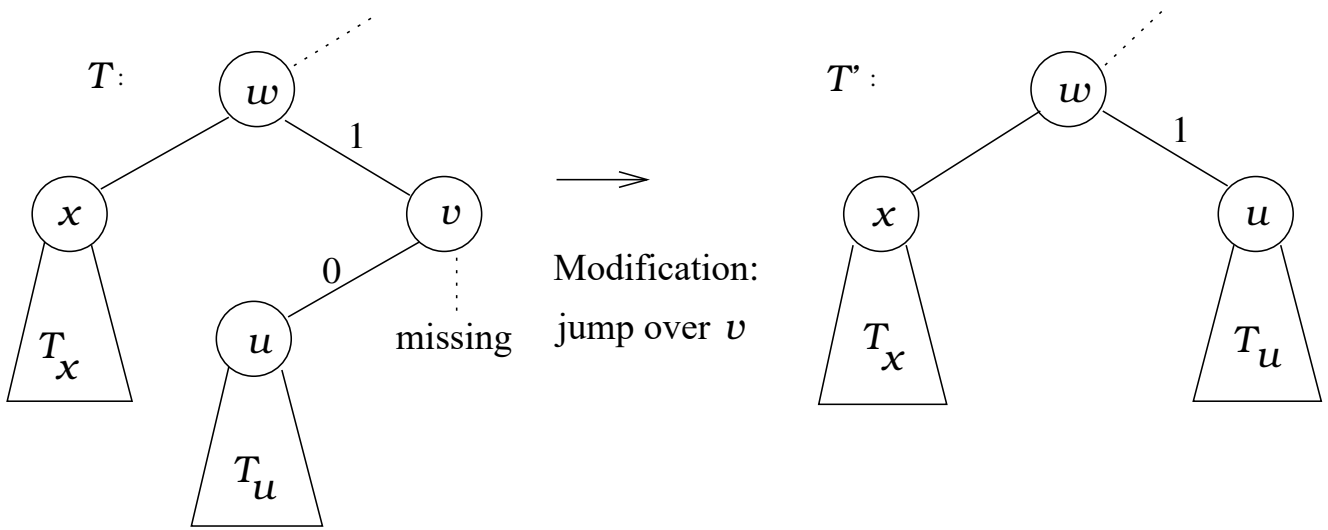
Is there always an optimal tree?

(An alphabet $A$ admits infinitely many coding trees!)

Yes, see next slide ...

**Lemma 5.2.4** If $T$ is a coding tree and $p$ is a probability distribution for $A$, then there is a coding tree $T'$ with $B(T', p) \leq B(T, p)$, such that in $T'$ every inner node has two children.

*Proof idea*:




Result: $T'$ for $A$ with the same marked leaves as $T$, and $B(T', p) \leq B(T, p)$.

**Consequence:** When looking for optimal trees, we can restrict our attention to trees in which every inner node has two children.

For fixed alphabet $A$ there are only finitely many $A$-coding trees $T$ in which every inner node has two children. One of these trees must have minimal cost for input $(A, p)$. Hence there are always optimal coding trees.

Optimal trees are not unique (excepting in the boring case $|A| = 1$).

**Task:** Given $(A, p)$, where $|A| \geq 2$, find an optimal tree.

**Method: "greedy"**.

## Lemma 5.2.5

Let $a$ and $a'$ be two letters with $p(a) \leq p(a') \leq p(b)$ for all $b \in A - \{a, a'\}$

