Greedy algorithms

Greedy algorithms

A general algorithm construction principle ("paradigm") for combinatorial optimization problems.

The idea is simple:

- Build the solution piece by piece by always choosing what currently looks like the "best choice" (without thinking about the rest of the solution).

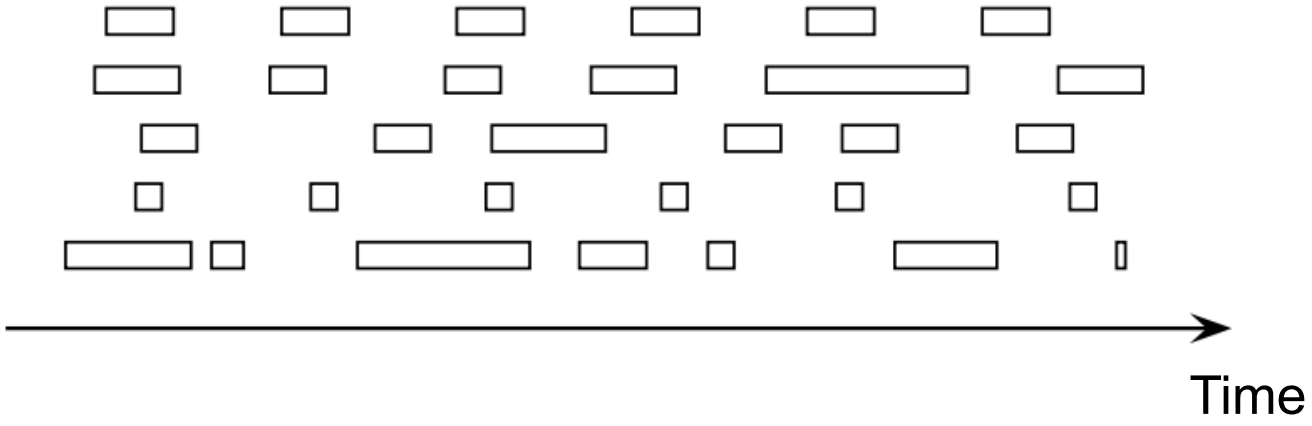
That is, one hopes that local optimization leads to global optimization.

More precisely, the method requires a definition of “best choice” (also called “greedy choice”), as well as a proof that repeated use of this choice ends in an optimal solution.

Example: a simple scheduling problem

Input: A collection of booking requests for a resource, each with a start time and end time.

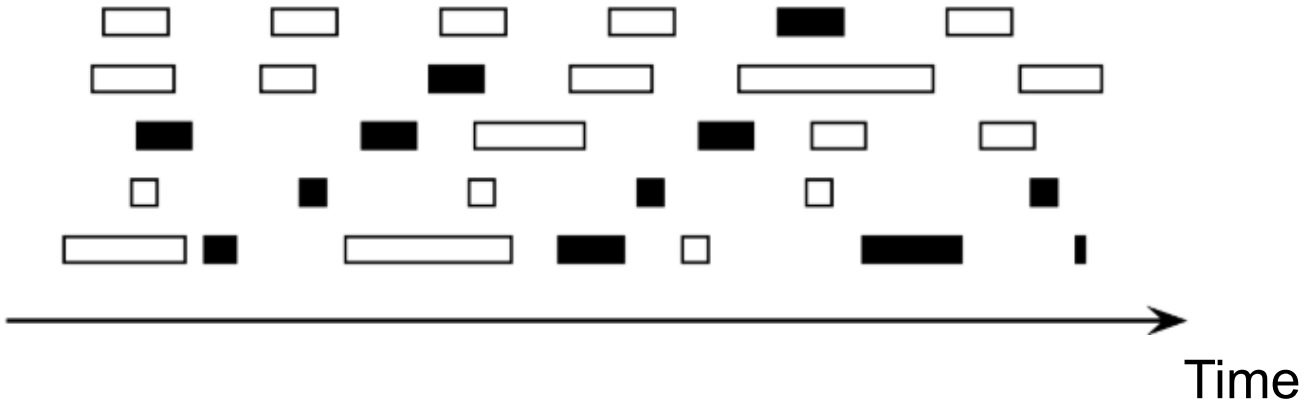
Output: The largest possible set of non-overlapping bookings.



Example: a simple scheduling problem

Input: A collection of booking requests for a resource, each with a start time and end time.

Output: The largest possible set of non-overlapping bookings.



Here are 12 non-overlapping booking requests.

