6 Greedy Algorithms

[BB chapter 6] with different examples or [CLRS3 chapter 16] with different approach to greedy algorithms

6.1 An activity-selection problem

Problem: We have a set of n activities $A_1, \ldots, A_n$ – activity A_i starts at time s_i and finishes at time f_i. We want to participate at as many activities as possible (but activities cannot overlap).

Example: Summer camp activity selection:

```
9  10  11  12   1   2   3   4   5
+---+---+---+---+---+---+---+---+
<--->   <-----> <------->   <->
Horseback Swi    Napping   Pizza
riding    mming
    <---> <->
 Canoeing Lunch
      <--->
     Kayaking

 9:00-10:00 Horseback riding *
10:00-11:00 Canoeing
11:00-12:30 Swimming
10:30-11:30 Kayaking *
11:30-12:00 Lunch *
 1:00- 3:00 Napping *
 4:00- 4:30 Pizza *
```

Q: Suggestions for algorithms to solve this problem?

A1: Shortest activity first

```
9  10  11  12   1   2   3   4   5
+---+---+---+---+---+---+---+---+
<---------------X--------------->
  Morning         Afternoon
  canoeing        canoeing
              <--->
              Lunch

Who needs lunch when you can canoe all day?
```

A2: First starting activity first

```
9  10  11  12   1   2   3   4   5
+---+---+---+---+---+---+---+---+
<------------------------------->
        All day trip
  <-----------X---X------------->
   Basketball Lunch Frisbee
```

A3: Activity with the smallest number of conflicts first

```
9  10  11  12   1   2   3   4   5
+---+---+---+---+---+---+---+---+
<--->   <----->   <----->   <--->
<--->        <------->      <--->
   <------->           <------->
   <------->           <------->
```

Solution: First ending activity first

```
Sort all activities by their finishing time
(now f[1]<=f[2]<=...<=f[n])

last_activity_end:=-infinity;

for i:=1 to n
  if (s[i]>=last_activity_end) then
     output activity (s[i],f[i]);
     last_activity_end:=f[i];
```

Running time: $\Theta(n \log n)$

Note:

- All previous examples correct
- There can be more than one optimal solution

Proof of correctness.

Assume without loss of generality:

- Activities are sorted by their finishing time, i.e. $f_1 \leq f_2 \leq \ldots \leq f_n$.
- We assume all solutions in the text below are sorted in the same order.

Lemma 1. *Assume the greedy solution selected activities $G = (G_1, \ldots, G_k)$. Then for any $0 \leq l \leq k$ there exists an optimal solution of the form $O = (G_1, \ldots, G_l, O_{l+1}, \ldots, O_m)$.*

Proof. **Proof by induction on l.**

Base case. If $l = 0$ then the statement holds trivially.

Induction step. Assume that the statement holds for l. Therefore there exists an optimal solution $O = (G_1, \ldots, G_l, O_{l+1}, \ldots, O_m)$.

Note that:

- $s_{O_{l+2}} \geq f_{O_{l+1}}$ (because O must be a correct solution of the activity selection problem),
- $f_{G_{l+1}} \leq f_{O_{l+1}}$ (because, otherwise, O_{l+1} would have been chosen by the greedy algorithm).

Therefore G_{l+1} can be substituted for O_{l+1} in the solution O, yielding solution O'. Solution O':

- is of the same size as O (therefore it is optimal),

- agrees with G on at least $l+1$ first activities

Thus the statement holds for $l+1$ as well.

□

Theorem 1. *The greedy algorithm always finds an optimal solution.*

Proof. Using previous lemma for $l = k$, we know that there exists an optimal solution of the form

$$O = (G_1, \dots, G_k, O_{k+1}, \dots, O_m).$$

Assume that $m > k$. Then this means that starting time $s_{O_{k+1}} \geq f_{G_k}$; but O_{k+1} would be added to G by the algorithm. **Contradiction.** □

6.2 Greedy algorithms – summary

Approach we have taken to solve the activity selection problem is, in general, called **greedy**.

