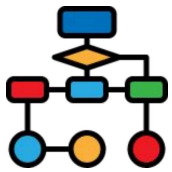
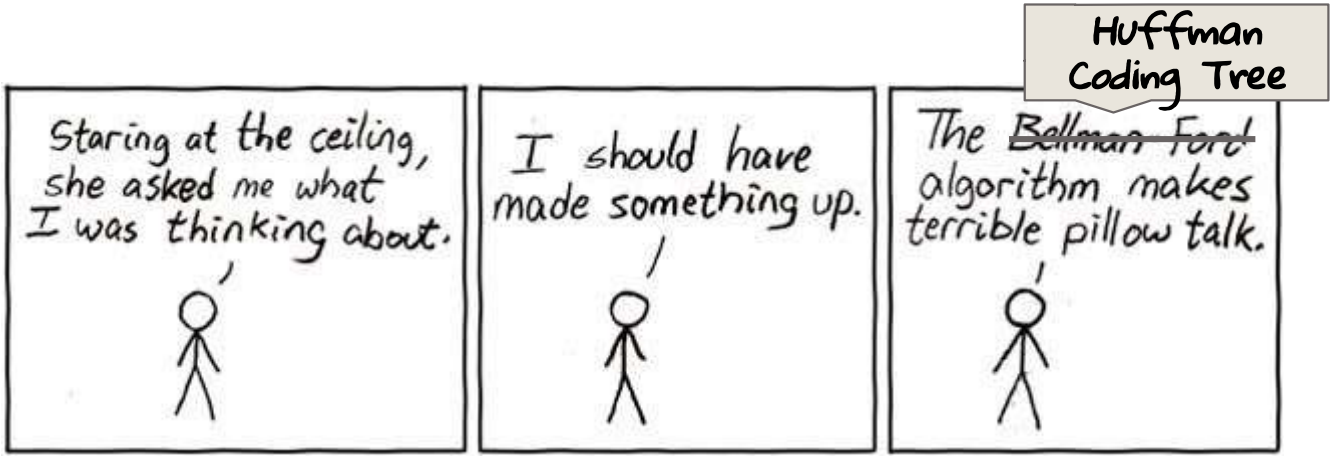


COSC 262
Algorithms

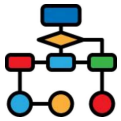
Optimization Methods Part 1

Greedy algorithms



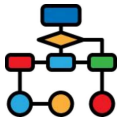
https://xkcd.com/69/

Richard Lobb and R Mukundan
Department of Computer Science and Software Engineering
University of Canterbury



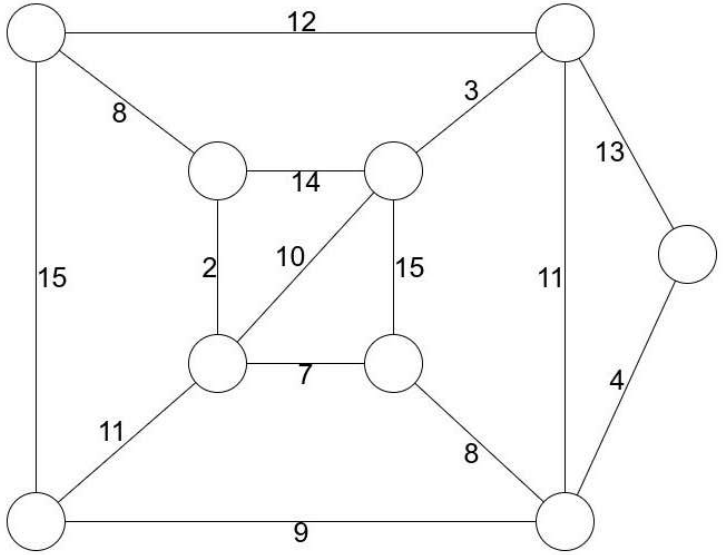
The Greedy Method ("Greedy heuristic")

- A simple *intuitive* method.
- Choose whatever option looks best at the moment
  - Hoping that a locally optimal choice will lead to a globally optimal solution.
- A decision made in one stage is not changed later.
  - No backtracking
- Greedy algorithms do not always yield optimal solutions, but for many problems, they do.
- In each iteration, greedy algorithms seek a *feasible* solution that minimizes or maximizes a given *objective* function.



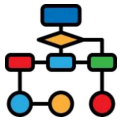
Examples you know and love already

Minimum Spanning Tree algorithms:



- Prim's algorithm
  - Maintain set of connected nodes
  - Repeatedly add "cheapest" node
    - GREEDY!
- Kruskal's algorithm
  - Maintain forest of disconnected trees
  - Repeatedly add "cheapest" edge that doesn't cause a cycle
    - GREEDY!

Both fairly easily shown to be **optimal**

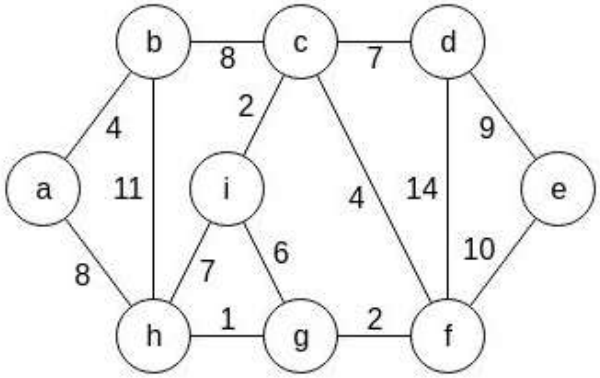


Kruskal's Algorithm

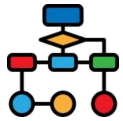
edges sorted by weight:

1:gh, 2:ci, 2:fg, 4:ab, 4:cf, 6:gi, 7:cd, 7:hi,

8:ah, 8:bc, 9:de, 10:ef, 11:bh, 14:df




```
forest = set()
for each v ∈ V:
    create new component {v}
for each (u,v) ∈ E in order of non-decreasing weight:
    if u and v are in different components:
        forest.add((u,v))
        merge components of u and v
```



The Coin Changing Problem

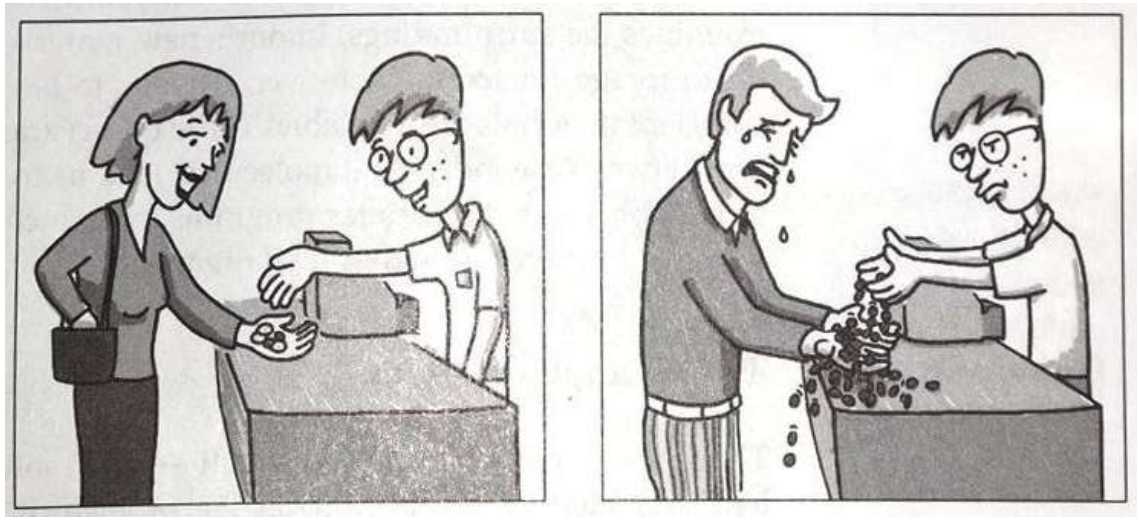
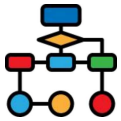


Image taken from Bioinformatics Algorithms, MIT Press, 2004.

You are a shopkeeper. You need to give change of V cents to a customer. The till has coins of denominations $C = \{c_1,..,c_m\}$. The goal is to give the customer the minimum number of coins whose total value equals the amount V.



The Coin Changing Problem

Assumptions:

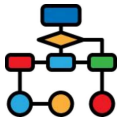
- $c_1 < c_2 < ....< c_m$ (Input sorted in ascending order)
- Each denomination is available in unlimited quantity
- $c_1 = 1$, so that a solution exists for any integer $V > 0$

Required solution:

Integers $n_1, n_2..., n_m$ such that

$V = n_1c_1 + n_2c_2 + ... + n_mc_m$

$N = n_1 + n_2 + ... + n_m$ is the minimum possible value.



