Greedy Algorithms

(Version of 21 November 2005)

- There are many problems where an *optimal* solution is sought.
- There are many choices to be explored at each solution step.
- One approach is to always make the choice that currently *seems*
  to give the highest gain, that is to be as *greedy* as possible
  and make a *locally optimal* choice in the hope
  that the remaining *unique* subproblem
  leads to a *globally optimal* solution.
- For many problems, a greedy algorithm gives an optimal solution,
  but not for all problems.

Example of a Greedy Algorithm

Coin change problem:

To give change of n units, given a set of denominations, what is the minimum number of coins to use?

Example:

$7 = 2 + 2 + 2 + 1$, hence four coins are needed.

Greedy algorithm:

Always give a coin of the *largest* possible denomination and then repeat on the remaining amount due.

Specification and SML Code

FUNCTION *change Denominations n*

TYPE: *int list* → *int* → *int*

PRE: *Denominations* is sorted by decreasing values and has 1;

n and all values in *Denominations* are natural numbers

POST: an ideally minimal number of coins, with values in *Denominations*, necessary to give change for an amount of *n* units

```
fun change Ds x =
    if x = 0 then 0
    else if (List.hd Ds) <= x then
              1 + change Ds (x - List.hd Ds)
         else change (List.tl Ds) x
```

Question: What is a variant for this function?

When Does it Work?

```
- change [10,5,2,1] 13 ;
val it = 3 : int
- change [5,4,3,1] 7 ;
val it = 3 : int
```

But the second answer is not the optimal one,
since we can also use a two-coin combination, because $7 = 4 + 3$.

The denominations 4 and 3 *leapfrog* over 5, that is $4 + 3 = 7 \geq 5$.
Leapfrogging *may* imply the need for more coins on some problems.

With the currency used in Sweden, there is no leapfrogging.
For such currencies, the given function is optimal: for them, we can add a no-leapfrogging pre-condition and rephrase the post-condition into "the minimum number of coins, …".

Example: Huffman Data-Compression Codes

Suppose we want to store compactly a file of 100000 characters (which normally takes $100000 \cdot 8 = 800000$ bits), with the following frequencies of characters:

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

Inefficient code: Suppose we use three bits for *each* character:

	a	b	c	d	e	f
Frequency	45%	13%	12%	16%	9%	5%
Codeword	000	001	010	011	100	101

In order to store the file, we would then need 300000 bits.

Variable-Length Codes

But suppose we use the following *variable-length code* instead:

	a	b	c	d	e	f
Frequency	45%	13%	12%	16%	9%	5%
Codeword	0	101	100	111	1101	1100

The string '`bed`' is then coded as '`1011101111`',
and there is *no* ambiguity, since no codeword is a prefix of another.

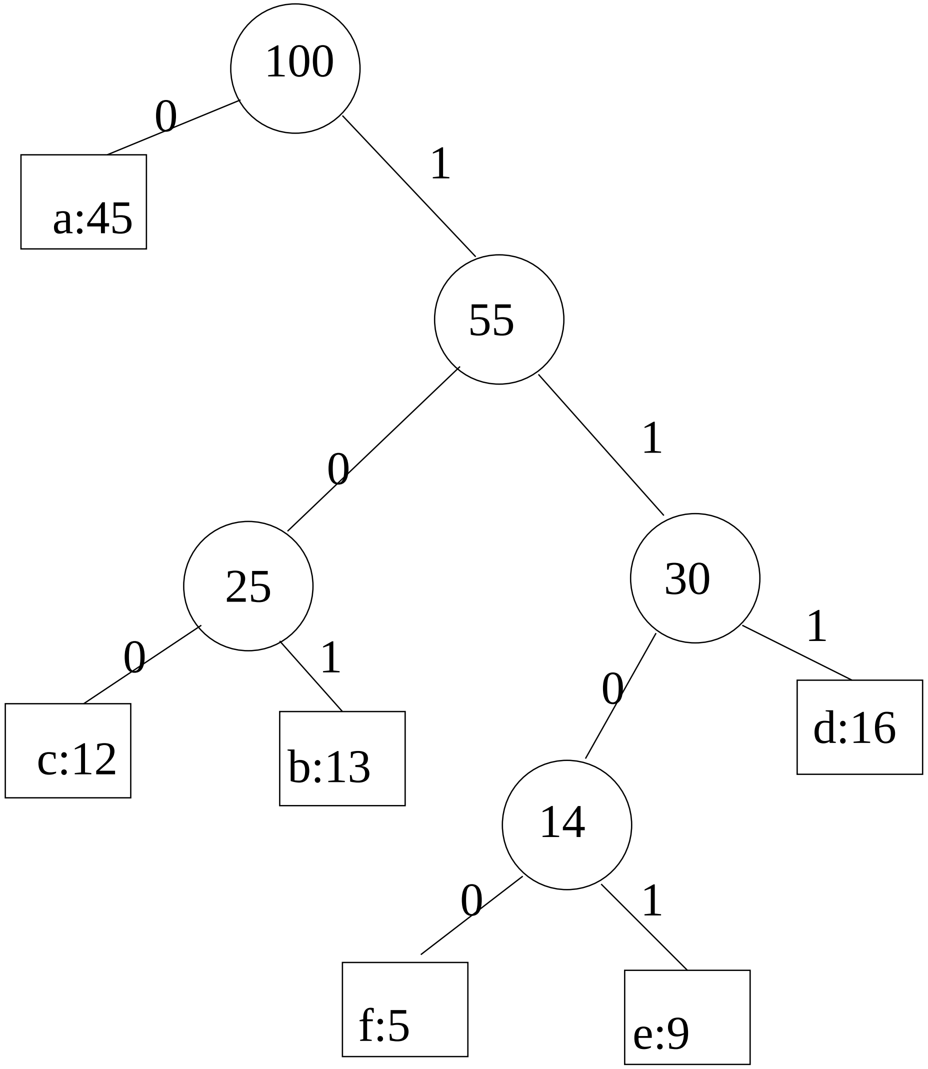
In order to store the file of 100,000 characters, we would now need