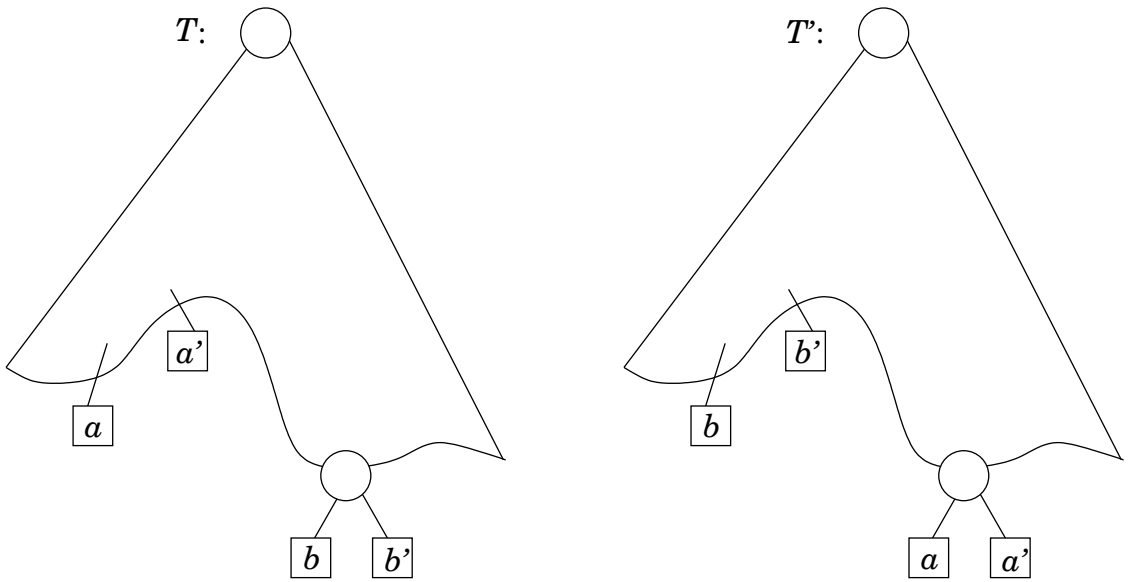
($a$, $a'$ are two **"rarest"** or **"least probable"** letters. They need not be unique.)

Then there is an optimal tree in which the leaves with $a$ and $a'$ are "siblings", i.e. children of the same node.

*Proof*:

Start with an arbitrary optimal tree $T$.

Without loss of generality (Le. 5.2.4): All inner nodes have two children.




Choose two sibling leaves at maximal depth $d = d(T)$, with letters $b$ and $b'$, say, where $p(b) \leq p(b')$.

Exchange $a$ with $b$ and $a'$ with $b'$ to obtain tree $T'$.

Then $B(T, p) \geq B(T', p)$, because $p(a) \leq p(b)$ and $p(a') \leq p(b')$.

Hence $T'$ is optimal for $(A, p)$ as well. □

We have done the first step in direction of a greedy approach!

The algorithm starts with

"Choose two rarest/lightest letters and make them siblings."

Then it is sure that this choice can be extended to an optimal solution.

We do the rest by recursion (that's the concept) respectively by **iteration** (in an actual implementation).

Algorithm is due to D. A. Huffman (1925–1999), an American computer scientist.

Nice story:

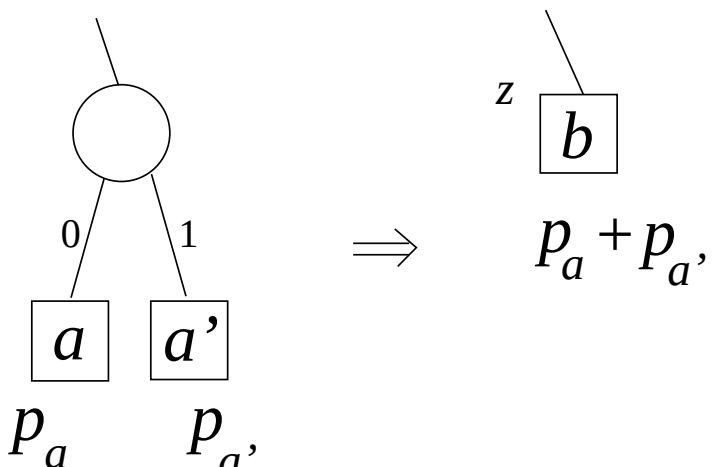
`http://www.huffmancoding.com/my-uncle/scientific-american`

**Huffman's algorithm** (recursive version):

We build a tree **"bottom-up"**.