The Coin Changing Problem

Greedy strategy:

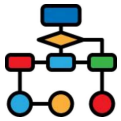
counts = defaultdict(int) # Counters of all coins

while $V > 0$:

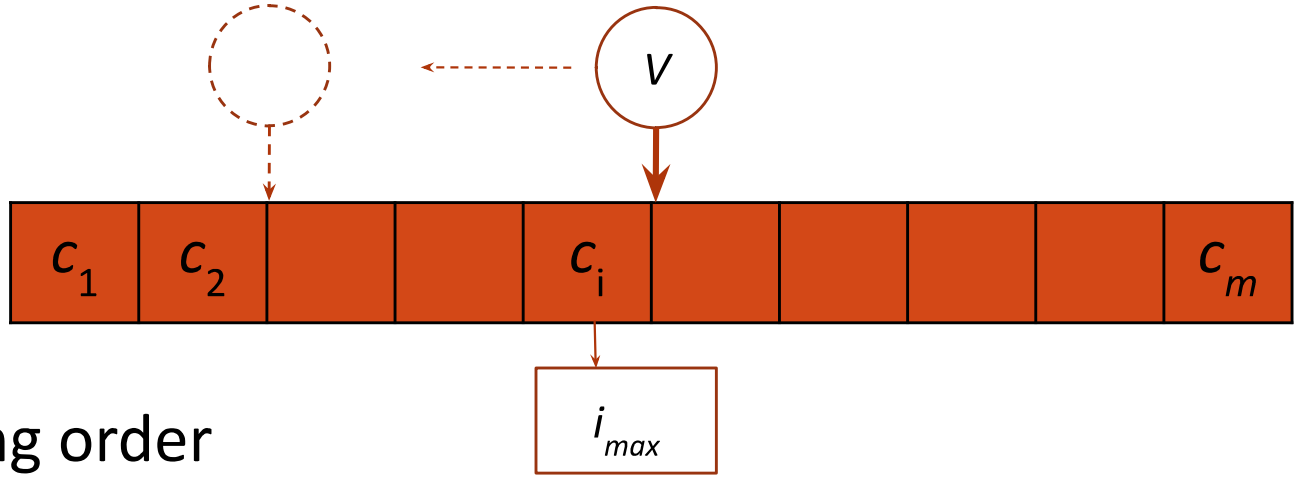
select the largest value $c_i \leq V$ (*i.e.*, $c_i \leq V < c_{i+1}$)

counts[c_i] += 1

$V = V - c_i$



The Coin Changing Problem



Sort C in ascending order

counts = defaultdict(int) # Coin counters

imax = *m*

while *V* > 0:

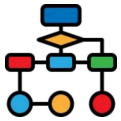
    while c_{imax} > V:

        imax = imax − 1

    counts[c_{imax}] += 1

    V = V − c_{imax}

Q: what precondition(s) must be satisfied to ensure these loops terminate?



The Coin Changing Problem

Examples:

- V = 82c C = {1c, 5c, 10c, 25c}

  Greedy Solution: **3***25c + **1***5c + **2***1c (N = 6 coins)

  Here, the solution is optimal.

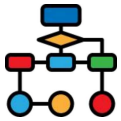
- V = 80c C = {1c, 10c, 25c}

  Greedy Solution: **3***25c + **5***1c (N = 8 coins)

  This is <u>not</u> an optimal solution.

  Optimal Solution: **2***25c + **3***10c (N = 5 coins)

Can you suggest *a sufficient property of the currency system* that ensures optimality of a greedy algorithm for all V ≥ 0?



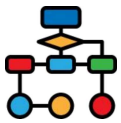
Activity-Selection Problem

How to get your money's worth out of a carnival?

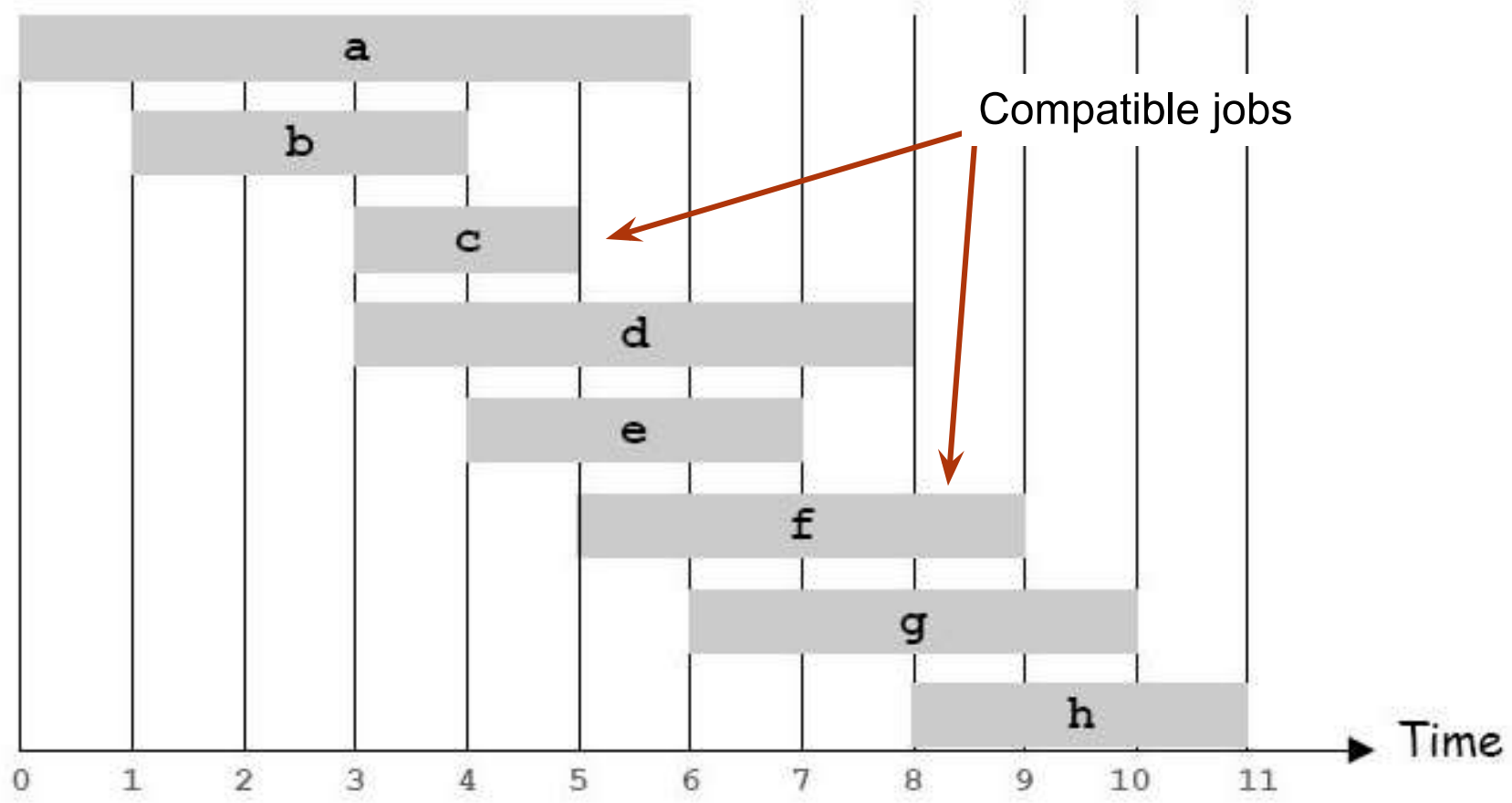
- Buy a wristband that lets you onto any ride
- Lots of rides, each starting and ending at different times
- Your goal: ride as many rides as possible

The Specific Problem: Interval scheduling

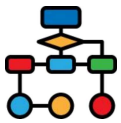
- Given a set of scheduled jobs
- Job j starts at time s_j and finishes at time f_j
- Two jobs are compatible if they don't overlap
- Your goal: find maximum subset of mutually compatible jobs