$$(0.45 \cdot 1 + 0.13 \cdot 3 + 0.12 \cdot 3 + 0.16 \cdot 3 + 0.09 \cdot 4 + 0.05 \cdot 4) \cdot 100000 = 224000 \text{ bits.}$$

Savings of 20% to 90% are typical with this technique.

Prefix Codes and their Representation

- A *code* is an assignment of messages (characters, strings, commands to do things, …) to sequences of bits.
- In a *prefix*(*-free*) *code*, no codeword is a prefix of another.
- A prefix code can be *decoded* unambiguously.
- Prefix codes achieve *optimal* data compression among all codes.
- A Huffman code is an optimal prefix code.
- A prefix code can be *represented* as a labelled binary tree:
  label each left branch 0 and each right branch 1.
  To decode a word, move down the appropriate branches
  until reaching a leaf with a character.


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

SML Representation of Huffman Codes

```
datatype huffTree = Leaf of int * char
                  | Node of int * huffTree * huffTree
REPRESENTATION CONVENTION:
 - a Huffman tree for character c of frequency f
   is represented by Leaf(f,c);
 - a Huffman tree with total frequency f, left subtree L,
   and right subtree R is represented by Node(f,L,R),
   and   the edge to L is implicitly labelled with the bit 0
   while the edge to R is implicitly labelled with the bit 1
REPRESENTATION INVARIANT: for Node(f,L,R):
 - the root frequency of L is at most the root freq. of R
 - f is the sum of the root frequencies of L and R
fun freq (Leaf(f,_)) = f | freq (Node(f,_,_)) = f
```

Using a Min-Heap

Maintain a *min-priority queue*, as a *min-heap*, with the Huffman trees as items and the frequencies at their roots as keys.

```
structure huffTreeOrder : totalOrder =
    struct
        type t = huffTree
        fun eq(x,y)  = (freq x) =  (freq y)
        fun lt(x,y)  = (freq x) <  (freq y)
        fun leq(x,y) = (freq x) <= (freq y)
    end
structure huffTreeHeap = leftistHeap(huffTreeOrder)
```

A *leftist heap* is another way of implementing heaps: see the program.

Constructing the Heap

```
fun listToHeap [] = huffTreeHeap.empty
  | listToHeap ((f,c)::xs) =
    huffTreeHeap.insert (Leaf(f,c), (listToHeap xs))
```

Merging Two Huffman Trees

```
(* PRE: freq t1 <= freq t2 *)
fun mergeHuffTree t1 t2 = Node((freq t1)+(freq t2),t1,t2)
```

Note that the given pre-condition saves an `if ...then ...else ...` in the `mergeHuffTree` function itself. Even some calling functions can do so: see the `collapseHeap` function below for an example.

Constructing the Huffman Code

Merge the 2 Huffman trees with the smallest frequencies at the root, until only one Huffman tree is left.

Help function to extract the tree with the smallest root frequency:

```
fun extractMin h =
    (huffTreeHeap.findMin h, huffTreeHeap.deleteMin h)
```

Exercise: Implement this function better
and add it to the `leftistHeap` functor.

Constructing the Huffman Code (base)

If the heap, which is *non-empty* by pre-condition, has only one element, then return that heap:

```
fun collapseHeap h =
    let
        val (min,h') = extractMin h
    in
        if (huffTreeHeap.isEmpty h') then
            h
```

Constructing the Huffman Code (step)

If the heap has at least two elements,
then delete its two smallest elements,
insert their merger into the heap, and recurse:

```
    else
        let
            val (nextmin,h'') = extractMin h'
            val newTree = mergeHuffTree min nextmin
            val h''' = huffTreeHeap.insert(newTree,h'')
        in
            collapseHeap h'''
        end
end
```

Top-Level Function to Construct a Huffman Code

Given a character-frequency list, which is *non-empty* by pre-condition, construct a Huffman code:

```
fun makeHuffTree freqList =
    let
        val initialHeap = listToHeap freqList
        val collapsedHeap = collapseHeap initialHeap
    in
        huffTreeHeap.findMin collapsedHeap
    end
val testFreq = [(16,#"d"),  (9,#"e"),  (5,#"f"),
                (45,#"a"), (13,#"b"), (12,#"c")]
val huffTree = makeHuffTree testFreq
```

Huffman's Algorithm

- Huffman's algorithm is another example of a greedy algorithm.
- It takes $O(n \lg n)$ time for a set of n characters.
- Proving that it actually gives the optimal code is another matter.

Greedy Algorithms

- Greedy algorithms are efficient.
- In some cases, they actually construct an optimal solution.
- Even when they do not construct an optimal solution,
  their solution can be used as a *starting point*
  to actually construct an optimal solution.