Presentation for use with the textbook, Algorithm Design and Applications, by M. T. Goodrich and R. Tamassia, Wiley, 2015

The Greedy Method

Civil War Knapsack. U.S. government image. Vicksburg National Military Park. Public domain.

1

1

Optimization Problems

- An optimization problem can be abstracted as (V, D, C, F), where V is a set of variables, D is the domain for variables, C is a set of constraints over V, and f is a numeric function over V. The goal is to find an assignment of σ:V$\rightarrow$D such that all C are satisfied and f generates the minimum (or maximum) value under the assignment.

- Example:

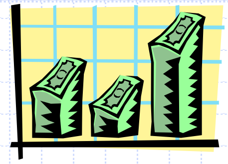
  Find the minimum value of $f(x) = -x^2+x$.

  V = {x}, D = R, C = true, F = f.

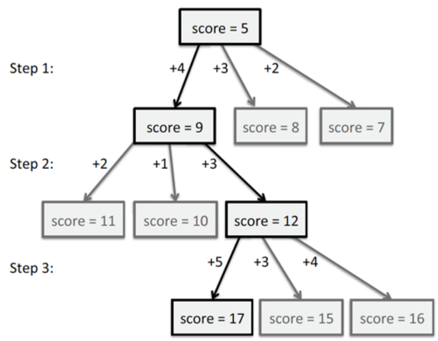
2

2

The Greedy Method



- Using an easy-to-compute order to make a sequence of choices, each of the choice is best from all of those that are currently possible (local optimal).



3

3

Knapsack Problem

- Example: Given a set of 5 items with value and weight. How to select a subset of items whose total weight is under 11, but the total value is maximal.
- Optimal Solution:
  - { 3, 4 } has weight 11 and value 40.

W = 11

#	value	weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

- Greedy choice: Choose item with maximum v_i / w_i.
- Greedy Solution:
  - { 5, 2, 1 } has weight 10 and value 35 $\Rightarrow$ greedy not optimal.

4

Knapsack Problem (general description)

- Given n items of weights $s_1, s_2, \ldots, s_n$, and values $v_1, v_2, \ldots, v_n$ and weight C, the knapsack capacity, the objective is to find integers $x_1, x_2, \ldots, x_n$ in { 0, 1 } that maximize the sum

$$\sum_{i=1}^{n} x_i v_i$$

subject to the constraint

$$\sum_{i=1}^{n} x_i s_i \le C$$

#	value	weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

e.g. $(x_1, x_2, x_3, x_4, x_5) = (0, 0, 1, 1, 0)$

5

Fractional Knapsack Problem

- Given n items of weights $s_1, s_2, \ldots, s_n$, and values $v_1, v_2, \ldots, v_n$ and weight C, the knapsack capacity, the objective is to find nonnegative real numbers $x_1, x_2, \ldots, x_n$ between 0 and 1 that maximize the sum

$$\sum_{i=1}^{n} x_i v_i$$

subject to the constraint

$$\sum_{i=1}^{n} x_i s_i \le C$$

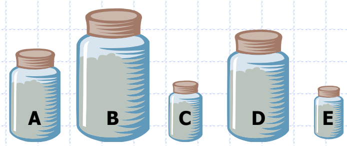
This problem can be solved by linear programming.

6

Fractional Knapsack Example

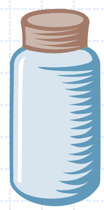
- Given: A set S of n items, with each item i having
  - b_i - value
  - w_i - weight
- Goal: Choose items with maximum total value but with weight at most W.

Items:



Weight:	4 ml	8 ml	2 ml	6 ml	1 ml
Value:	$12	$32	$40	$30	$50
Unit value: ($ per ml)	3	4	20	5	50

"knapsack"



10 ml

Solution:
- 100% of E (1 ml)
- 100% of C (2 ml)
- 100% of D (6 ml)
- 12.5% of B (1 ml)

7

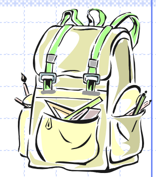
7

Fractional Knapsack Problem

- This problem can easily be solved using the following greedy strategy:
  - For each item compute $v_i = b_i / w_i$, the ratio of its value to its weight.
  - Sort the items by decreasing ratio, and fill the knapsack with as much as possible from the first item, then the second, and so forth.
  - This problem reveals many of the characteristics of a greedy algorithm: *The algorithm consists of a simple iterative procedure that selects an item which produces the largest immediate gain while maintaining feasibility (i.e., no violation of constraints).*