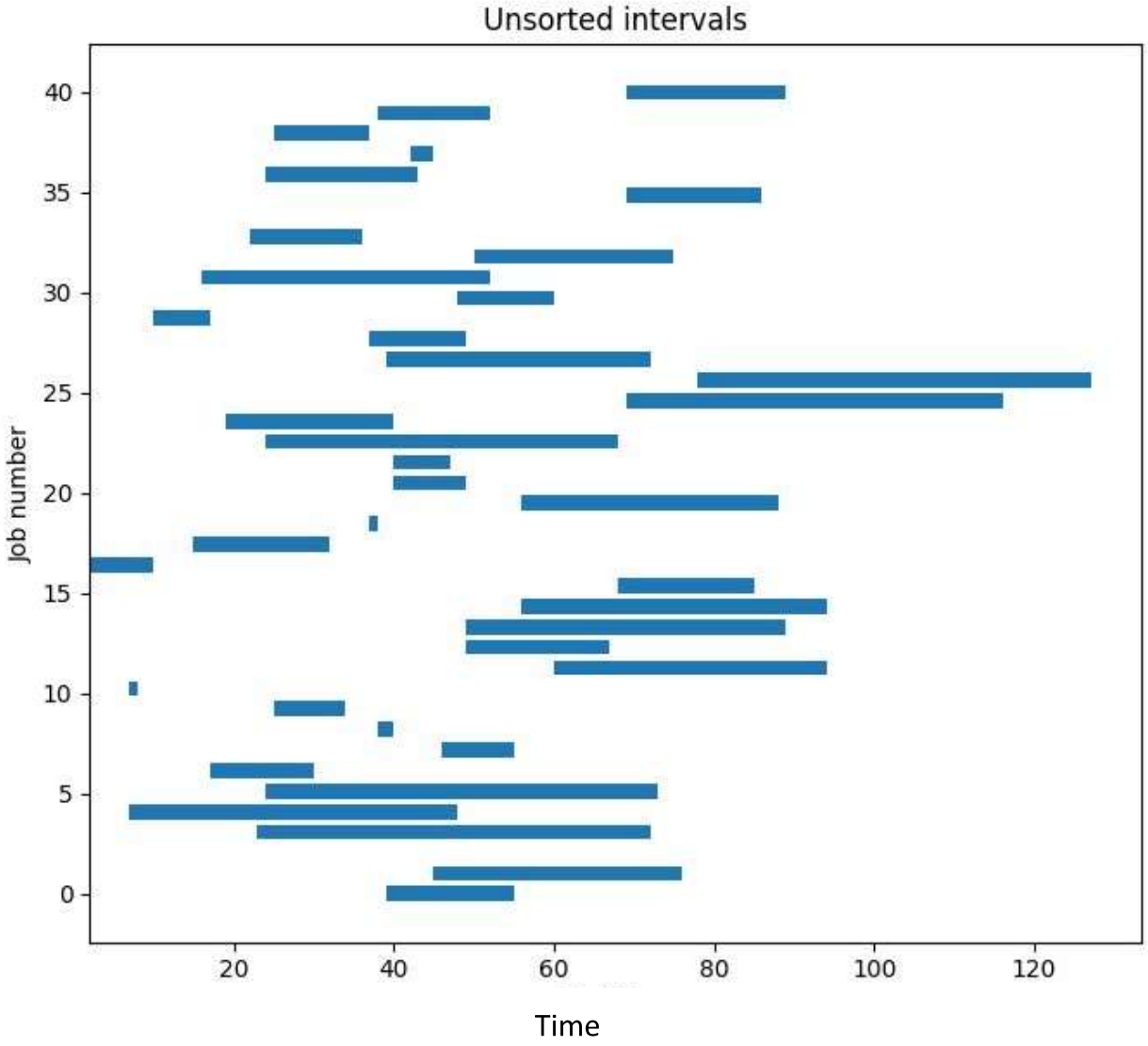
Interval Scheduling



What is the (or an) optimal solution? Answer: {b, e, h}

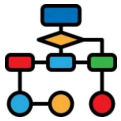


A more challenging example

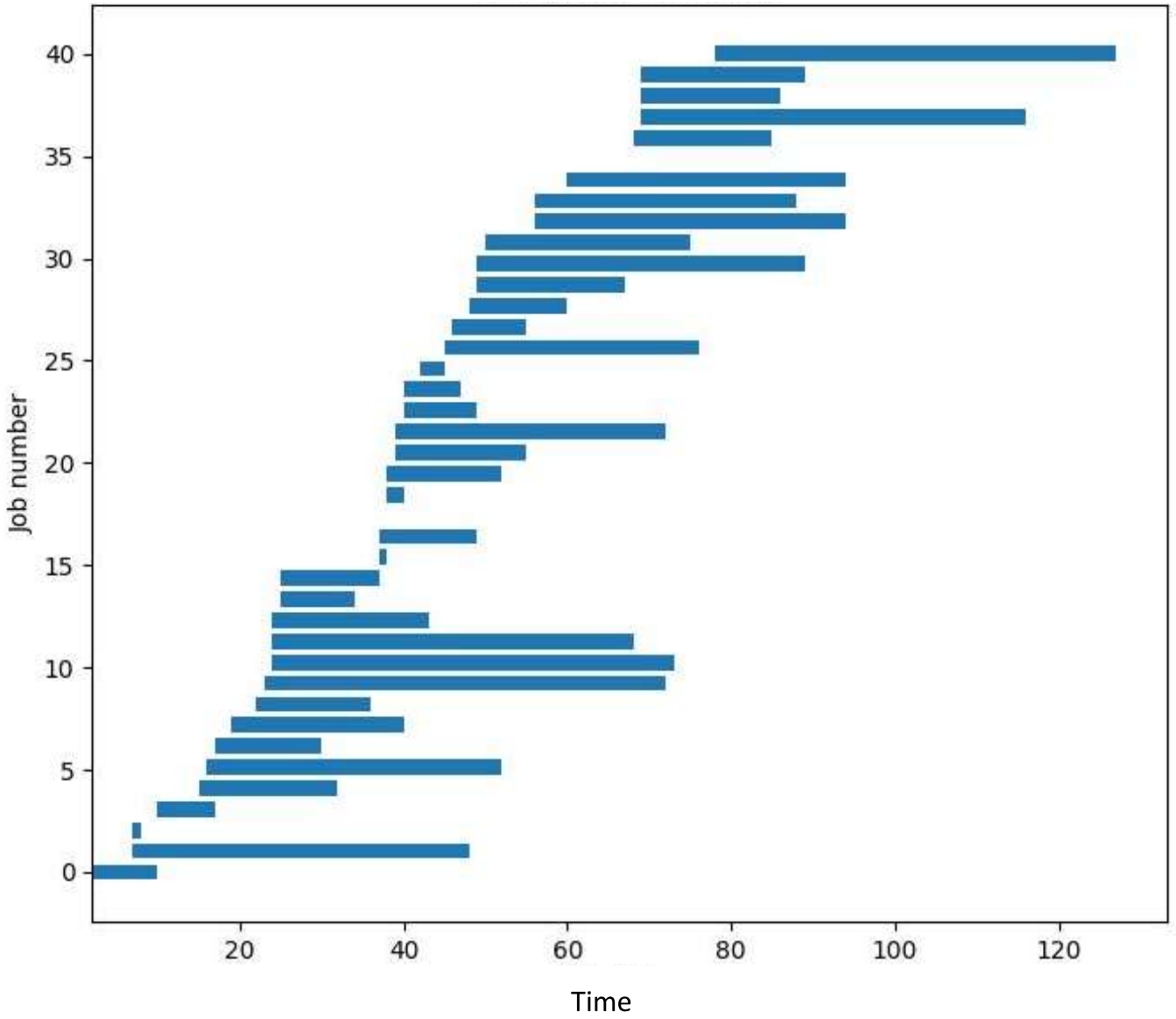


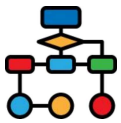
What *algorithm* will you use?

Assume *greedy heuristic* … what is the sort key?