Outline of typical greedy algorithm.

- Every solution can be obtained by series of choices.
  e.g.: *choice of activities in activity selection problem*
- But not all choices lead to an optimal solution.
  e.g.: *some sets of activities are smaller than the optimal set; not all sets of activities can be extended to an optimal set*
- In each step:
  - Consider all options for the current choice.
    e.g.: *what activity to choose next?*
  - Weight the options by a $\boxed{\text{weighting function}}$.
    e.g.: *finishing time of the activity*
  - Take the option which has the largest weight
    (or: choose whatever seems best right now)
    e.g.: *choose activity with the smallest finishing time*

The most challenging part is to **prove that a greedy algorithm yields $\boxed{\textbf{an}}$ optimal solution.** (Remember: usually there can be more than one optimal solution.)

Outline of typical proof. (one possible way)

Lemma Template 1. *Assume the greedy algorithm gives the solution G.*
There exists an optimal solution which agrees with G on first k choices.

Proof. **By induction on k.**

Base case. For $k = 0$ – any optimal solution will do.
(Who could make a mistake when presented with no choice?)

Induction step. (Assume we did not make mistake in first k choices; show that $(k+1)$st choice was OK as well.)

- Assume that there exists an optimal solution OPT which agrees with the greedy solution on first k choices.

- Create a new solution OPT' such that:
  - OPT' has the same value as OPT (and therefore is optimal as well)
  - It agrees with G on one more $(k+1)$st choice.

$\square$

Points to take home:

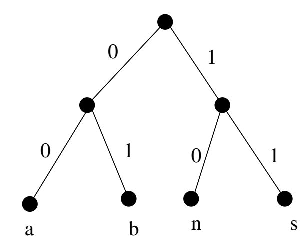
- Greedy algorithms are usually simple to describe and have fast running times ($\Theta(n)$ or $\Theta(n \log n)$).
- The hard part is demonstrating that the solution is optimal.
- This can be often done by induction: "change" any optimal solution to the greedy one without changing its cost.

6.3 Huffman codes

Binary prefix codes. Assume we have an alphabet of four characters: a, b, n, s. Let us represent these characters in binary code as follows:

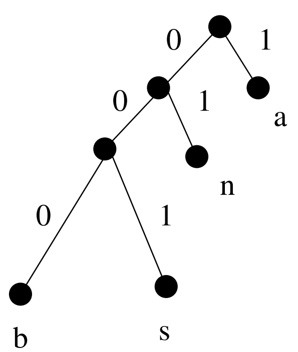
```
a 00
b 01
n 10
s 11

bananas   01001000100011 (14 bits)
```

Binary tree representation: leaves = characters of the alphabet; path to a leaf = binary code for the character



Q: Must all leaves have the same depth?

A: No!



Encoding: For each character locate corresponding leaf and follow the path, adding 0s when going left and 1s when going right.
`bananas   0001011011001 (13 bits - wow!)`

Decoding: Start from the root of the tree, when you see 0 go left, when you see 1 go right, when you enter leaf write-out the letter and start from the root again.

Note: Binary codes that can be represented by a tree are called *prefix codes* (code of any character cannot be a prefix of code of any other character).

Idea: For a given string, different trees give a different length of the encoding. Thus by choosing a proper tree we can **compress the string.**

Problem: Given a string $S = s_1 s_2 \dots s_m$ over alphabet Σ ($|\Sigma| = n$), find a prefix code (i.e. binary tree) that yields the shortest encoding of the string.

