Chapter 16: Greedy Algorithms

Greedy is a strategy that works well on optimization problems with the following characteristics:

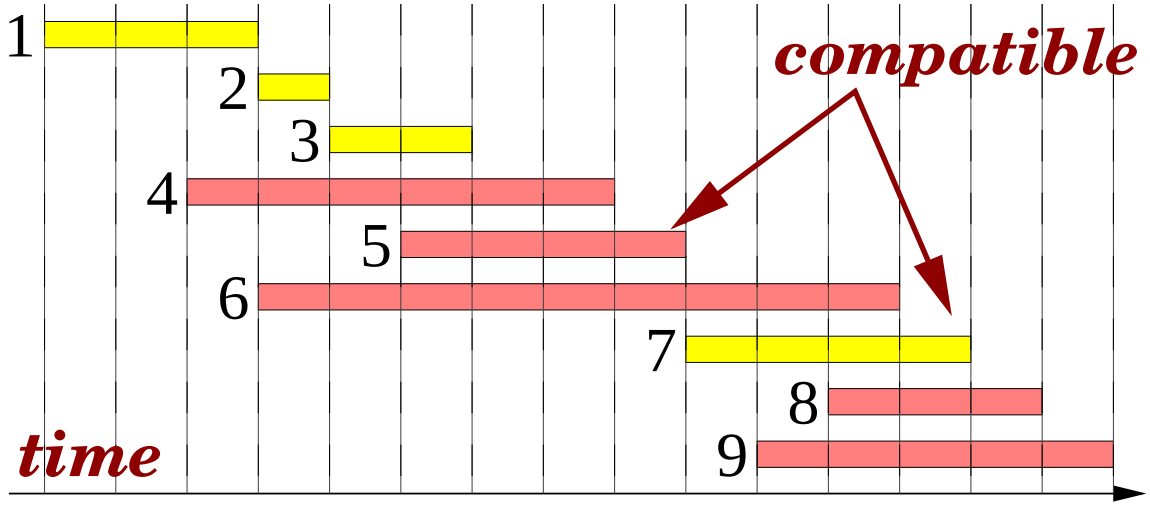
1. **Greedy-choice property:** A global optimum can be arrived at by selecting a local optimum.
2. **Optimal substructure:** An optimal solution to the problem contains an optimal solution to subproblems.

The second property may make greedy algorithms look like dynamic programming. However, the two techniques are quite different.

1. An Activity-Selection Problem

Let $S = \{1, 2, \ldots, n\}$ be the set of **activities** that compete for a resource. Each activity i has its **starting time** s_i and **finish time** f_i with $s_i \leq f_i$, namely, if selected, i takes place during time $[s_i, f_i)$. No two activities can share the resource at any time point. We say that activities i and j are **compatible** if their time periods are disjoint.

The **activity-selection problem** is the problem of selecting the **largest set** of mutually compatible activities.


1
2
3
4
5
6
7
8
9
compatible
time

Greedy Activity Selection Algorithm

In this algorithm the activities are first sorted according to their finishing time, from the earliest to the latest, where a tie can be broken arbitrarily. Then the activities are **greedily** selected by going down the list and by picking whatever activity that is compatible with the current selection.

What is the running time of this method?

Well, it depends on which sorting algorihtm you use.

The sorting part can be as small as $O(n \log n)$ and the other part is $O(n)$, so the total is $O(n \log n)$.

Theorem A **Greedy-Activity-Selector** solves the activity-selection problem.

Proof The proof is by induction on n.

For the base case, let $n = 1$. The statement trivially holds.

For the induction step, let $n \geq 2$, and assume that the claim holds for all values of n less than the current one. We may assume that the activities are already sorted according to their finishing time.

Let p be the number of activities in each optimal solution for $[1, \ldots, n-1]$ and let q be the number for $[1, \ldots, n]$.

Here $p \leq q$ holds.

Can you explain why?

It's because every optimal solution for $[1, \ldots, n-1]$ is a solution for $[1, \ldots, n]$.

How about the fact that $p \geq q - 1$?

How about the fact that $p \geq q - 1$*?*

Assume that $p \leq q - 2$*. Let* W *be any optimal solution for* $[1, \ldots, n]$*. Let* $W' = W - \{n\}$ *if* W *contains* n *and* $W' = W$ *otherwise. Then* W' *does not contain* n *and is a solution for* $[1, \ldots, n-1]$*. This contradicts the assumption that optimal solutions for* $[1, \ldots, n-1]$ *have* p *activities.*

Optimality Proof

We must first note that the greedy algorithm always finds some set of mutually compatible activities.

(Case 1) Suppose that $p = q$. Then each optimal solution for $[1, \ldots, n-1]$ is optimal for $[1, \ldots, n]$. By our induction hypothesis, when $n-1$ has been examined an optimal solution for $[1, \ldots, n-1]$ has been constructed. So, there will be no addition after this; otherwise, there would be a solution of size $> q$. So, the algorithm will output a solution of size p, which is optimal.