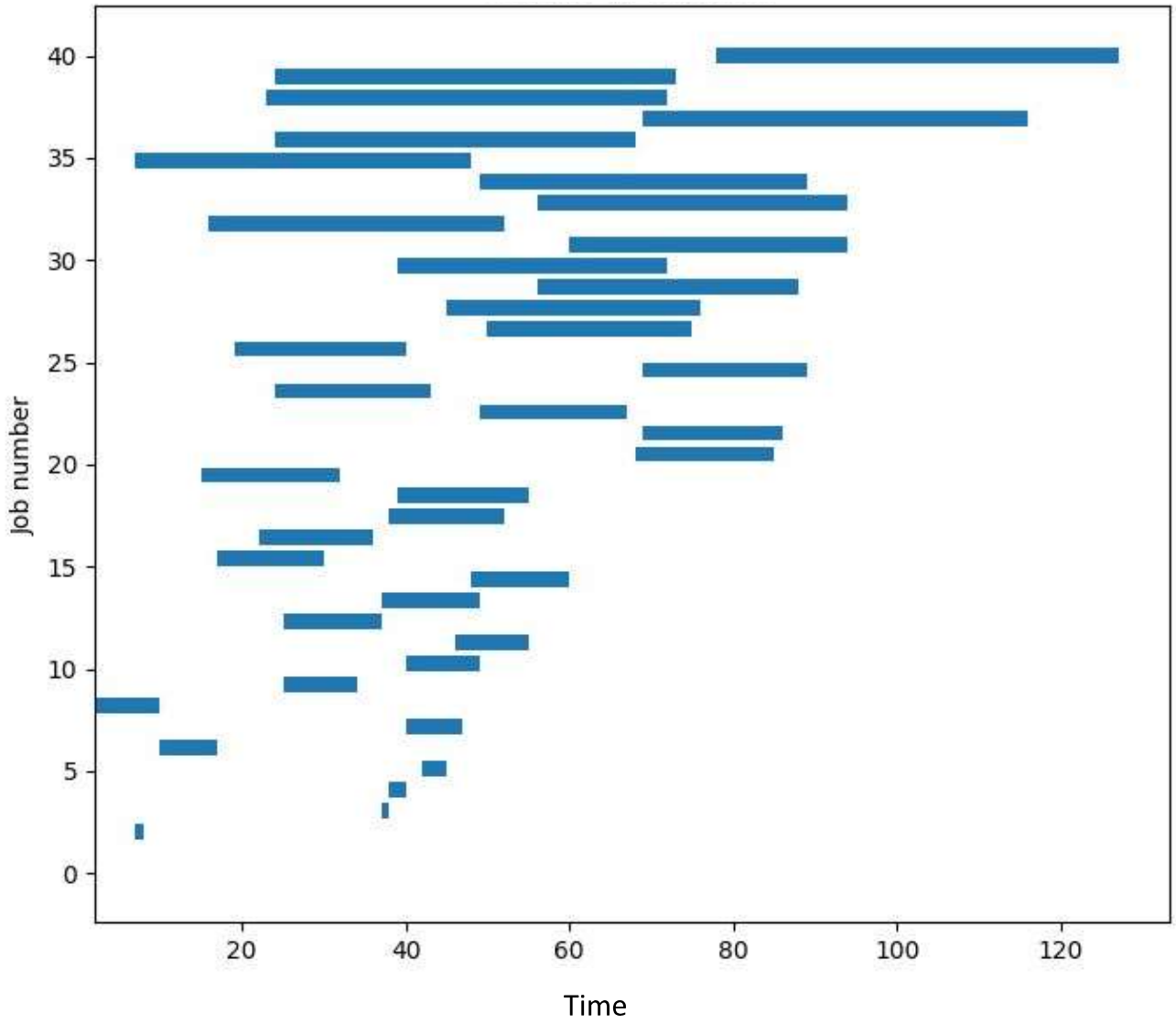


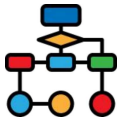
Sort by start time?



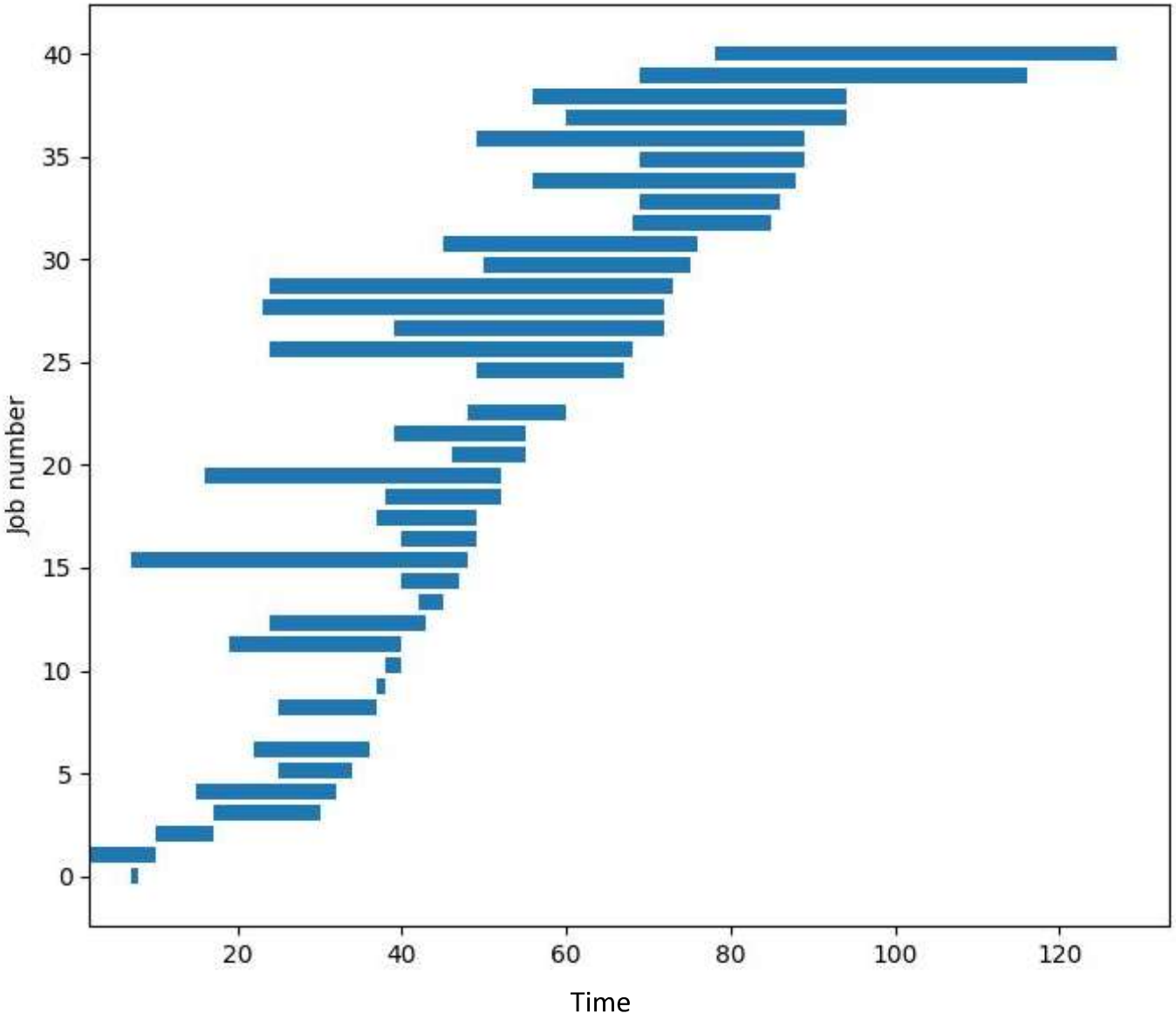


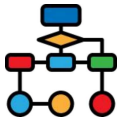
Sort by duration?





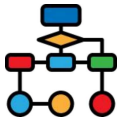
Sort by end time?





Interval Scheduling

- Have to *order* the jobs. Then take each job, provided it is compatible with the ones already taken.
- Bad choices for ordering:
  - Increasing order of start times s_j
    - "Oh no, the first job takes all day"
  - Increasing order of job intervals $f_j - s_j$
    - "Oh no, the shortest job overlaps two mutually compatible jobs"
- The right choice for ordering:
  - Increasing order of finish times f_j
  - On completion of job with finish time f_j, start next job (s_k, f_k) in the sorted order such that $s_k \geq f_j$.



Interval Scheduling Algorithm

Greedy algorithm:

Sort jobs by finish times: $f_1 \leq f_2 \leq \ldots \leq f_n$

$S = \{\}$ (Set of jobs selected)

t_current = 0 (Finish time of the last job selected)

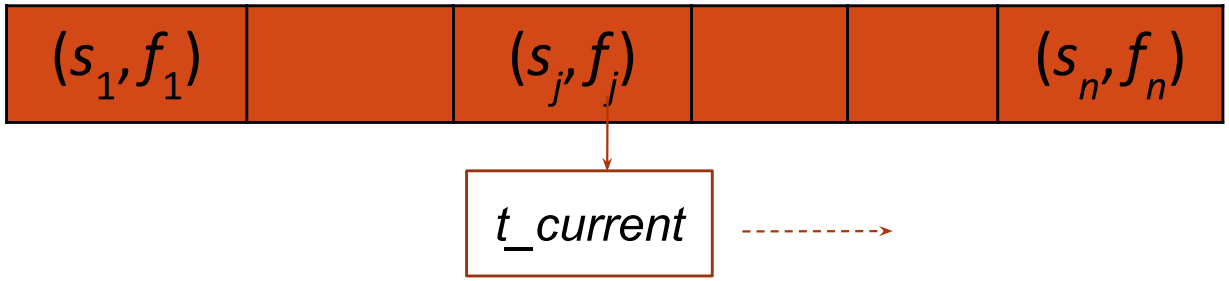
for j = 1..n:

  if $s_j \geq$ t_current :

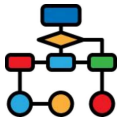
    $S = S \cup \{ j \}$

    t_current = f_j

return S



Is this always optimal? If so, prove it. If not, give a counter-example.

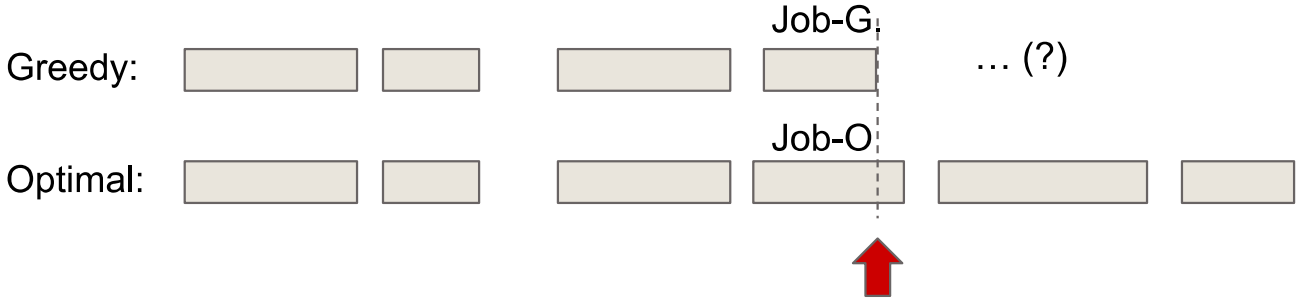


Interval Scheduling Optimality

Yes, greedy algorithm (sorting by finish time) *is* optimal.