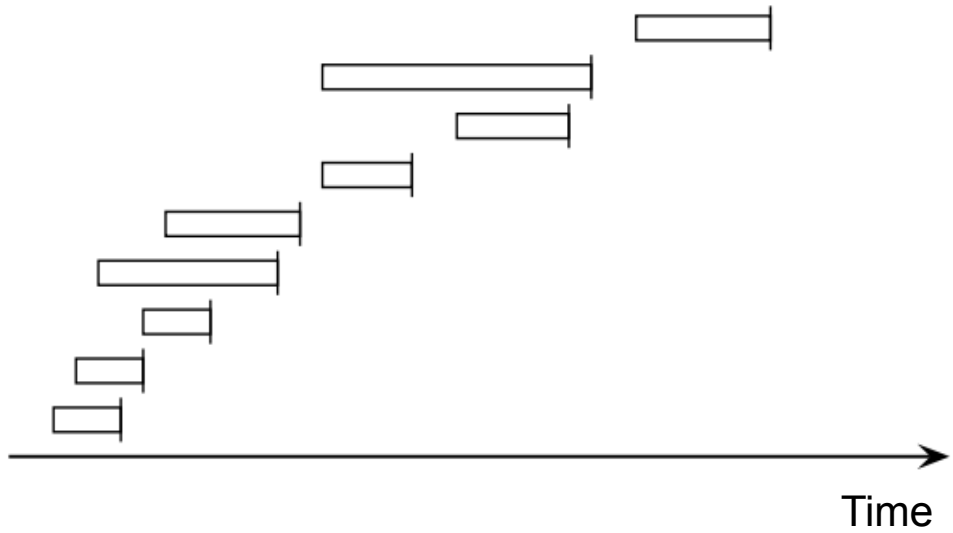
Is 12 the maximum number for this input?

Yes, but it may not be easy to see. An algorithm is needed to find the maximum number. We will try the greedy method.

Example: a simple scheduling problem

Proposed greedy choice: Choose the one that finishes earliest (among those remaining that do not overlap with already selected ones). As code:

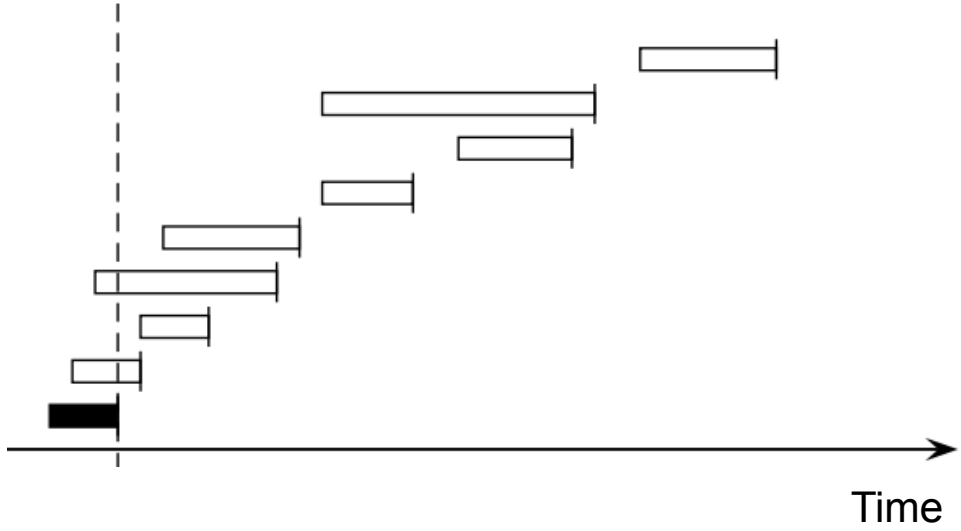
Sort activities by increasing end time
For each activity a in that order:
  If a overlaps with already selected activities:
    Skip a
  Else
    Select a



Example: a simple scheduling problem

Proposed greedy choice: Choose the one that finishes earliest (among those remaining that do not overlap with already selected ones). As code:

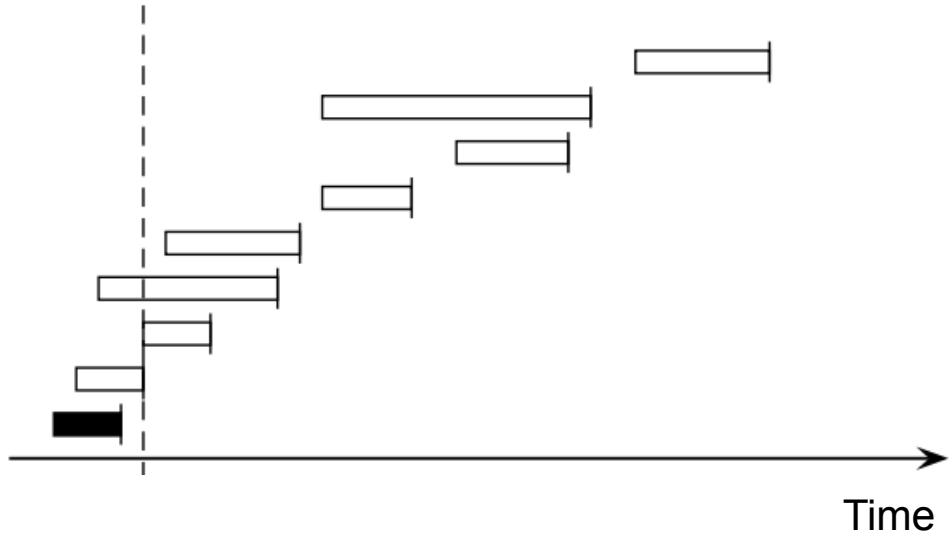
Sort activities by increasing end time
For each activity a in that order:
  If a overlaps with already selected activities:
    Skip a
  Else
    Select a