(Case 2) Suppose that $p = q - 1$. Then every optimal solution for $[1, \ldots, n]$ contains n. Let k be the largest i, $1 \leq i \leq n - 1$, such that $f_i \leq s_n$. Since $f_1 \leq \cdots \leq f_n$, for all i, $1 \leq i \leq k$, i is compatible with n, and for all i, $k + 1 \leq i \leq n - 1$, i is incompatible with n. This means that each optimal solution for $[1, \ldots, n]$ is the union of $\{n\}$ and an optimal solution for $[1, \ldots, k]$. So, each optimal solution for $[1, \ldots, k]$ has p activities. This implies that no optimal solutions for $[1, \ldots, k]$ are compatible with any of $k + 1, \ldots, n - 1$.

Let W be the set of activities that the algorithm has when it has finished examining k. By our induction hypothesis, W is optimal for $[1, \ldots, k]$. So, it has p activities. The algorithm will then add no activities between $k+1$ and $n-1$ to W but will add n to W. The algorithm will then output $W \cup \{n\}$. This output has $q = p+1$ activities, and thus, is optimal for $[1, \ldots, n]$.

2. Knapsack

The **0-1 knapsack problem** is the problem of finding, given an integer $W \geq 1$, items $1, \ldots, n$, and their values, $v_1, \ldots, v_n$, and their weights, $w_1, \ldots, w_n$, a selection, $I \subseteq \{1, \ldots, n\}$, that maximizes $\sum_{i \in I} v_i$ under the constraint $\sum_{i \in I} w_i \leq W$.

An example: George is going to a desert island. He is allowed to carry one bag with him and the bag holds no more than 16 pounds, so he can't bring all what he wants. So, he weighted and values each item he wants to bring. What should he be putting in the bag?

item	weight (lb.)	value
A CD player with Bernstein Mahler box	8	20
CLRS 2nd Ed.	10	25
Twister Game	2	8
SW Radio	4	12
Harmonica	1	5
Roller Blades	4	6
Inflatable Life-Size R2D2 Doll	1	8

Tell me what I should bring?

$\star$ There is an $O(nW)$ step algorithm based on dynamic programming

A greedy approach might be to:

- Sort the items in the decreasing order of values. Then scan the sorted list and grab whatever can squeeze in.

This approach does not work.
Can you tell us why?

Sure.

This strategy does not work because it does not take into consideration that a combination of less valued items may weigh less and still have a larger value.

item	weight (lb.)	value
CLRS 2nd Ed.	10	25
CD player with Mahler	8	20
SW Radio	4	10
Twister Game	2	8
R2D2 Doll	1	8
Harmonica	1	5
Roller Blades	4	2

With $W = 10$ you should pick CLRS 2nd Ed. but there is a better combination.

3. Huffman Coding

Storage space for files can be saved by compressing them, i.e. by replacing each symbol by a unique binary string.

Here the codewords can differ in length. Then they need to be *prefix-free* in the sense that no codeword is a prefix of another code. Otherwise, decoding is impossible.

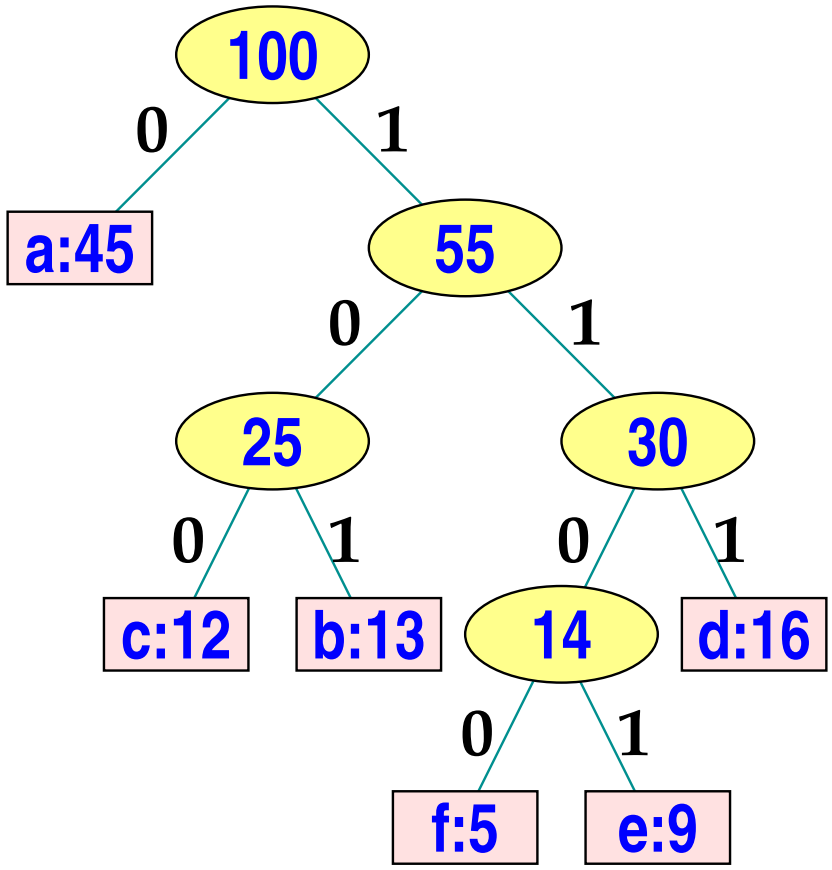
The **character coding problem** is the problem of finding, given an alphabet $C = \{a_1, \ldots, a_n\}$ and its frequencies $f_1, \ldots, f_n$, a set of prefix-free binary code $W = [w_1, \ldots, w_n]$ that minimizes the average code length