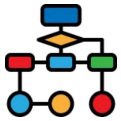
Proof by contradiction (informal):

Assume algorithm is **not** optimal. Then for some problem there exists a better solution than that found by the greedy algorithm. Consider the point at which the results first differ non-trivially (e.g. excluding simple swapping of one job for another).



Optimal algorithm has Job-O that finishes later than Job-G. But if optimal algorithm had instead used Job-G the rest of its jobs could remain unchanged and it would still be optimal. Therefore this **isn't** the point at which the results first differ non-trivially, contradicting the assumption.

More formally: see https://en.wikipedia.org/wiki/Charging_argument

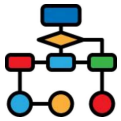


Fractional Knapsack Problem

- Given a set S of n items, such that each item i has a positive value b_i, and a positive weight w_i, determine the subset of S that yields the maximum value without exceeding a given weight W.
- We can select a fraction x_i of an item *i* giving a value $x_i b_i$ and a weight $x_i w_i$.

$$W = 9$$

i	Item	Benefit b_i	Weight w_i
1	Popcorn (1 carton)	12	4
2	Potato chips (1 bag)	15	3
3	Pizza (1)	14	2
4	Chocolate cookies (bag)	20	5



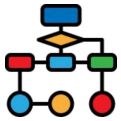
Fractional Knapsack Problem

Greedy Solution with measure b_i

i	**Item**	x_i	**Benefit** $x_i b_i$	**Weight** $x_i w_i$
4	Chocolate cookies	1	20	5
2	Potato chips	1	15	3
3	Pizza	½	7	1
	TOTAL		42	9

Greedy Solution with measure b_i / w_i

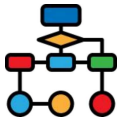
i	**Item**	x_i	**Benefit** $x_i b_i$	**Weight** $x_i w_i$
3	Pizza	1	14	2
2	Potato chips	1	15	3
4	Chocolate cookies	⅘	16	4
	TOTAL		45	9



Fractional Knapsack Problem

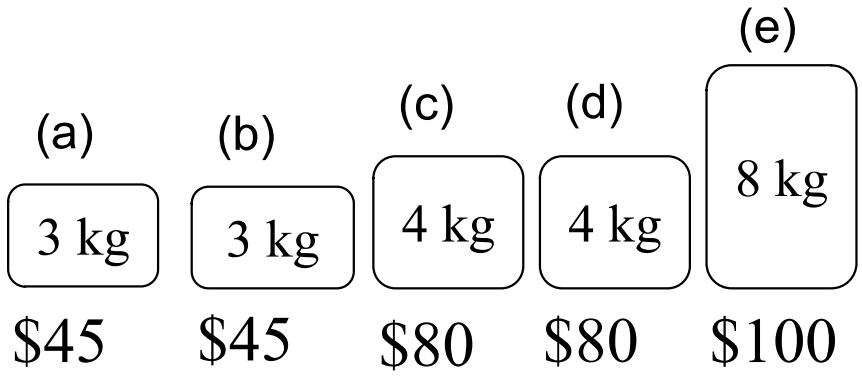
Correct greedy algorithm (optimal)

- Sort items in descending order of b_i/w_i.
- Take items (or a fraction of the final item) in order until knapsack full.
- Gives an optimal solution in $O(n \log n)$ time.



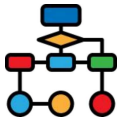
The 0/1 Knapsack
(*not* amenable to a greedy solution)

- A thief robbing a store finds n items; the i^{th} item is worth b_i dollars, and weighs w_i kgs (b_i , w_i are integers). The thief can carry at most W kgs. Which items should (s)he take for maximum payload?
- A greedy approach is in general sub-optimal.
- Example:



- Greedy solution using only b_i : (e) = $100

  or using b_i/w_i : (c) + (d) = $160
- Optimal solution: (a) + (b) + (c) = $170
- Needs dynamic programming - next week.



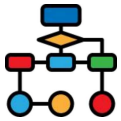
Representing text in binary

The ASCII encoding system uses a fixed-length 7-bit binary string to store each character (but using 8-bit bytes)

- ASCII = American Standard Code for Information Interchange

0100000		0110000	0	1000000	@	1010000	P	1100000	`	1110000	p
0100001	!	0110001	1	1000001	A	1010001	Q	1100001	a	1110001	q
0100010	"	0110010	2	1000010	B	1010010	R	1100010	b	1110010	r
0100011	#	0110011	3	1000011	C	1010011	S	1100011	c	1110011	s
0100100	$	0110100	4	1000100	D	1010100	T	1100100	d	1110100	t
0100101	%	0110101	5	1000101	E	1010101	U	1100101	e	1110101	u
0100110	&	0110110	6	1000110	F	1010110	V	1100110	f	1110110	v
0100111	'	0110111	7	1000111	G	1010111	W	1100111	g	1110111	w
0101000	(	0111000	8	1001000	H	1011000	X	1101000	h	1111000	x
0101001	)	0111001	9	1001001	I	1011001	Y	1101001	i	1111001	y
0101010	*	0111010	:	1001010	J	1011010	Z	1101010	j	1111010	z
0101011	+	0111011	;	1001011	K	1011011	[	1101011	k	1111011	{
0101100	,	0111100	<	1001100	L	1011100	\	1101100	l	1111100	|
0101101	-	0111101	=	1001101	M	1011101	]	1101101	m	1111101	}
0101110	.	0111110	>	1001110	N	1011110	^	1101110	n	1111110	~
0101111	/	0111111	?	1001111	O	1011111	_	1101111	o		

For a more complete table, see https://www.ascii-code.com/



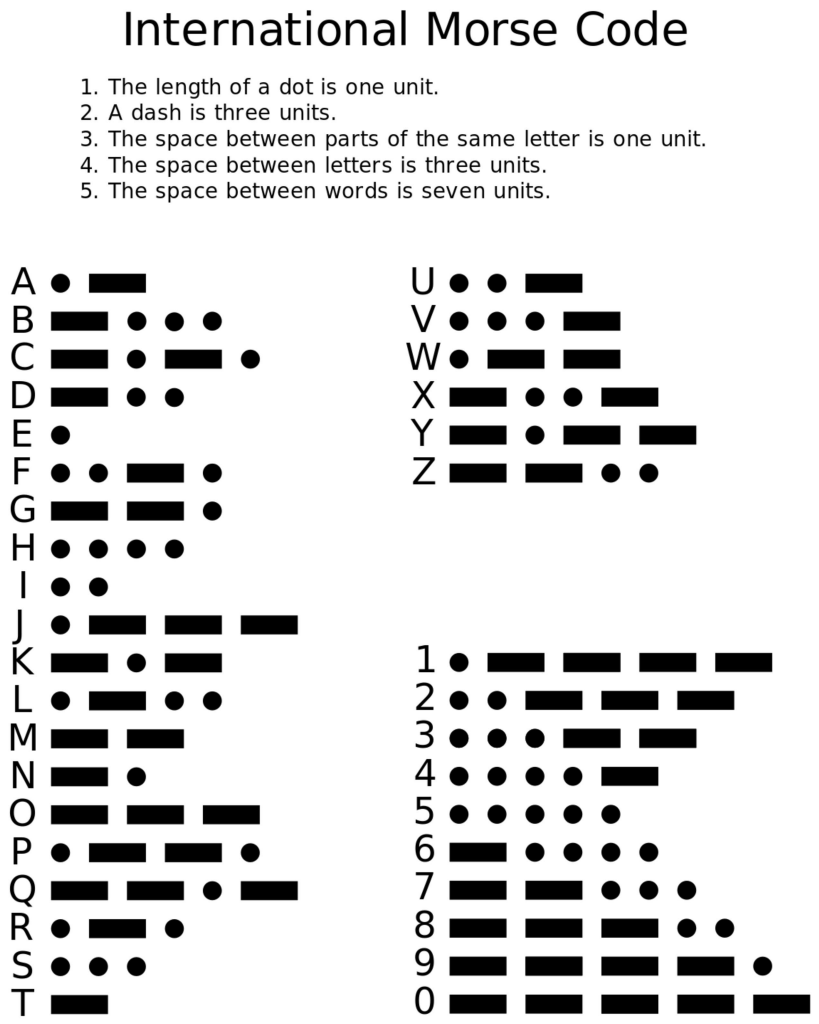
Fixed-length versus variable length codes

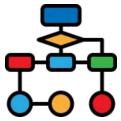
Fixed-length codes (e.g. 8-bit ASCII) are convenient:

- Easy indexing, counting etc

But variable-length codes are (potentially) more efficient

- Example 1: Morse Code
  - Most-frequent letters have short encodings
- Example 2: UTF-8
  - Most-common chars are 1-byte, less common 2, 3 and 4 bytes





An example problem

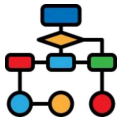
Consider the following string of characters containing only the six symbols ‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’:

acbcfccccbccadbcbcccdceadeaaaaacbcbebefbea

The character frequencies are:

Char	Frequency	No. of bits (8-bit ASCII)
‘a’ = 01100001	9	72
‘b’ = 01100010	8	64
‘c’ = 01100011	15	120
‘d’ = 01100100	3	24
‘e’ = 01100101	5	40
‘f’ = 01100110	2	16
TOTAL	42 characters	42 * 8 = 336 bits

How to encode this “document” efficiently?