Example: a simple scheduling problem

Proposed greedy choice: Choose the one that finishes earliest (among those remaining that do not overlap with already selected ones). As code:

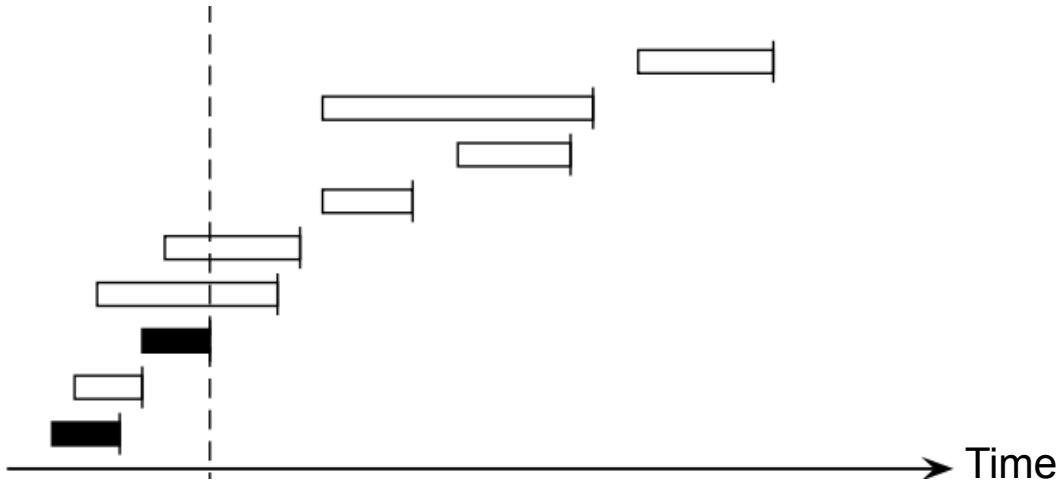
Sort activities by increasing end time
For each activity a in that order:
  If a overlaps with already selected activities:
    Skip a
  Else
    Select a



Example: a simple scheduling problem

Proposed greedy choice: Choose the one that finishes earliest (among those remaining that do not overlap with already selected ones). As code:

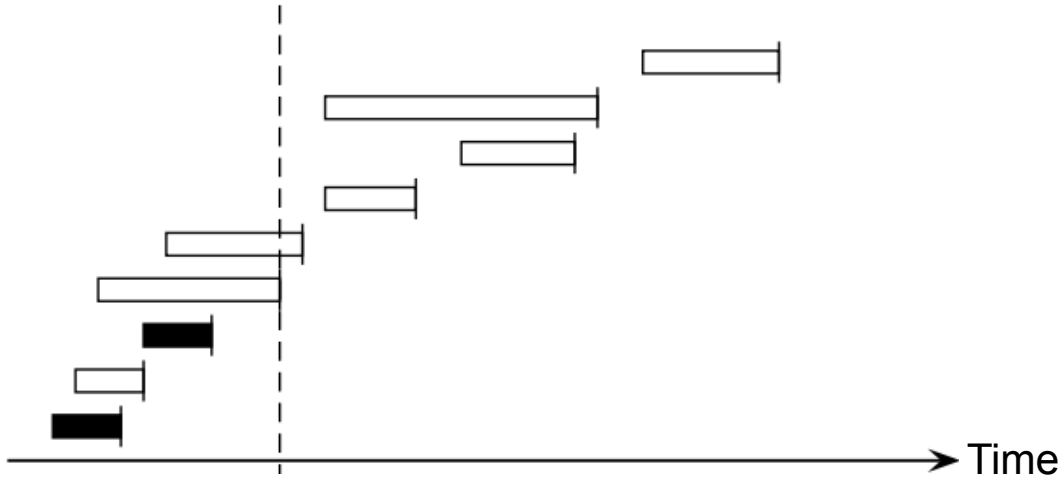
Sort activities by increasing end time
For each activity a in that order:
  If a overlaps with already selected activities:
    Skip a
  Else
    Select a