If $|A| = 2$: We make the leaves for the two letters children of the root. It is clear that this is optimal.

Otherwise, for $|A| > 2$, two "rarest" letters $a$, $a'$ from $A$ are placed into sibling nodes:

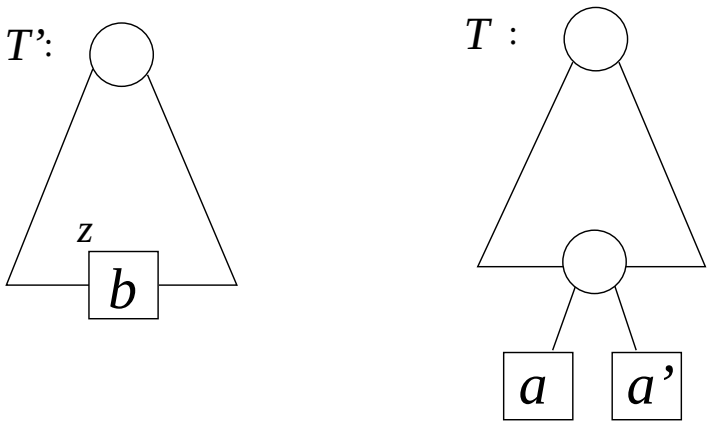


The root of this mini tree is regarded as a new leaf node $z$ with "artificial" letter $b$, with $p(b) = p(a) + p(a')$.

New alphabet: $A' := (A - \{a, a'\}) \cup \{b\}$; new probability distribution:

$$p'(d) := \begin{cases} p(d) & \text{if } d \neq b \\ p(a) + p(a') & \text{if } d = b \end{cases}$$

Now we call the algorithm **recursively** to build a tree $T'$ for $A'$ and $p'$.




In $T'$ we replace leaf node $z$ (with letter $b$) with the mini tree for $a$ and $a'$.

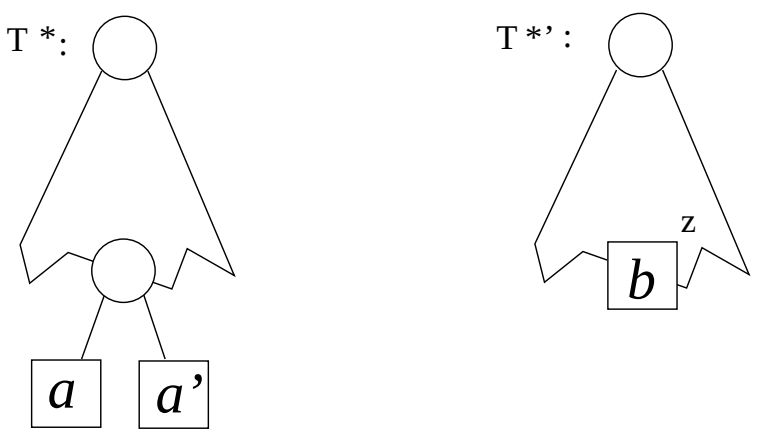
Result: A **coding tree** $T$ for $A$, with $B(T,p) = B(T',p') + p(a) + p(a')$. (Check!)

**Lemma 5.2.6** $T$ is an **optimal** tree for $(A, p)$.

*Proof*: By induction over recursive calls.

By Lemma 5.2.5. there is an **optimal** tree $T^*$ for $(A, p)$ in which the $a$ and $a'$ leaves are siblings.

From $T^*$ we obtain $T^{*\prime}$ by replacing the mini tree for $a$ and $a'$ by the new leaf node $z$ with artificial letter $b$.
Then $T^{*\prime}$ is a coding tree for $A'$.




We have: $B(T^*, p) = B(T^{*\prime}, p') + p(a) + p(a')$.