8

Fractional Knapsack Algorithm



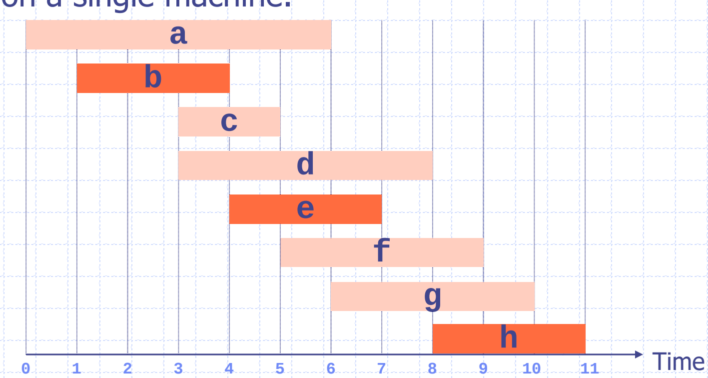
- Greedy choice: Choose item with highest value (per unit weight)
  - Run time: O(n log n). Why?
- Correctness: Suppose there is a better solution than the greedy one.
  - Then there must be an item i with higher value than a chosen item j, $v_i>v_j$ but $x_i<w_i$ and $x_j>0$.
  - so that we substitute some i with j, we get a better solution.
  - However, the algorithm will assign $x_i + \min\{ w_i-x_i, x_j \}$ to x_i, not the current x_i.
  - Thus, there is no better solution than the greedy one.

Algorithm *fractionalKnapsack(S, W)*
  Input: set S of items w/ values b_i and weight w_i; max. weight W
  Output: amount x_i of each item i to maximize value w/ weight at most W
  for *each item i in S*
    $x_i \leftarrow 0$
    $v_i \leftarrow b_i / w_i$ {value}
  $w \leftarrow 0$ {total weight}
  while $w < W$ && $|S| > 0$
    remove item i with highest v_i
    $x_i \leftarrow \min\{w_i, W - w\}$
    $w \leftarrow w + \min\{w_i, W - w\}$
  return $(x_1, x_2, \dots, x_n)$

9

Task Scheduling

- Given: a set T of n tasks, start time, s_i and finish time, f_i (where $s_i < f_i$)
- Goal: Perform a maximum number of compatible jobs on a single machine.



10

10

Task Scheduling: Greedy Algorithms

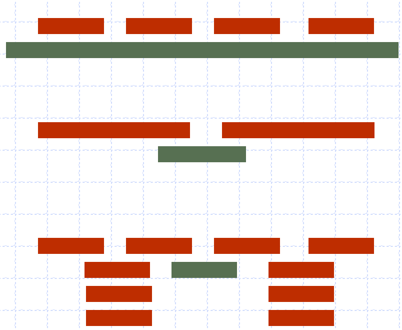
- Greedy template. Consider jobs in some order. Repeatedly take each job in the order, provided it's compatible with the ones already taken.
  - [Earliest start time] Consider jobs in ascending order of s_j.
  - [Earliest finish time] Consider jobs in ascending order of f_j.
  - [Shortest interval] Consider jobs in ascending order of $f_j - s_j$.
  - [Fewest conflicts] For each job j, count the number of conflicting jobs c_j. Schedule in ascending order of c_j.

11

11

Task Scheduling: Greedy Algorithms

- Greedy template. Consider jobs in some natural order. Take each job provided it's compatible with the ones already taken.



counterexample for earliest start time

counterexample for shortest interval

counterexample for fewest conflicts

12

12

Task Scheduling: Greedy Algorithm

- Greedy algorithm. Consider jobs in increasing order of finish time. Take each job in the order, provided it's compatible with the ones already taken.

```
Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.
   set of jobs selected
A ← ϕ
for j = 1 to n {
   if (job j compatible with A)
      A ← A ∪ {j}
}
return A
```

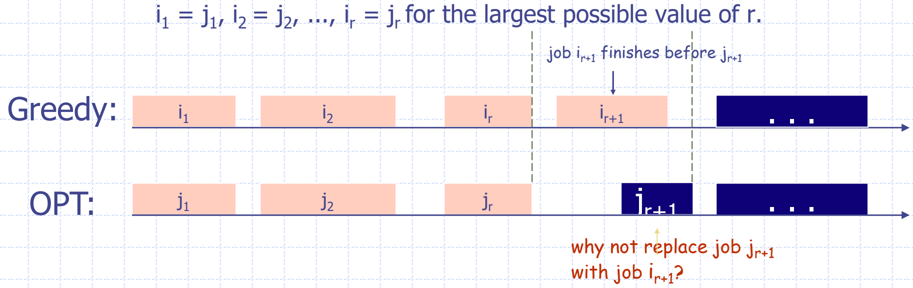
- Implementation: O(n log n).
  - Let job j* denote the job that was added last to A.
  - Job j is compatible with A if $s_j \geq f_{j^*}$, i.e., j starts after j* finished.