Example: a simple scheduling problem

Proposed greedy choice: Choose the one that finishes earliest (among those remaining that do not overlap with already selected ones). As code:

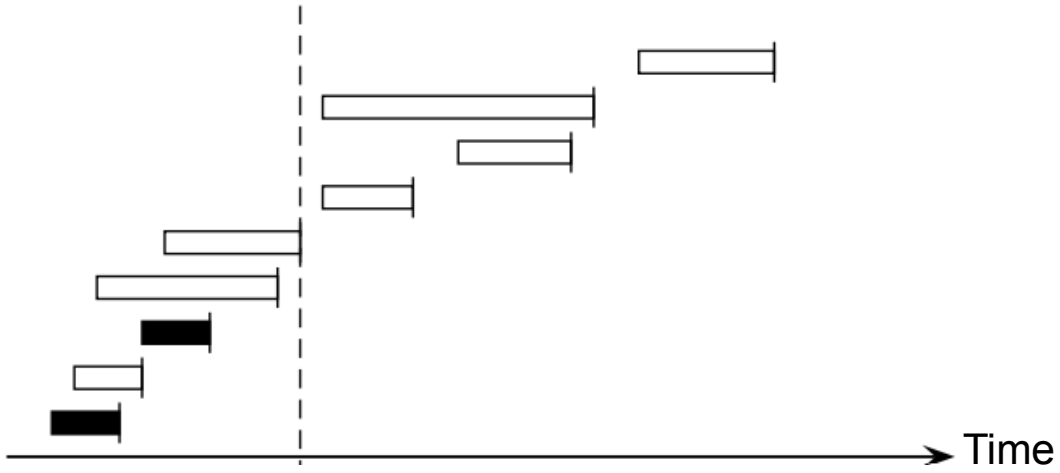
Sort activities by increasing end time
For each activity a in that order:
  If a overlaps with already selected activities:
    Skip a
  Else
    Select a



Example: a simple scheduling problem

Proposed greedy choice: Choose the one that finishes earliest (among those remaining that do not overlap with already selected ones). As code:

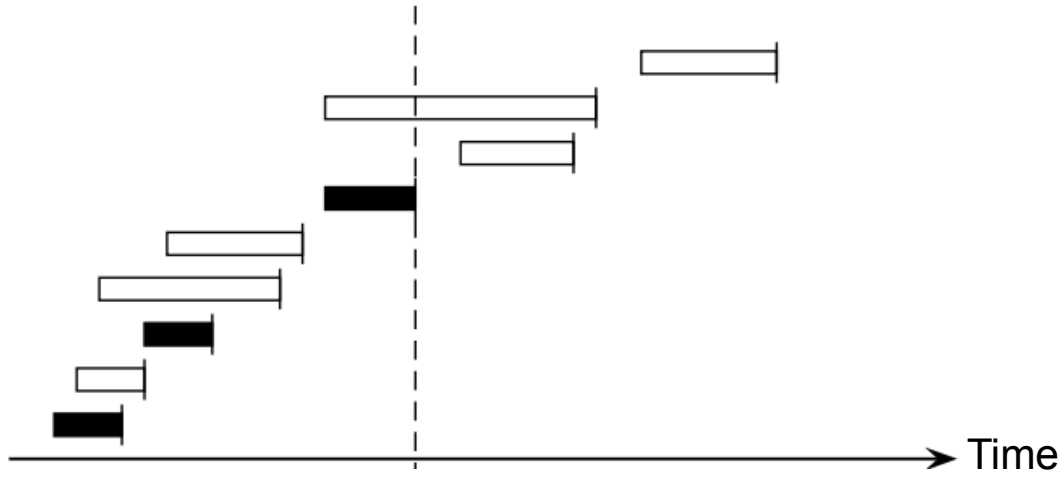
Sort activities by increasing end time
For each activity a in that order:
  If a overlaps with already selected activities:
    Skip a
  Else
    Select a



Example: a simple scheduling problem

Proposed greedy choice: Choose the one that finishes earliest (among those remaining that do not overlap with already selected ones). As code:

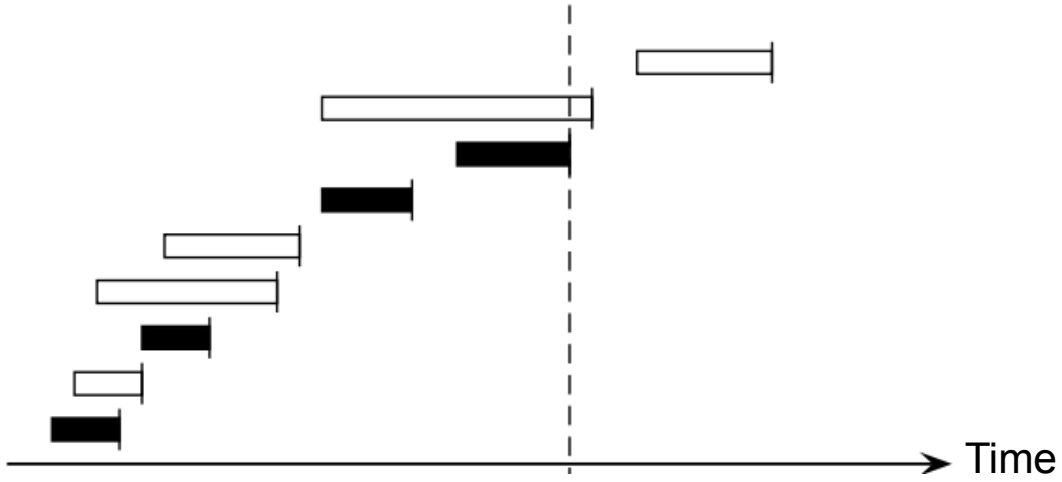
Sort activities by increasing end time
For each activity a in that order:
  If a overlaps with already selected activities:
    Skip a
  Else
    Select a



Example: a simple scheduling problem

Proposed greedy choice: Choose the one that finishes earliest (among those remaining that do not overlap with already selected ones). As code:

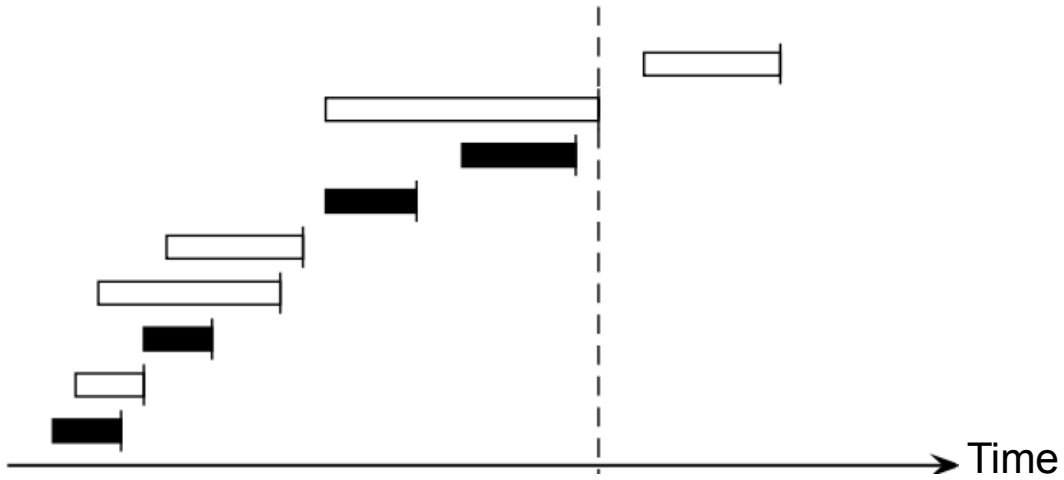
Sort activities by increasing end time
For each activity a in that order:
  If a overlaps with already selected activities:
    Skip a
  Else
    Select a



Example: a simple scheduling problem

Proposed greedy choice: Choose the one that finishes earliest (among those remaining that do not overlap with already selected ones). As code:

Sort activities by increasing end time
For each activity a in that order:
  If a overlaps with already selected activities:
    Skip a
  Else
    Select a



Example: a simple scheduling problem

Proposed greedy choice: Choose the one that finishes earliest (among those remaining that do not overlap with already selected ones). As code:

Sort activities by increasing end time
For each activity a in that order:
  If a overlaps with already selected activities:
    Skip a
  Else
    Select a

Analysis

We want to show the following invariant:

> An optimal solution OPT exists that includes
> the activities selected so far by the algorithm.

When the algorithm finishes, the correctness follows from the above invariant:

The algorithm’s selected activities are part of an optimal solution OPT. Due to how the algorithm works, all non-selected activities will overlap with one of the selected ones.Thus, the algorithm’s solution cannot be extended and it is still a valid solution.In particular, OPT cannot be larger than the algorithm's selected solution, which means the algorithm’s solution is OPT.

Analysis

Proof of invariance by induction:

Basis:
Clearly before the first iteration of the **for-**loop (since no activities have been selected).

Step:
Let OPT be the optimal solution from the induction hypothesis before the iteration. We need to show that there exists an optimal solution OPT′ in the induction hypothesis after the iteration.