By the induction hypothesis $T'$ is optimal for $(A', p')$, hence $B(T', p') \leq B(T^{*\prime}, p')$. This implies the following.

$$\begin{aligned} B(T,p) &= B(T',p') + p(a) + p(a') \\ &\leq B(T^{*\prime},p') + p(a) + p(a') \\ &= B(T^*,p). \end{aligned}$$

Since $T^*$ is an optimal tree for $(A, p)$:
$B(T, p) = B(T^*, p)$, and $T$ is optimal as well. □

One could program the algorithm as a recursive procedure. It is also often programmed with a priority queue as a data structure.

Much more efficient: **Iterative** implementation with some tricks.

- sort at the beginning s.t. $p(a_1) \leq \cdots \leq p(a_n)$ (takes time $O(n \log n)$).
- number the leaves $1, \ldots, n$ (letters from $A$), $n+1, \ldots, 2n-1$ for inner nodes (artificial letters).
- special representation of coding tree (only indices, no pointers).
- Observation: The newly created probabilities $p(b)$ are (weakly) increasing.

Data structures are three arrays:

$\texttt{p}[1..2n-1]$ (probabilties for $a \in A$ and for artificial letters),
$\texttt{pred}[1..2n-2]$ (predecessor node, $2n-1$ is the root)
$\texttt{mark}[1..2n-2]$ (is edge to node $i$ marked 0 or 1?)

| $a_i$ | A | B | C | D | E | F | G | H | I | K |
|---|---|---|---|---|---|---|---|---|---|---|
| $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| $p_i$ | 0.15 | 0.08 | 0.07 | 0.10 | 0.21 | 0.08 | 0.07 | 0.09 | 0.06 | 0.09 |
| pred | | | | | | | | | | |
| mark | | | | | | | | | | |

Sort and re-number.

(In general: running time $O(n \log n)$.)

| $a_i$ | I | G | C | F | B | K | H | D | A | E |
|---|---|---|---|---|---|---|---|---|---|---|
| $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| $p_i$ | 0.06 | 0.07 | 0.07 | 0.08 | 0.08 | 0.09 | 0.09 | 0.10 | 0.15 | 0.21 |
| pred | | | | | | | | | | |
| mark | | | | | | | | | | |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| $i$ | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| $p_i$ | | | | | | | | | |
| pred | | | | | | | | | — |
| mark | | | | | | | | | — |

| $a_i$ | I | G | C | F | B | K | H | D | A | E |
|---|---|---|---|---|---|---|---|---|---|---|
| $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| $p_i$ | 0.06 | 0.07 | 0.07 | 0.08 | 0.08 | 0.09 | 0.09 | 0.10 | 0.15 | 0.21 |
| pred | **11** | **11** | | | | | | | | |
| mark | **0** | **1** | | | | | | | | |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| $i$ | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| $p_i$ | **0.13** | | | | | | | | |
| pred | | | | | | | | | — |
| mark | | | | | | | | | — |

| $a_i$ | I | G | C | F | B | K | H | D | A | E |
|---|---|---|---|---|---|---|---|---|---|---|
| $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| $p_i$ | 0.06 | 0.07 | 0.07 | 0.08 | 0.08 | 0.09 | 0.09 | 0.10 | 0.15 | 0.21 |
| pred | 11 | 11 | 12 | 12 | 13 | 13 | **14** | **14** | | |
| mark | 0 | 1 | 0 | 1 | 0 | 1 | **0** | **1** | | |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| $i$ | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| $p_i$ | 0.13 | 0.15 | 0.17 | **0.19** | | | | | |
| pred | | | | | | | | | — |
| mark | | | | | | | | | — |

| $a_i$ | I | G | C | F | B | K | H | D | A | E |
|---|---|---|---|---|---|---|---|---|---|---|
| $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| $p_i$ | 0.06 | 0.07 | 0.07 | 0.08 | 0.08 | 0.09 | 0.09 | 0.10 | 0.15 | 0.21 |
| pred | 11 | 11 | 12 | 12 | 13 | 13 | 14 | 14 | **15** | |
| mark | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | **1** | |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| $i$ | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| $p_i$ | 0.13 | 0.15 | 0.17 | 0.19 | **0.28** | | | | |
| pred | **15** | | | | | | | | — |
| mark | **0** | | | | | | | | — |