13

13

Task Scheduling: Analysis

- Theorem. The greedy algorithm is optimal.
- Proof. (by contradiction)
  - Assume greedy is not optimal, and let's see what happens.
  - Let $i_1, i_2, \ldots i_k$ denote set of jobs selected by greedy.
  - Let $j_1, j_2, \ldots j_m$ denote set of jobs in the optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ for the largest possible value of r.

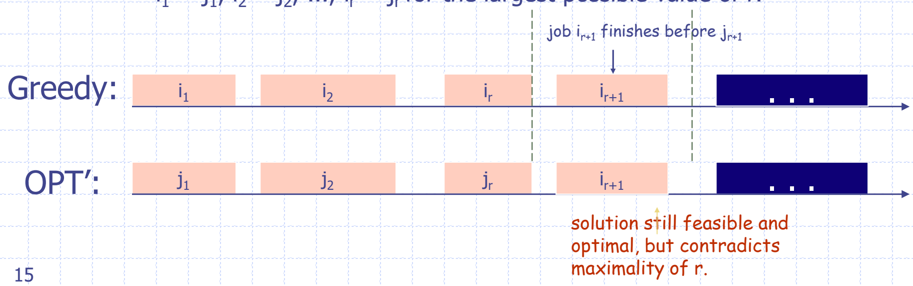


14

14

Task Scheduling: Analysis

- Theorem. The greedy algorithm is optimal.
- Proof. (by contradiction)
  - Assume greedy is not optimal, and let's see what happens.
  - Let i_1, i_2, ... i_k denote set of jobs selected by greedy.
  - Let j_1, j_2, ... j_m denote set of jobs in the optimal solution with $i_1 = j_1$, $i_2 = j_2$, ..., $i_r = j_r$ for the largest possible value of r.



15

15

How to Show a Greedy Method is Optimal?

- In general, a greedy method is simple to describe, efficient to run, but difficult to prove.
- To show a greedy method is **not** optimal, we need to find a **counterexample**.
- To show a greedy method is indeed optimal, we use the following proof strategy:
- Suppose S is the solution found by the greedy method and Opt is an optimal solution that differs from S minimally. If S = Opt, we are done. If not, we "modify" Opt to obtain another optimal solution Opt', such that Opt' has less difference than Opt comparing to S. That's a contradiction to the assumption that Opt differs from S minimally.

16

16

Data Compression

- Given a string X, efficiently encode X into a smaller string Y
  - Saves memory and/or bandwidth
- A good approach: **Huffman encoding**
  - Compute frequency f(c) for each character c.
  - Encode high-frequency characters with short code words
  - No code word is a prefix for another code
  - Use an optimal encoding tree to determine the code words

17

17

Motivation

The motivations for data compression are obvious:

- reducing the space required to store files on disk or tape
- reducing the time to transmit large files.

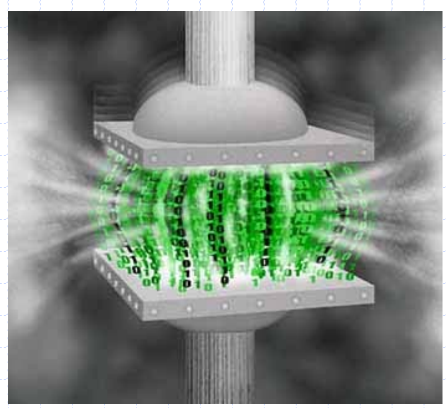


Image Source : plus.maths.org/issue23/ features/data/data.jpg

Huffman savings are between 20% - 90%

18

Basic Idea :

Let the set of characters in the file be $C=\{c_1, c_2, \ldots, c_n\}$. Let also $f(c_i)$, $1\leq i\leq n$, be the frequency of character c_i in the file, i.e., the number of times c_i appears in the file.

It uses a variable-length code table for encoding a source symbol (such as a character in a file) where the variable-length code table has been derived in a particular way based on the frequency of occurrence for each possible value of the source symbol.

19

Example:

Suppose you have a file with 100K characters.

For simplicity assume that there are only 6 distinct characters in the file from *a* through *f*, with frequencies as indicated below.

We represent the file using a unique binary string for each character.

	a	b	c	d	e	f
Frequency (in 100s)	45	13	12	16	9	5
Fixed-length code-word	000	001	010	011	100	101

Space = (45*3 + 13*3 + 12*3 + 16*3 + 9*3 + 5*3) * 1000

= **300K bits**

20

Can we do better ?? **YES !!**

By using **variable-length** codes instead of fixed-length codes.