If-case:
Here, the algorithm selects nothing new, so OPT can be used as OPT′.

Analysis

Else-case: OPT = optimal

Look at the activities sorted by increasing end time: a_1, a_2, ..., a_n. Let a_i be the most recently selected activity by the algorithm, and let a_j be the activity selected in this iteration.

Because of the invariant, OPT contains a_i. In OPT, a_i cannot be the last (otherwise, OPT could be extended with a_j). Let a_k be the next activity in OPT after a_i. Since a_j is the first activity (in the above sorting) after a_i that does not overlap with a_i, it must be that $j \leq k$.

If $j = k$, OPT can be used as OPT'. Otherwise, replace a_k with a_j in OPT. This does not cause overlap with other activities in OPT (they either end before a_i or end after a_k — in the latter case, they must start after a_k and therefore after a_j), and it maintains the size. So, after the replacement, we have a new optimal solution OPT' that satisfies the invariant.

[NB: In the first iteration, a_i doesn't exist, but we can set $j = 1$ and make a similar argument.]

Runtime: Sorting + O(n)

The knapsack problem

A knapsack that can carry W kg.

Items with value and weight.

Item no. i	1	2	3	4	5	6	7
Weight w_i	4	6	2	15	7	4	5
Value v_i	45	32	12	50	23	9	15

sort it in desecending order after value

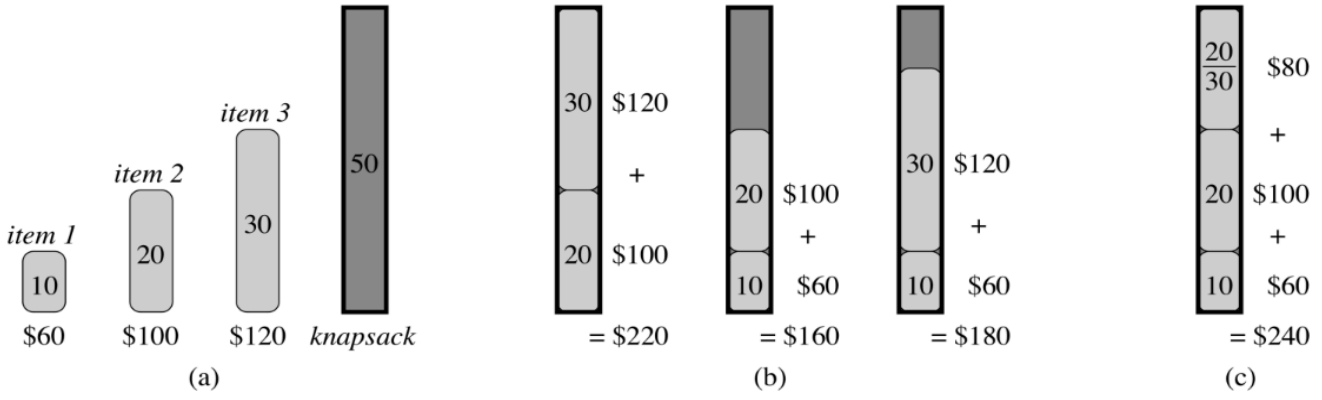
Goal: Take as much total value as possible without exceeding the weight limit.

The knapsack problem

The fractional version of the problem (where parts of items can be included in the knapsack) can be solved with a greedy algorithm: choose items in decreasing order of "utility" = value/weight.

A simple exchange argument shows that the optimal solution can only be as the one chosen by the algorithm.

Note: This greedy algorithm does NOT work for the 0-1 version of the problem (where only whole items can be included):



fractional, then greedy is best

This is an example showing that greedy choices cannot simply be assumed to work for all problems (local optimization does not always lead to global optimization).

Bit patterns

`01101011 0001100101011011...`

Bit patterns must be interpreted to have meaning:

- Letters
- Numbers (integers, decimals)
- Computer instructions (programs)
- Pixels (image files)
- Amplitude (audio files)
-

...

Today's focus: Letters (and other characters)

Representation of characters

A classic representation: ASCII

.
a: 1100001
b: 1100010
c: 1100011
d: 1100100
.

All characters occupy 7 bits (fixed-width codes)

Huffman codes

Is fixed-width coding the shortest possible representation of a file of characters?

That depends on the file's content! Example:

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

The smallest fixed-length code is 2^3 = 8 < 6.
so each sequence must be of 3 bits

Fixed-width version: bit length * frequenceny

3. (45.000+ 13.000+ …..+ 5.000) = 300.000 bits

Variable-width version:

1. 45.000+ 3 . 13.000+ …..+ 4 . 5.000 = 224.000 bits

So the variable-width has the shortes length

Goal: the shortest (possible) representation of a file. Saves disk space and time when transmitting over a network.