Variable length binary encoding

- How might we improve that by using variable length encodings?
- Ambiguous decoding:

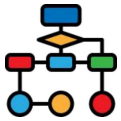
  ‘a’=0, ‘b’=1, ‘c’=01, ‘d’=10, ‘e’=11, ‘f’=001

  010 could mean either “aba”, or “ad”, or “ca”

  - We just have a stream of 0s and 1s, no spacers between them (c.f. Morse).
- A binary code allows for unambiguous decoding **if and only if no binary string in the code is a prefix of any other binary string** in the code. Example:

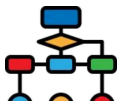
  ‘a’=000, ‘b’=0010, ‘c’=0011, ‘d’=01, ‘e’=10, ‘f’ =11

  01000100010 = “daeb”
- Called a ***binary prefix code***.

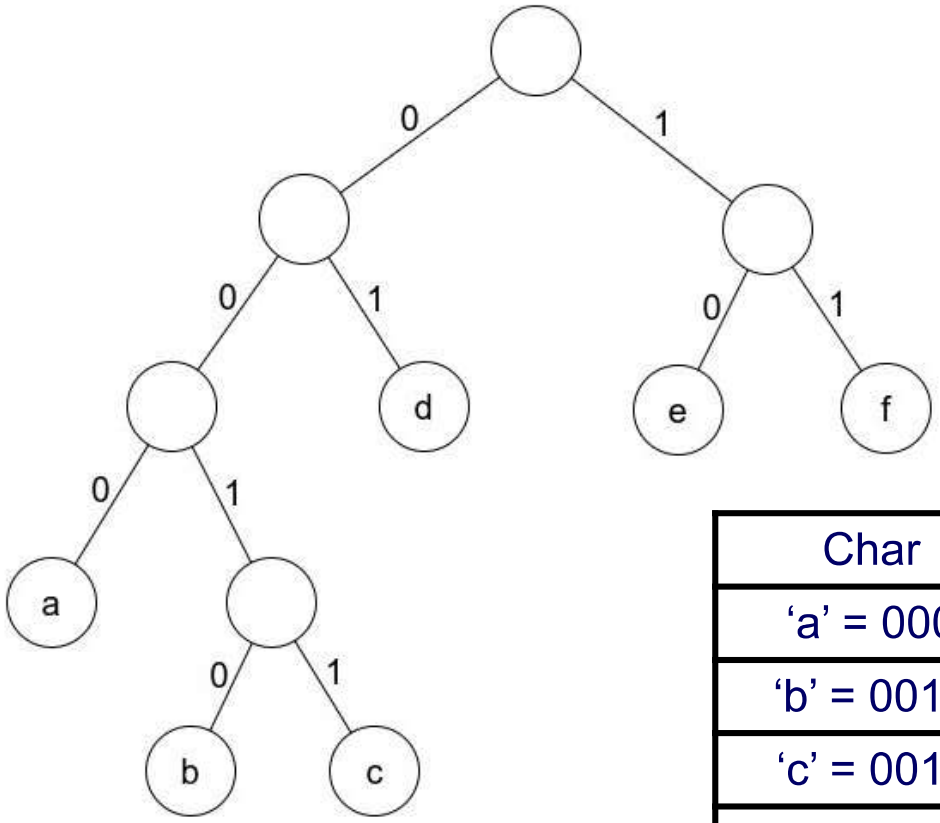


Representing Binary Prefix Codes

- Binary prefix codes are represented using 2-trees (a binary tree where every non-leaf node has exactly two children).
  - At every internal node, the left branch is labelled 0, and the right branch 1.
- Leaf nodes represent characters.
- The prefix code of a character is the edge-label sequence along the path from the root to the leaf node containing the character.

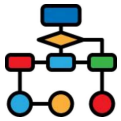


Example of a Binary Prefix Code



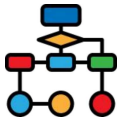
Characters are at the leaves of the tree.

Char	Frequency	No. of bits
'a' = 000	9	27
'b' = 0010	8	32
'c' = 0011	15	60
'd' = 01	3	6
'e' = 10	5	10
'f' = 11	2	4
TOTAL	**42 characters**	**139 bits**



The Encoding Problem

- Given a frequency table for characters in a document, find the/an **optimum** binary-prefix coding tree.
  - Optimum means *minimum length encoded document*.
- Clearly we require short encodings for frequent characters, longer encodings for infrequent ones.
- Example on previous slide is clearly *not* optimal.
- It's worse than a fixed-length 3-bit coding
  - 3 bits is sufficient since we only have 6 characters
  - 3 * 42 = 126 bits



Greedy, Take #1

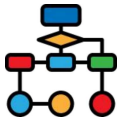
Let's greedily give the most frequent character the shortest possible code (a single 0) then allocate the next available code (10) to the next most frequent character, and so on.

Gives:

Char	Frequency	No. of bits
'c' = 0	15	15
'a' = 10	9	18
'b' = 110	8	24
'e' = 1110	5	20
'd' = 11110	3	15
'f' = 11111	2	10
TOTAL	**42 characters**	**102 bits**

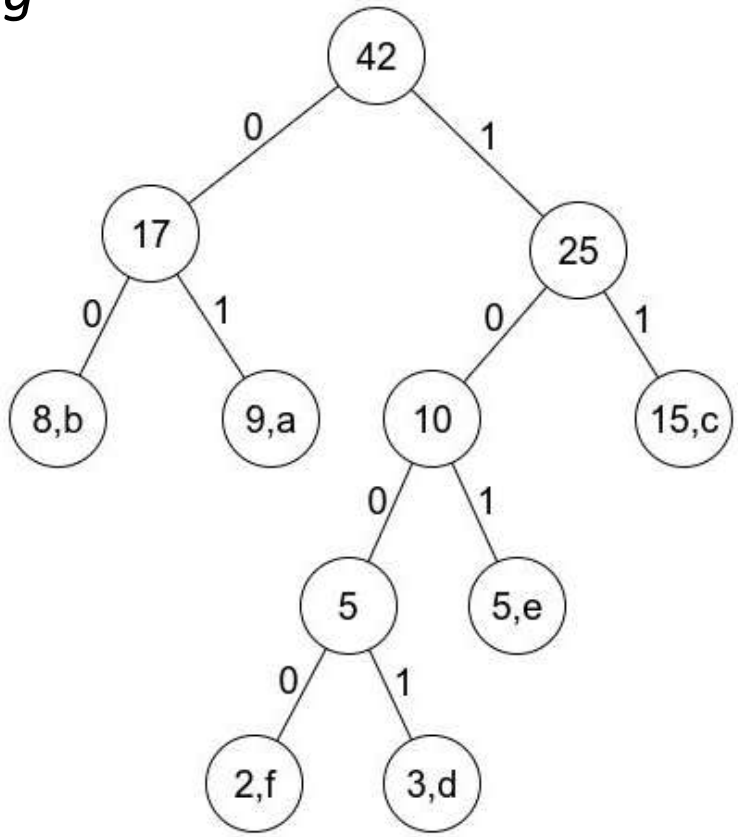
Exercise:
Draw the tree

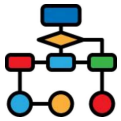
Better, but is it optimal? Can you find a failing case?