Idea : Giving frequent characters short code-words, and infrequent characters long code-words.

i.e. The length of the encoded character is inversely proportional to that character's frequency.

	a	b	c	d	e	f
Frequency (in 1000s)	45	13	12	16	9	5
Fixed-length code-word	000	001	010	011	100	101
Variable-length code-word	0	101	100	111	1101	1100

Space = (45*1 + 13*3 + 12*3 + 16*3 + 9*4 + 5*4) * 1000

= **224K bits** (Savings = 25%)

21

PREFIX CODES :

Codes in which no code-word is also a prefix of some other code-word.

("prefix-free codes" would have been a more appropriate name)

Variable-length code-word	0	101	100	111	1101	1100

It is very easy to encode and decode using prefix codes.

No Ambiguity !!

It is possible to show (although we won't do so here) that the optimal data compression achievable by a character code can always be achieved with a prefix code, so there is no loss of generality in restricting attention to prefix codes.

22

Benefits of using Prefix Codes:

Example:

	a	b	c	d	e	f
Variable-length code-word	0	101	100	111	1101	1100

	f	a	c	e		
Encoded as	1100	0	100	1101	=	110001001101

To decode, we have to decide where each code begins and ends, since they are no longer all the same length. But this is easy, since, no codes share a prefix. This means we need only scan the input string from left to right, and as soon as we recognize a code, we can print the corresponding character and start looking for the next code.

In the above case, the only code that begins with "1100.." is "f", so we can print "f" and start decoding "0100...", get "a", etc.

23

Benefits of using Prefix Codes:

Example:

To see why the no-common prefix property is essential, suppose that we encoded "e" with the shorter code "110"

	a	b	c	d	e	f
Variable-length code-word	0	101	100	111	1101	1100
Variable-length code-word	0	101	100	111	110	1100

FACE = 11000100110

When we try to decode "1100"; we could not tell whether

1100 = "f"

or

1100 = 110 + 0 = "ea"

24

Representation:

The Huffman algorithm is represented as:

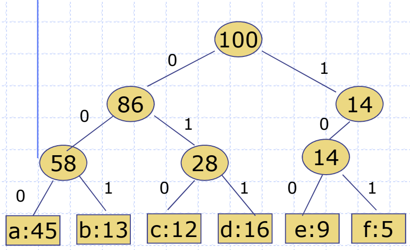
- binary tree
- each edge represents either 0 or 1
  - 0 means "go to the left child"
  - 1 means "go to the right child."
- each leaf corresponds to the sequence of 0s and 1s traversed from the root to reach it, i.e. a particular code.

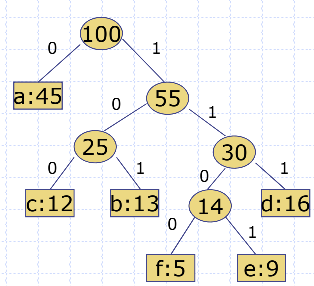
Since no prefix is shared, all legal codes are at the leaves, and decoding a string means following edges, according to the sequence of 0s and 1s in the string, until a leaf is reached.

25

	a	b	c	d	e	f
Frequency (in 1000s)	45	13	12	16	9	5
Fixed-length code-word	000	001	010	011	100	101

	a	b	c	d	e	f
Frequency (in 1000s)	45	13	12	16	9	5
Variable-length code-word	0	101	100	111	1101	1100





Labeling :

leaf -> character it represents : frequency with which it appears in the text.

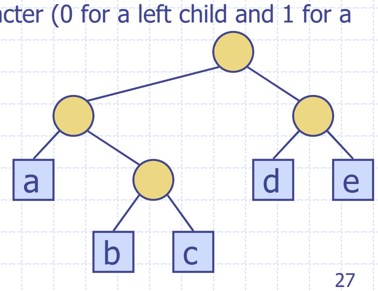
internal node -> frequency with which all leaf nodes under it appear in the text (i.e. the sum of their frequencies).

26

Encoding Tree Summary

- A **code** is a mapping of each character of an alphabet to a binary code-word
- A **prefix code** is a binary code such that no code-word is the prefix of another code-word
- An **encoding tree** represents a prefix code
  - Each external node stores a character
  - The code word of a character is given by the path from the root to the external node storing the character (0 for a left child and 1 for a right child)

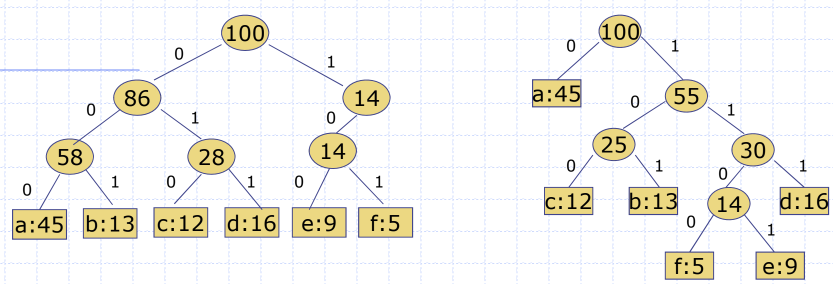
00	010	011	10	11
a	b	c	d	e



27

27

Optimal Code



An optimal code for a file is always represented by a ***proper* binary tree**, in which every non-leaf node has two children.