Prefix code= trees

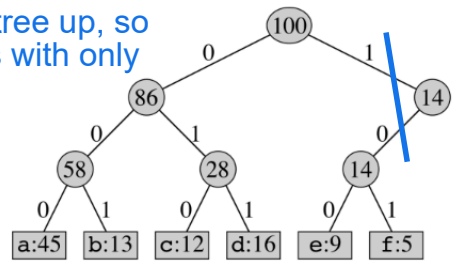
Codeword = path in a binary tree: 0 ~go left, 1 ~go right

Prefix-free code: no code for a character is the prefix of another character's code (⇒ decoding is unambiguous). So characters correspond to nodes with zero children (leaves).

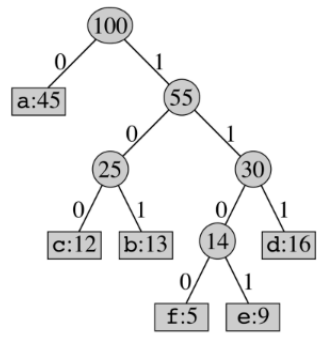
	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

can not have same prefix, fx 10 is not allowed, because than we don't know if it is 101, or 100.

cut this offand move the sub tree up, so no nodes with only 1 child



(a)



(b)

nodes are sum of subtrees

depth of tree * amount of thounds in nodes, fx 45, 12 etc.

For a given file (characters and their frequencies), find the best variable-width prefix code. That is, for Cost(tree) = |encoded file|, find the tree with the lowest cost. Optimal trees cannot have nodes with only one child (all characters in the subtree of such a node can be shortened by one bit, cf. (a) above).

So only nodes with either two or zero children are allowed.

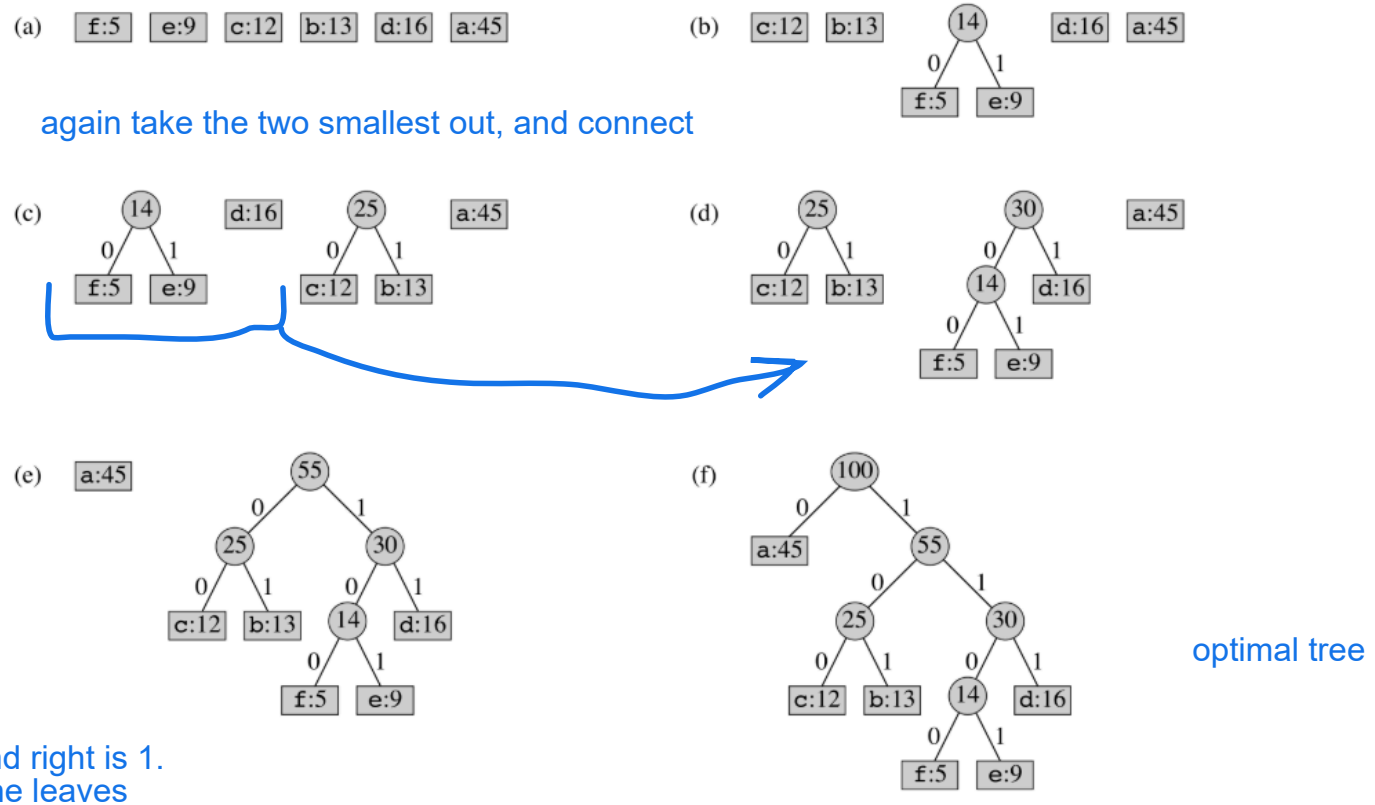
Huffman's algorithm

[David Huffman, 1952]