$$\sum_{i=1}^{n} f_i \cdot |w_i|.$$

Depict a prefix-free binary code using a binary tree, where each left branch corresponds to the bit 0, each right branch corresponds to the bit 1, and the leaves are uniquely labeled by the symbols in C.

The codeword of a symbol a in C is the concatenation of the edge labels that are encountered on the path from the root to a.

Each node v is labeled by the frequency sum of the symbols in $subtree(v)$.


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

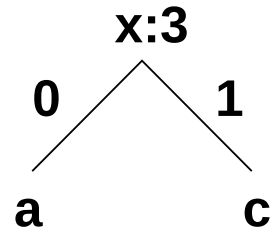
The Huffman coding is a greedy method for obtaining an optimal prefix-free binary code, which uses the following idea: For each i, $1 \leq i \leq n$, create a leaf node v_i corresponding to a_i having frequency f_i. Let $D = \{v_1, \ldots, v_n\}$. Repeat the following until $\|D\| = 1$.

- Select from D the **two nodes with the lowest frequencies.** Call them x and y.
- Create a node z having x as the left child and y as the right child.
- Set $f[z]$ to $f[x] + f[y]$.
- Remove x and y from D and add z to D.

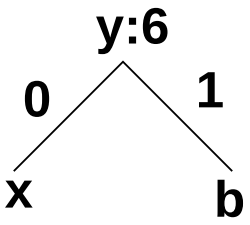
$\star$ The replacement will force the codeword of x (y) to be that of z followed by a 0 (a 1).

An example: a:1, b:3, c:2, d:4, e:5

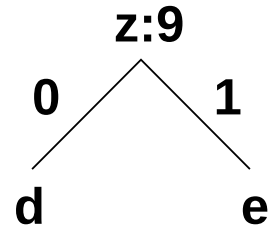
1. a & c $\rightarrow$ x:



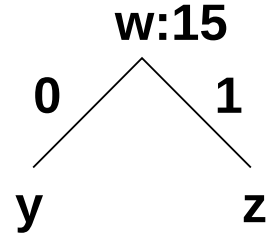
2. x & b $\rightarrow$ y:



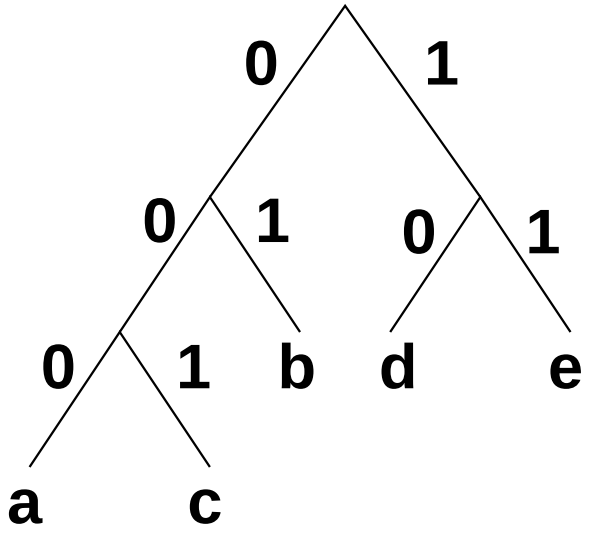
3. d & e $\rightarrow$ z:



4. y & z $\rightarrow$ w:



The resulting tree



The idea can be implemented using a priority-queue that is keyed on f.

The Correctness of The Greedy Method

Lemma B If x and y have the lowest frequencies in an alphabet C, then C has an optimal prefix code in which x and y are siblings.

Proof Let T be an optimal code and let h be the height of T. There are two leaves at depth h that are siblings. If they are not x and y, exchange the nodes with x and y. This will not increase the average code length. ∎

Lemma C Create an alphabet D from C by replacing x and y by a single letter z such that $f[z] = f[x] + f[y]$. Then there exists a one-to-one correspondence between

- the set of code trees for D in which z is a leaf and
- the set of code trees for C in which x and y are siblings.

Proof By contradiction:

$B(T)$ is cost of tree for C, $B(T')$ is cost of tree for D.

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

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

Suppose $B(T'') < B(T)$.

Create T''' by merging x and y in T''.

$$\begin{aligned} B(T''') &= B(T'') - f[x] - f[y] \\ &< B(T) - f[x] - f[y] \\ &= B(T') \end{aligned}$$

Suppose that x and y are letters with the lowest frequencies in C. Obtain an optimal code T for D and replace z by a depth-one binary tree with x and y as the leaves. Then we obtain an optimal code for C.