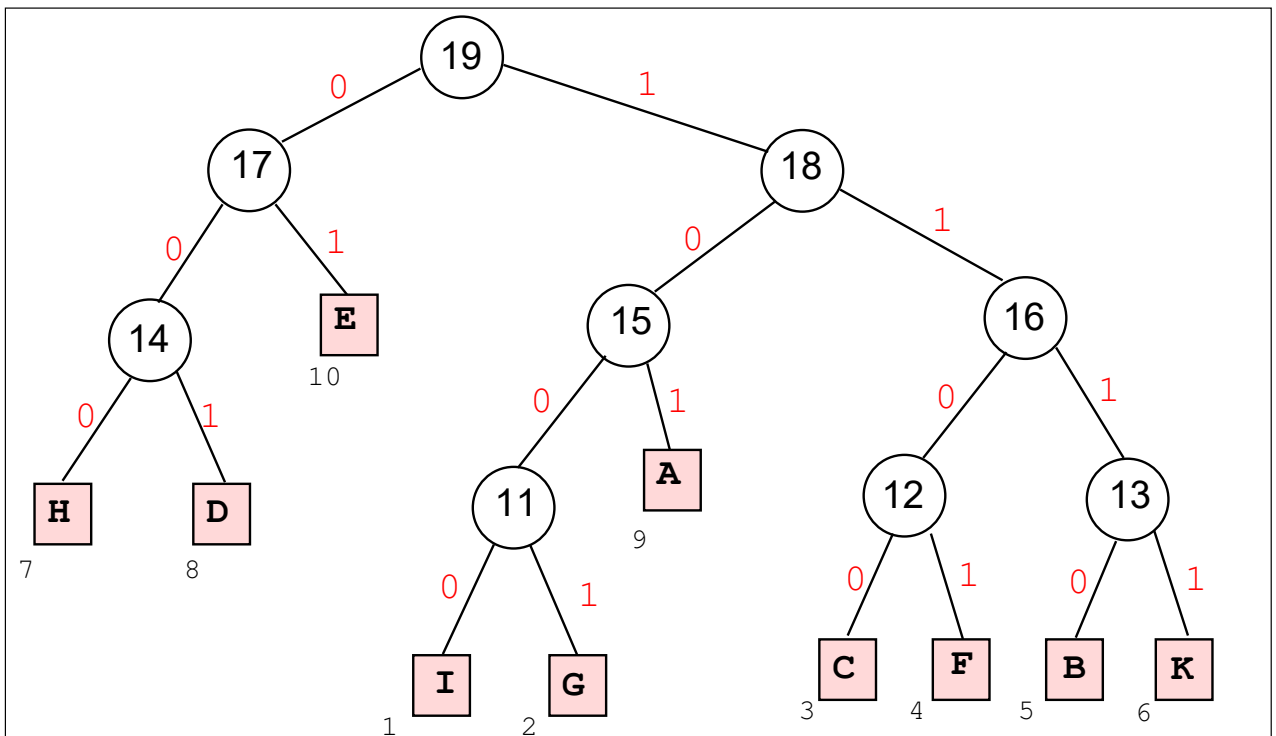
| $a_i$ | I | G | C | F | B | K | H | D | A | E |
|---|---|---|---|---|---|---|---|---|---|---|
| $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| $p_i$ | 0.06 | 0.07 | 0.07 | 0.08 | 0.08 | 0.09 | 0.09 | 0.10 | 0.15 | 0.21 |
| pred | 11 | 11 | 12 | 12 | 13 | 13 | 14 | 14 | 15 | |
| mark | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| $i$ | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| $p_i$ | 0.13 | 0.15 | 0.17 | 0.19 | 0.28 | **0.32** | | | |
| pred | 15 | **16** | **16** | | | | | | — |
| mark | 0 | **0** | **1** | | | | | | — |