smallest to the left, biggest to the right

Build from the bottom up (from smallest to largest frequencies) by repeatedly making the following "greedy choice": Combine the two subtrees with the smallest total frequencies.

still sorted in increasing frequencies



again take the two smallest out, and connect

optimal tree

decode: left is 0 and right is 1.
follow the root to the leaves

Run time

Given a table with **n** characters and their frequencies, Huffman's algorithm performs:

- n − 1 iterations.

(There are **n** trees at the start, one final tree at the end, and each iteration reduces the number of trees by exactly one.)

By using a (min-)priority queue, e.g., implemented with a heap, each iteration can be performed with:

- Two `ExtractMin` operations
- One `Insert` operation
- O(1) other work

Each priority queue operation takes O(log n) time.

So the total runtime for the n iterations is O(n log n).

Correctness

Summary:

Huffman's algorithm maintains a collection of trees F. The weight of a tree is the sum of the frequencies in its leaves. At each step, Huffman combines the two trees with the smallest weights, until only one tree remains.

We will prove the following invariant:

The trees in F can be combined into an optimal tree.

When the algorithm finishes, F contains only one tree, which therefore — by the invariant — must be an optimal tree.

Correctness

We want to prove the following invariant:

The trees in F can be combined into an optimal tree.

The proof is by induction over the number of steps in the algorithm.

Base case: No steps taken yet. Then **F** consists of n trees, each just a leaf. These are exactly the leaves of any optimal tree, so the invariant clearly holds.

Inductive step: Assume the invariant holds before a step in the algorithm, and let us show it holds afterward.

Correctness

Let the trees in F (before the step) be:

$$t_1, t_2, t_3, \ldots, t_k .$$

such that the algorithm combines $\mathbf{t}_1$ and $\mathbf{t}_2$. That is, with regard to weight, we have:

$$t_1 \leq t_2 \leq t_3 \leq \text{- - - -} \leq t_k ,$$

By the inductive hypothesis, the trees can be combined into an optimal tree. Let T be the top of this tree — a tree whose leaves are the roots of t_1, t_2, t_3, ..., t□.

Correctness

Case 1: The roots of t_1 and t_2 are siblings in T.

Then, the new set of trees (after combining t_1 and t_2) can still be assembled into the same optimal tree mentioned in the invariant before the step.

So the invariant still holds after the step.

Correctness

Case 2: The roots of t_1 and t_2 are not siblings in T.
We will find another top-tree T′ (i.e., another way to combine the trees in F) which also gives an optimal tree and where t_1 and t_2 *are* siblings.
For this optimal tree, we fall back into Case 1, and we're done.

We construct T′ from T as follows:
Look at a leaf in T at the greatest depth. Since $k \geq 2$ (otherwise Huffman's algorithm would be finished), this leaf has a parent. This parent must have at least one subtree, and hence has two (since optimal trees don't have nodes with only one child, as noted earlier).

Its other subtree must be a leaf — otherwise, the first one wouldn't be at the greatest depth.
Thus, we find two sibling leaves in T, both at the maximum depth.
Let these contain the roots of t_i and t□, with $i < j$.

Correctness

Possible situations:

1	2	3 . . .
i	j	
i		j
	i	j
		i, j

Actions to obtain T' from T:

Do nothing (if we are in Case 1)

Swap t_2 and t_j

Swap t_1 and t_j

Swap t_1 and t_i , and t_2 and t_j

For a swap of t_1 and t_i, it holds that since t_i's root has at least the same depth as t_1's root, and the total frequency of t_i is at least as large as the total frequency of t_1, there will be more characters in the file that get shorter codewords than there are characters that get longer codewords. Furthermore, the change in the length of the codewords is the same for both shortening and lengthening (namely, the difference in depth between t_1 and t_i).
Therefore, the encoded file's length does not increase when swapping t_1 and t_i, so the tree does not become worse.

The same logic applies for swaps of t_1 and t□, and t_2 and t□.
Since the tree was optimal before the swap, it remains optimal after. And now, t_1 and t_2 are siblings.