Greedy, Take #2: Huffman Coding

- Let's greedily give the least common characters the longest encodings!
  - An algorithm called *Huffman coding*
- It generates ***optimal*** binary prefix codes for a given set of frequencies.
- Solution for previous example:
  - Leaf nodes are labelled with (frequency, char)
  - Internal nodes are labelled with total frequency of all leaves in their subtree

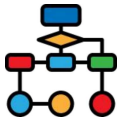




Huffman Coding algorithm

- Uses a "bottom-up" greedy approach.
- Maintains a list of partial coding trees, sorted in order of total frequency of all characters in the tree
  - Actually we don't need a fully sorted list; a min-heap will do
  - Start with each character as a trivial tree (just a leaf)
- Repeatedly combine the two trees of lowest frequency

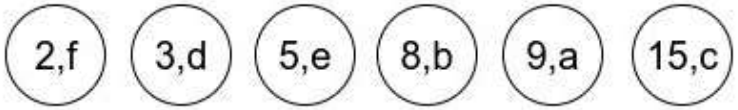
```
trees = MinHeap()
for each character in document:
    trees.insert(Leaf(frequency, character))
while length(trees) > 1:
    left = trees.pop_min()
    right = trees.pop_min()
    trees.insert(Node(left.freq + right.freq, left, right))
```



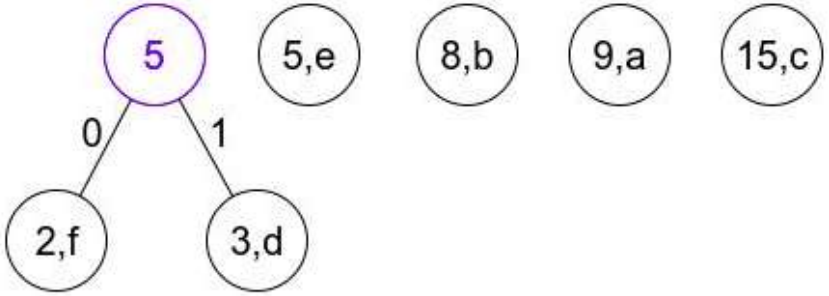
Huffman Coding - Example

Increasing frequency

Stage 1:

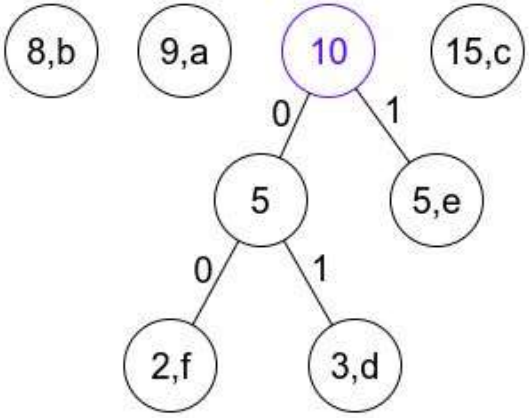


Stage 2:

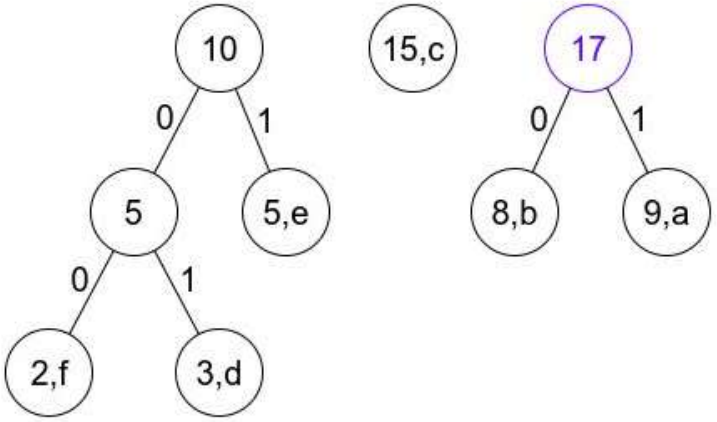


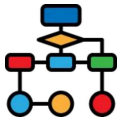
Char	Freq
'a'	9
'b'	8
'c'	15
'd'	3
'e'	5
'f'	2

Stage 3:



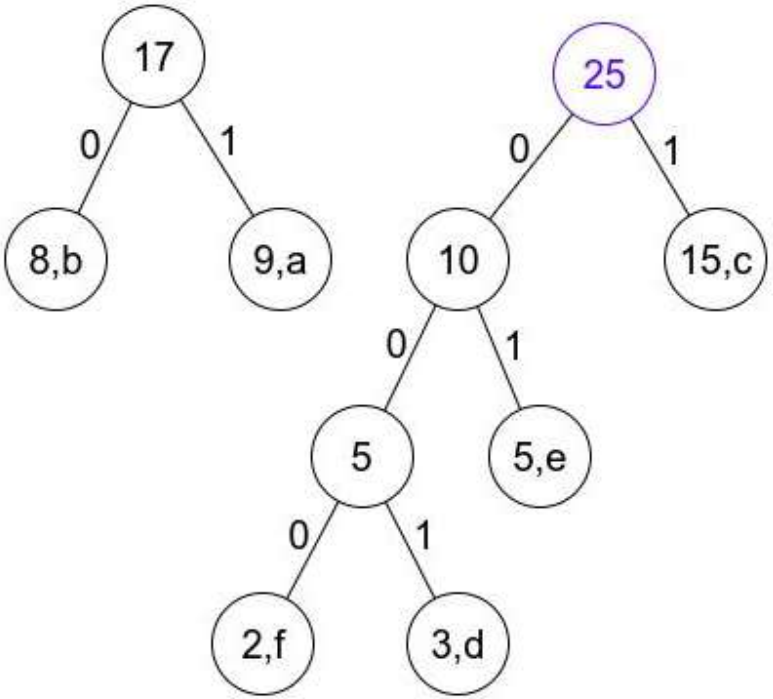
Stage 4:



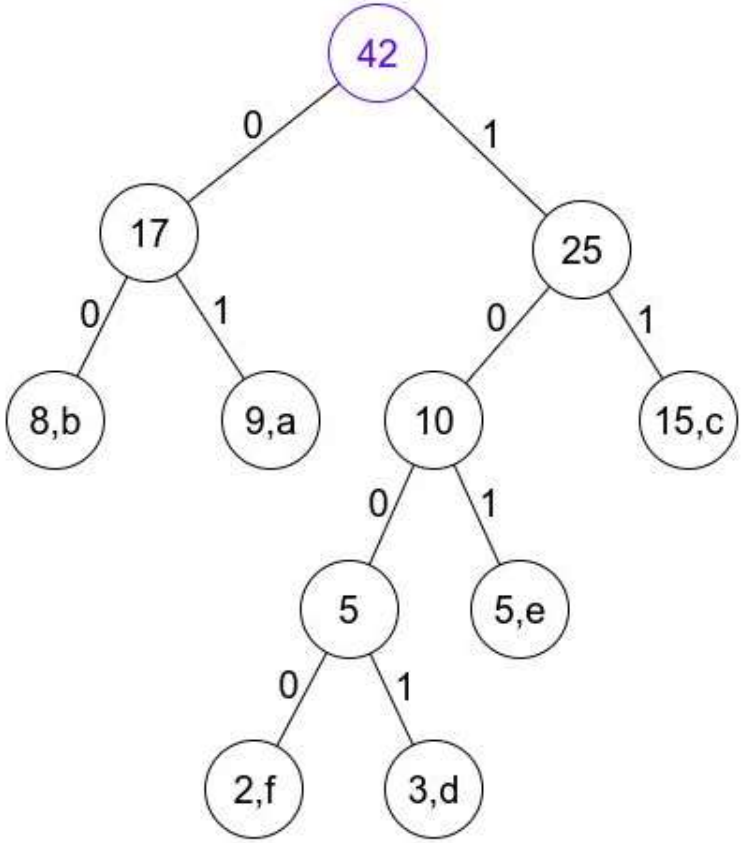


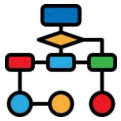
Huffman Code - Example (cont'd)

Stage 5:



Stage 6:

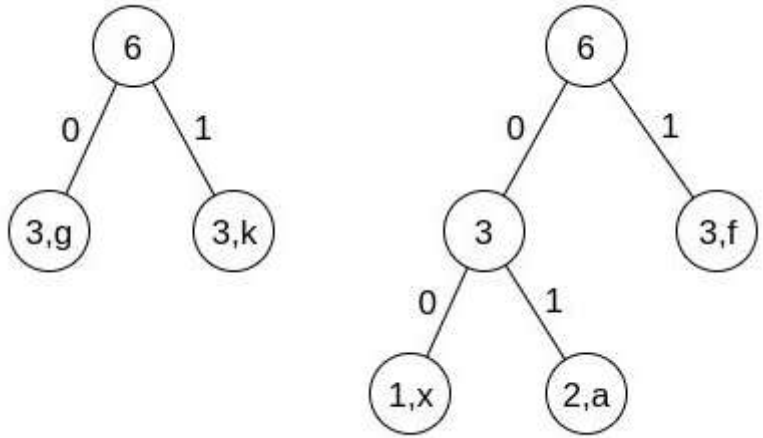


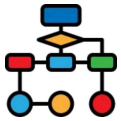


Ordering trees when frequencies are equal

To ensure a unique solution in lab exercises we require that if multiple trees have the same frequency, they are ordered by the alphabetically smallest leaf node character they contain.

For example: in the image below, both trees have a frequency of 6 but the tree on the right, which has the character 'a' as it smallest leaf, should be sorted to come *before* the tree on the left, which has 'g' as its smallest leaf.