| $a_i$ | I | G | C | F | B | K | H | D | A | E |
|---|---|---|---|---|---|---|---|---|---|---|
| $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| $p_i$ | 0.06 | 0.07 | 0.07 | 0.08 | 0.08 | 0.09 | 0.09 | 0.10 | 0.15 | 0.21 |
| pred | 11 | 11 | 12 | 12 | 13 | 13 | 14 | 14 | 15 | 17 |
| mark | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| $i$ | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| $p_i$ | 0.13 | 0.15 | 0.17 | 0.19 | 0.28 | 0.32 | 0.40 | 0.60 | 1.00 |
| pred | 15 | 16 | 16 | 17 | 18 | 18 | 19 | 19 | — |
| mark | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | — |

The resulting optimal coding tree $T$ can now be built top-down.



The coding function $c_3$ can then be read off from the tree:

| | A | B | C | D | E | F | G | H | I | K |
|---|---|---|---|---|---|---|---|---|---|---|
| $p(a)$ | 0.15 | 0.08 | 0.07 | 0.10 | 0.21 | 0.08 | 0.07 | 0.09 | 0.06 | 0.09 |
| $c_3$ | 101 | 1110 | 1100 | 001 | 01 | 1101 | 1001 | 000 | 1000 | 1111 |

$B(T,p) = 0.21 \cdot 2 + (0.15 + 0.10 + 0.09) \cdot 3 + (0.08 + 0.07 + 0.08 + 0.07 + 0.06 + 0.09) \cdot 4 =$ **3.24**.

Our text of length $100\,000$ letters would lead to a code of length $324\,000$. This is optimal.

**Algorithm Huffman**(p[1..$n$])

**Input:** probability/weight vector p[1..$n$], sorted: p[1] $\leq \cdots \leq$ p[$n$].

**Output:** optimal tree $T$, represented as pred[1..$2n-2$] and mark[1..$2n-2$]

```
(1)   pred[1] ← n + 1;
(2)   pred[2] ← n + 1;
(3)   p[n + 1] ← p[1] + p[2];
(4)   m ← 3 // first node in [1..n] not yet processed
(5)   h ← n + 1 // first node in [n + 1..2n − 1] not yet processed
(6)   for b from n + 2 to 2n − 1 do
(7)     // the following two lines find, in i and j, the positions of the
(8)     // two "lightest" letters available at present
(9)     if m ≤ n and p[m] ≤ p[h] then i ← m; m++ else i ← h; h++;
(10)    if m ≤ n and (h = k or p[m] ≤ p[h]) then j ← m; m++ else j ← h; h++;
(11)    pred[i] ← b;
(12)    mark[i] ← 0;
(13)    pred[j] ← b;
(14)    mark[j] ← 1;
(15)    p[b] ← p[i] + p[j];
(16)  Output: pred[1..2n − 2] and mark[1..2n − 2].
```

Using `pred`$[1..2n-1]$ and `mark`$[1..2n-1]$, one can build the optimal Huffman tree as follows:

Allocate array `leaf`$[1..n]$ with leaf node objects
and array `inner`$[n+1..2n-1]$ with inner node objects.

```
(1)   for i from 1 to n do
(2)     leaf[i].letter ← letter a_i.
(3)     if mark[i] = 0
(4)       then inner[pred[i]].left ← leaf[i]
(5)       else inner[pred[i]].right ← leaf[i]
(6)   for i from n + 1 to 2n − 2 do
(7)     if mark[i] = 0
(8)       then inner[pred[i]].left ← inner[i]
(9)       else inner[pred[i]].right ← inner[i]
(10)  return inner[2n − 1] // root node
```

Beware: If carrying out the algorithm by hand, it is much simpler to draw the tree starting from the root.

## Theorem 5.2.7

The algorithm **Huffman** is correct and has running time $O(n \log n)$, if $n$ denotes the number of letters of the alphabet $A$.