The fixed-length code in our example is not optimal since its tree, is not a full binary tree: there are code-words beginning 10 . . . , but none beginning 11 ..

Since we can now restrict our attention to full binary trees, we can say that if *C* is the alphabet from which the characters are drawn, then the tree for an optimal prefix code has exactly **|C|** *leaves, one for each letter of the alphabet, and exactly* **|C| - 1** internal nodes.

28

Given a tree **T** corresponding to a prefix code, it is a simple matter to compute the number of bits required to encode a file.

For each character **c** in the alphabet **C**,

- **f(c)** denote the <u>frequency</u> of c in the file
- $\mathbf{d_T(c)}$ denote the <u>depth</u> of c's leaf in the tree.

($\mathbf{d_T(c)}$ is also the <u>length</u> of the code-word for character c)

The number of bits required to encode a file is thus

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

which we define as the cost of the tree.

29

<u>Constructing a Huffman code</u>

Huffman invented a <u>greedy</u> algorithm that constructs an optimal prefix code called a *<u>Huffman code</u>. The algorithm builds the tree T corresponding to the optimal code in a <u>bottom-up</u> manner.*

It begins with a set of |C| leaves and performs a sequence of |C| - 1 "merging" operations to create the final tree.

Greedy Choice?

The <u>two smallest nodes</u> are chosen at each step, and this local decision results in a globally optimal encoding tree.

In general, greedy algorithms use local minimal/maximal choices to produce a global minimum/maximum.

30

HUFFMAN(*C*)

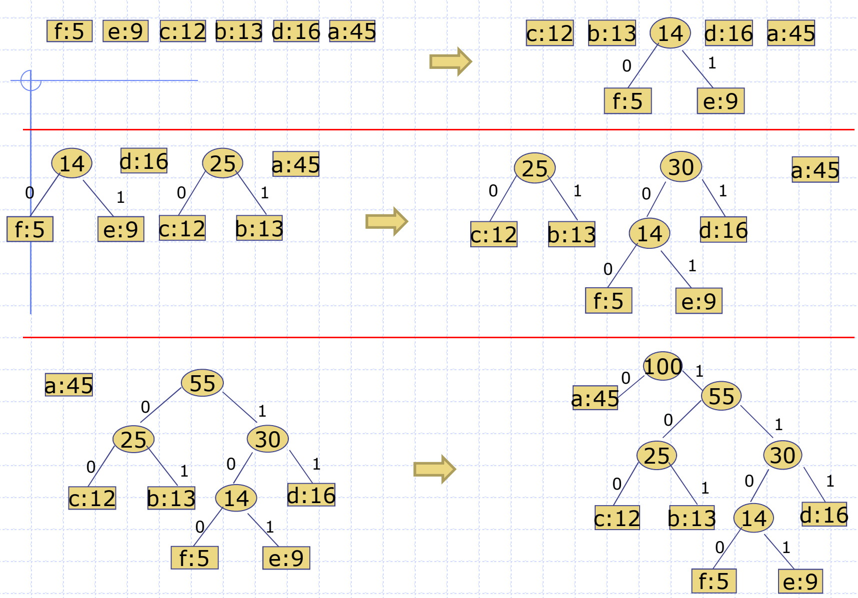
```
n ← |C|
Q ← BUILD-MIN-HEAP(C)   // using frequency f[c] for c in C
for i ← 1 to n - 1
    do ALLOCATE-NODE(z)    // create a new node z
       left[z] ← x ← EXTRACT-MIN(Q)
       right[z] ← y ← EXTRACT-MIN(Q)
       f[z] ← f[x] + f[y]   // frequency of z
       INSERT(Q, z)
return EXTRACT-MIN(Q)
```

C is a set of *n* characters: each character *c* in *C* is an object with a defined frequency *f*[*c*].

A min-priority queue *Q*, keyed on *f*, is used to identify the two least-frequent objects to merge together and produce z. For Q, z is a new character with frequency f[z] = f[x]+f[y]. For the tree, z is a new internal node with children x and y.