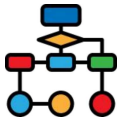




Huffman Code - Example

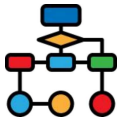
Char	Frequency	No. of bits
‘a’ = 01	9	18
‘b’ = 00	8	16
‘c’ = 11	15	30
‘d’ = 1001	3	12
‘e’ = 101	5	15
‘f’ = 1000	2	8
TOTAL	**42 characters**	**99 bits**

- Requires 99/42 ≈ 2.36 bits/character
- This is an optimal result (as always, with Huffman coding)
  - For proof see www.cs.utoronto.ca/~brudno/csc373w09/huffman.pdf



Huffman Coding: complexity

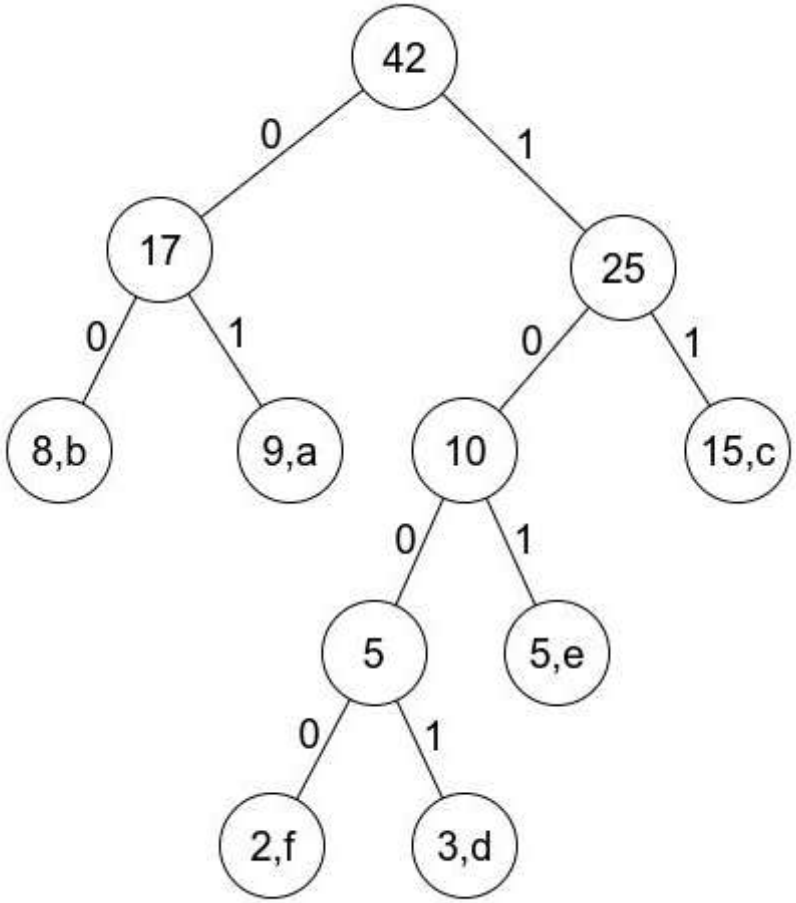
- The *trees* list must support the priority queue operations 'insert', and 'pop_min'
  - Each is $\mathrm{O}(\log n)$ using a min-heap.
- At each iteration, two elements are removed from the priority queue, and the combined tree's root is added back to the list. The size reduces by one, and hence the total number of iterations is $\mathrm{O}(n)$.
- Therefore the total time complexity is $\mathrm{O}(n \log n)$.

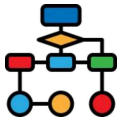


Decoding Huffman Codes

- To decode a stream of bits, start at the root of the encoding tree, and follow a left-branch for a 0, a right branch for a 1.
- When you reach a leaf, write the character stored at the leaf, and start again at the top of the tree.
- E.g. 0110011100

  = adcb

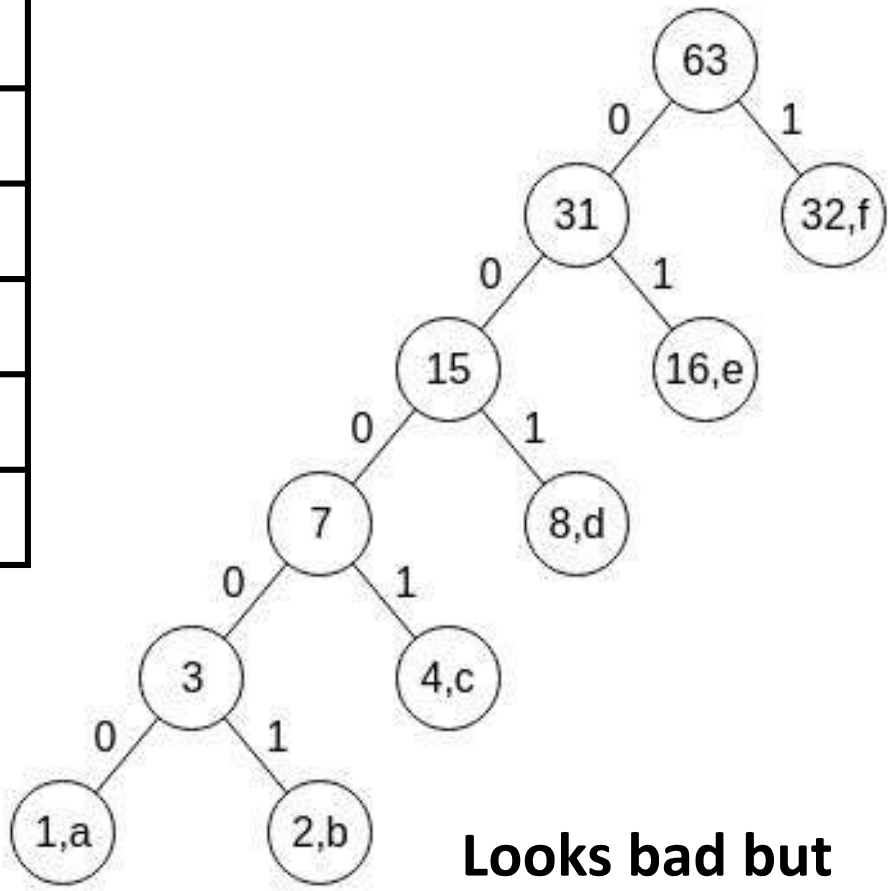




A Skewed Encoding Tree

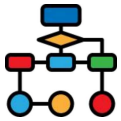
Certain frequency tables can generate unbalanced binary trees

Char	Freq
‘a’	1
‘b’	2
‘c’	4
‘d’	8
‘e’	16
‘f’	32



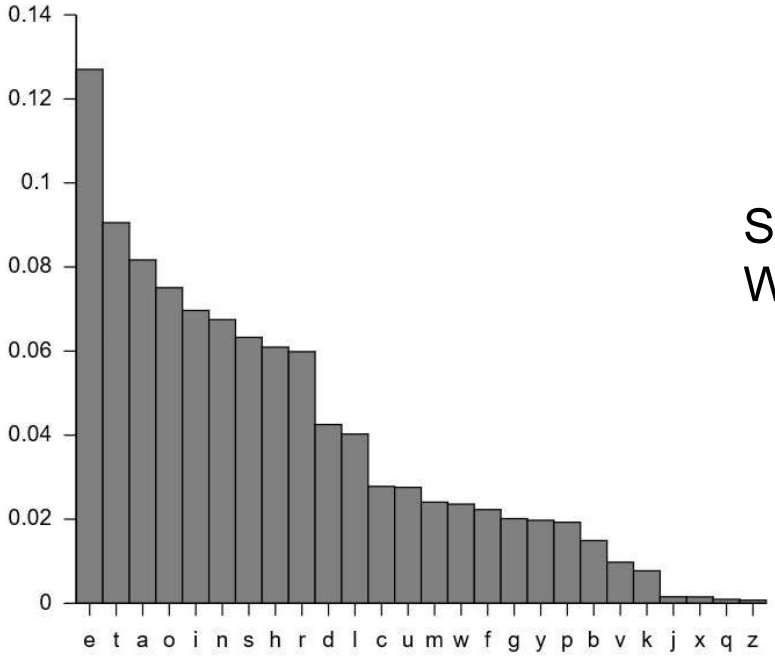
Looks bad but still optimal.

Char	Freq	Enc Bits	Total Bits
‘a’	1	5	5
‘b’	2	5	10
‘c’	4	4	16
‘d’	8	3	24
‘e’	16	2	32
‘f’	32	1	32
Total no. of bits			119
Avg. no. of bits per char			1.8

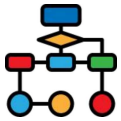


Caveat

- The encoding information must be transmitted along with the text to the decoding system.
  - An added cost
- OR: the frequency information for generic English text (say) may be used, e.g.



Source:
Wikipedia - “Letter frequency”



Advantages/disadvantages

Huffman coding gives optimal encoding based on *global* character frequencies only, but:

- Whole document must be processed first.
- Can't do local optimisation (e.g. change in char frequencies half way through a long document).
- Encoding table must itself be encoded into the document.
- Can't recognise *patterns*, e.g. common words.

Lempel-Ziv algorithm (and derivatives like LZW) generally give better compression by using a dynamic compression table with common substring recognition.

Deflate algorithm (https://en.wikipedia.org/wiki/DEFLATE) combines LZ and Huffman coding.

- Used by zip, pkzip, gzip, png, etc