31

The steps of Huffman's algorithm



32

HUFFMAN(*C*)

```
1 n ← |C|
2 Q ← BUILD-MIN-HEAP(C)                              // O(n)
3 for i ← 1 to n - 1                                 // n *
4     do ALLOCATE-NODE(z)                            // O(1)
5        left[z] ← x ← EXTRACT-MIN(Q)                // O(log n)
6        right[z] ← y ← EXTRACT-MIN(Q)               // O(log n)
7        f[z] ← f[x] + f[y]   // frequency of z      // O(1)
8        INSERT(Q, z)                                // O(log n)
9 return EXTRACT-MIN(Q)                              // O(1)
```

C is a set of *n* characters: each character *c* in *C* is an object with a defined frequency *f*[*c*].

A min-priority queue *Q*, keyed on *f*, is used to identify the two least-frequent objects to merge together and produce z. For Q, z is a new character with frequency f[z] = f[x]+f[y]. For the tree, z is a new internal node with children x and y.

33

Running Time Analysis

Assumes that *Q* is implemented as a binary min-heap.

- For a set *C* of *n* characters, the initialization of *Q* in line 2 can be performed in *O*(*n*) time using the BUILD-MIN-HEAP procedure.

- The *for* loop in lines 3-8 is executed exactly *n* - 1 times. Each heap operation requires time *O(log n)*. The loop contributes = (*n* - 1) * *O(log n)* = *O(nlog n)*

Thus, the total running time of HUFFMAN on a set of *n* characters = *O*(*n*) + *O(nlog n)*

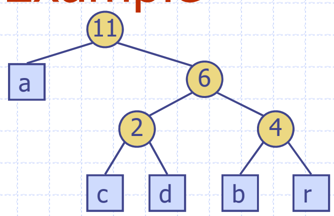
= *O*(*n* log *n*)

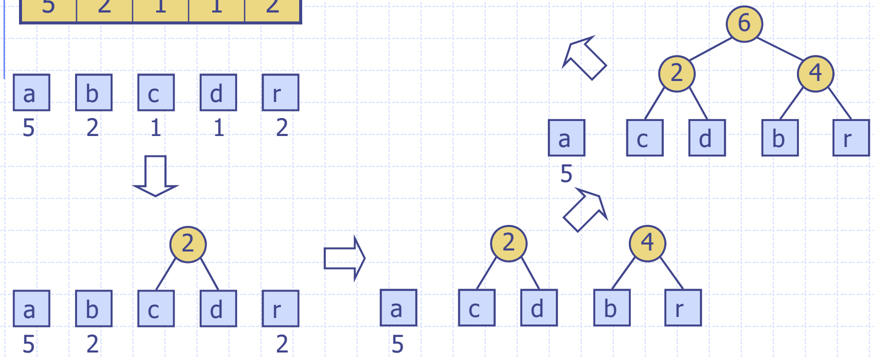
34

Huffman Code Example

X = abracadabra
Frequencies

a	b	c	d	r
5	2	1	1	2





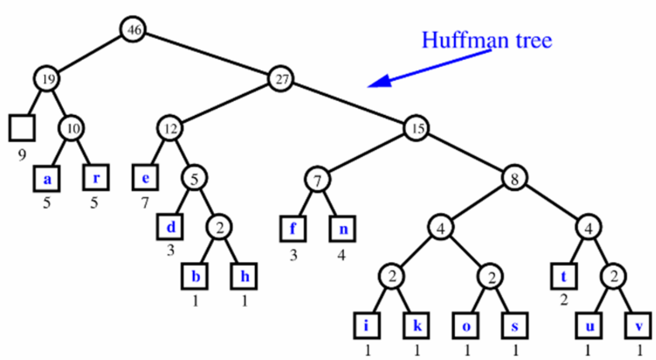
35

35

Huffman Code Example

String: **a fast runner need never be afraid of the dark**

Character		a	b	d	e	f	h	i	k	n	o	r	s	t	u	v
Frequency	9	5	1	3	7	3	1	1	1	4	1	5	1	2	1	1



36

36

Possible Quiz Question:

Suppose you have a file with 100K characters.

For simplicity assume that there are only 8 distinct characters in the file from *a* through *h*, with frequencies as indicated below.

We represent the file using a unique binary string for each character.

	a	b	c	d	e	f	g	h
Frequency (in 100s)	35	13	12	16	9	5	5	5
Code-word								

Question: What is the prefix code for this set of characters which produces the shortest binary string for this file?

37

Possible Quiz Question:

Given two sorted lists A and B, merge(A, B) will return a new list consisting of elements from A and B with cost $O(|A| + |B|)$, where $|X|$ is the length of X, i.e., the number of elements in list X.

Please design an efficient algorithm (as fast as you can) which merge n sorted lists into a single list by calling merge(A, B), where the sizes of these n lists $S = \{L_1, L_2, \ldots, L_n\}$, are as follows: for $1 \leq i < n$, $|L_i| = 2^i$, and $|L_n| = 2$. Thus, the total number of elements in these n lists is 2^n. Please analyze the complexity of your algorithm in terms of n.

38

Correctness of Huffman's algorithm

To prove that the greedy algorithm HUFFMAN is correct, we show that the problem of determining an optimal prefix code exhibits the greedy-choice and optimal-substructure properties.

39

The Greedy-Choice Property

Lemma 1: Let *C* be an alphabet in which each character *c* in *C* has frequency *f*[*c*]. Let *x* and *y* be two characters in *C* having the lowest frequencies. Then there exists an optimal prefix code for *C* in which the code words for *x* and *y* have the same length and differ only in the last bit.

Why ?

Must be on the bottom (least frequent)

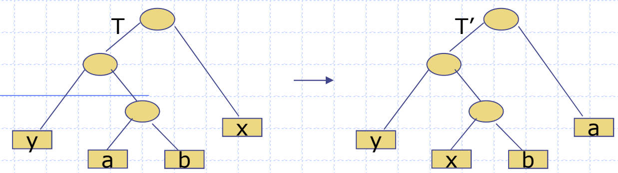
Full tree, so arrange them as siblings, and so differ in the last bit.

Proof Idea of **Lemma 1**:

The idea of the proof is to take the tree *T* representing an arbitrary optimal prefix code and modify it to make a tree representing another optimal prefix code such that the characters *x* and *y* appear as sibling leaves of maximum depth in the new tree. If we can do this, then their code words will have the same length and differ only in the last bit.

40

Proof of **Lemma 1**



Let a and b be two characters that are are sibling leaves of maximum depth in *T*, and x and y are the two characters of the minimum frequency. Without loss in generality, assume that f[x] < f[y] < f[a] < f[b]. Then we must have $d_T(x) = d_T(y) = d_T(a) = d_T(b)$.

Proof: Exchange the positions of a and x in T, to produce T′.

The difference in cost between T and T′ is

$B(T) - B(T') = \Sigma f(c)\, d_T(c) - \Sigma f(c)\, d_{T'}(c)$

$= f[x]\, d_T(x) + f[a]\, d_T(a) - f[x]\, d_{T'}(x) - f[a]\, d_{T'}(a)$

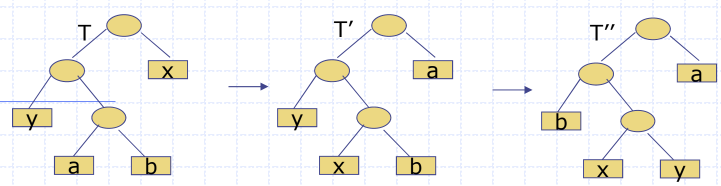
$= f[x]\, d_T(x) + f[a]\, d_T(a) - f[x]\, d_T(a) - f[a]\, d_T(x)$

$= (f[a] - f[x])(d_T(a) - d_T(x)) \geq 0$ // f[x] is min and $d_T(a)$ is max

On the other hand, $B(T) - B(T') \leq 0$, because B(T) is minimal.

So $B(T) - B(T') = 0$ and $d_T(a) = d_T(x)$.

41



Similarly exchanging the positions of b and y in T′, to produce T′′ does not increase the cost,

B(T′) – B(T′′) is 0.

Since T is optimal, so is T′ and T′′.

Thus, T′′ is an optimal tree in which x & y appear as sibling leaves of maximum depth from which **Lemma 1** follows.

42

Lemma 2: Let C be a given alphabet with frequency $f[c]$ defined for each character $c \in C$. Let x and y be two characters in C with minimum frequency. Let C′ be the alphabet C with characters x,y removed and (new) character z added, so that **C' = C – {x,y} U {z};** define f for C′ as for C, except that f[z] = f[x] + f[y]. Let T′ be any tree representing an optimal prefix code for the alphabet C′. Then the tree T, obtained from T′ by replacing the leaf node for z with an internal node having x and y as children, represents an optimal prefix code for the alphabet C.

Proof:

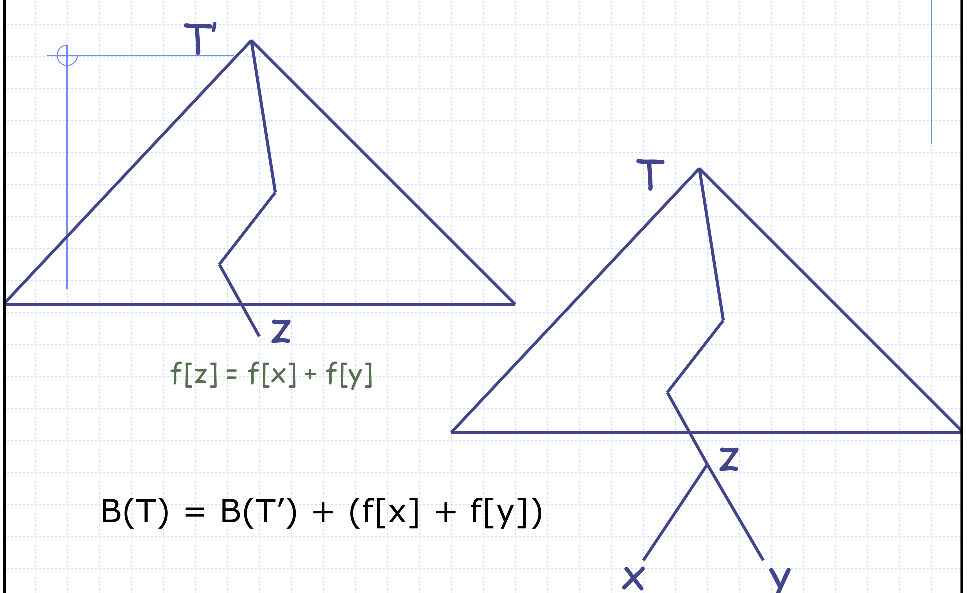
We first express B(T) in terms of B (T')

$\forall c \in C - \{x,y\}$ we have $d_T(c) = d_{T'}(c)$, and hence

$f[c]d_T(c) = f[c]d_{T'}(c)$.

43

Claim: If T′ is optimal, so is T.



B(T) = B(T′) + (f[x] + f[y])

44

Since $d_T(x) = d_T(y) = d_{T'}(z) + 1$, we have

$f[x]d_T(x) + f[y]d_T(y) = (f[x] + f[y]) (d_{T'}(z) + 1) = f(z)d_{T'}(z) + (f[x] + f[y])$

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

From which we conclude that

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

Proof of **Claim** by contradiction

Suppose that T does not represent an optimal prefix code for C.
Then there exists a tree Opt such that B(Opt) < B(T).

Without loss in generality (by **Lemma 1**) Opt has x & y as siblings. Let T'' be the tree Opt with the common parent of x & y replaced by a leaf z with frequency f[z] = f[x] + f[y].

Then, $B(T'') = B(Opt) - (f[x] – f[y])$

$< B(T) - (f[x] - f[y])$ (assume B(Opt) < B(T))

$= B(T')$

Yielding a contradiction to the assumption that T' represents an optimal prefix code for C'. Thus, T must represent an optimal prefix code for the alphabet C.

45

Theorem: Huffman Code is optimal for n characters.

Proof: Induction on n.

Base case: n = 2 and one character is 0 and the other is 1. Optimal.

Inductive hypothesis: Huffman Code is optimal for n – 1 characters.

Induction case: We have n characters in C. Let x & y be the least frequent characters in C. We replace x & y by z with f(z) = f(x) + f(y) to obtain
C' = C – { x, y } U {z}. By induction hypothesis, we have optimal code for C'. Let code(z) = c. Then let code of x be c0 and code of y be c1. By Lemma 2, the resulting code is optional for C.

46

Drawbacks

The main disadvantage of Huffman's method is that it makes two passes over the data:

- one pass to collect frequency counts of the letters in the message, followed by the construction of a Huffman tree and transmission of the tree to the receiver; and
- a second pass to encode and transmit the letters themselves, based on the static tree structure.

This causes delay when used for network communication, and in file compression applications the extra disk accesses can slow down the algorithm.

We need one-pass methods, in which letters are encoded "***on the fly***".

47

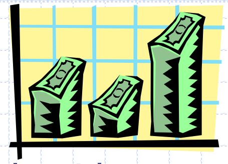
Huffman Code Summary

- Given a string X, Huffman's algorithm construct a prefix code that minimizes the weight of the encoding of X
- It runs in time $O(m + n \log n)$, where m is the length of X and n is the number of distinct characters of X
- A heap-based priority queue is used as an auxiliary structure

48

48

The Greedy Method



- **The greedy method** is a general algorithm design paradigm for optimization problems, built on the following elements:
  - **configurations**: different choices, collections, or values to find
  - **objective function**: a score assigned to configurations, which we want to either maximize or minimize
- It works best when applied to problems with the **greedy-choice** property:
  - a globally-optimal solution can always be found by a series of local improvements from a starting configuration.

49

49