(Such a tree is called **Huffman's tree**)

Notation:

- **Frequency** $f(x)$ of a character x in string S is the number of characters x occurring in string S.
- We can extend this to a **frequency of a subtree** C **of the tree** T**:**

$$f(C) = \sum_{x \text{ is a leaf in } C} f(x)$$

- Let $\text{depth}_T(x)$ be the **depth** of a leaf x in a tree T.
- **Weight** $w(T)$ of a tree T is the length of the encoding of string S using tree T (in bits):

$$w(T) = \sum_{i=1}^{m} \text{depth}_T(s_i) = \sum_{x \in \Sigma} f(x) \text{depth}_T(x)$$

- We can extend this to a **weight of a subtree** C **of the tree** T**:**

$$w(C) = \sum_{x \text{ is a leaf in } C} f(x) \cdot \text{depth}_C(x)$$

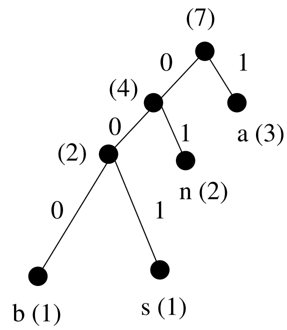
Observation: The characters which occur less often should be located deeper in the tree.

Greedy algorithm:

```
Compute frequencies of all characters in S

F:=empty-forest;
for all characters x in the alphabet do
  T:=new leaf(x);
  add T to F;

while F contains more than one tree do
  T1:=extract tree with minimum frequency from F;
  T2:=extract tree with minimum frequency from F;
  T:=new tree where T1 is a left child
              and   T2 is a right child;
  add T to F;

return F;
```

Example:

```
bananas:

x      f(x)
-----------
b         1
a         3
n         2
s         1
```



Proof of correctness.

Lemma 2. *Let $F = (T_1, T_2, \ldots, T_k)$ is a forest obtained by the greedy algorithm after i steps. Then there exists an optimal coding tree which contains $T_1, T_2, \ldots, T_k$ as subtrees.*

Note: From the lemma: after $n-1$ steps of the greedy algorithm we obtain an optimal tree.

Proof. By induction on i.

Base case. After 0 steps, we have a forest composed of singleton vertices – the lemma holds trivially.

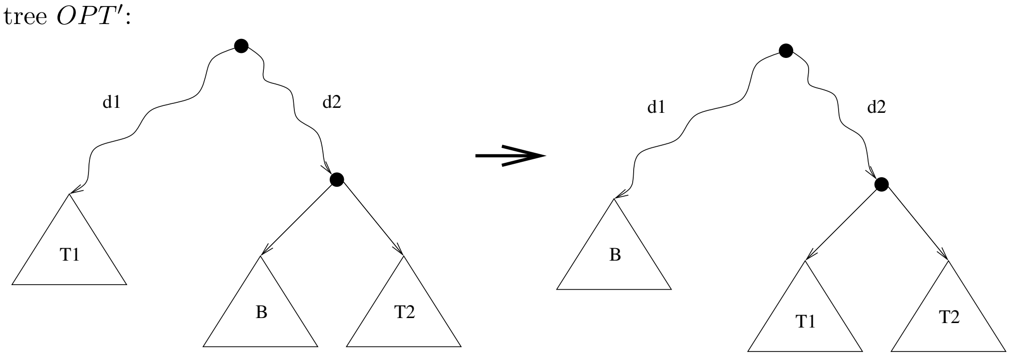
Induction step.

- Assume that after i steps the greedy algorithm has a forest $F = (T_1, T_2, \ldots, T_k)$.
- From IH we can assume that there exists an optimal tree OPT which contains all $T_1, \ldots, T_k$ as subtrees.
- Without loss of generality, we can assume that these trees are labeled in the following way:
  - Trees T_1 and T_2 are the trees that are subsequently chosen in step $i+1$ of the greedy algorithm and combined into a new tree T'. Consequently, for all $j \geq 3$, $f(T_j) \leq f(T_1)$ and $f(T_j) \leq f(T_2)$.
  - In OPT, tree T_2 is the one that is positioned **deeper or in the same depth** as T_1.

  (Note that this can be easily achieved by simpy relabeling the trees.)

Now, let us compare the tree OPT with the the subtrees $(T', T_3, \ldots, T_k)$.

- If T_1 and T_2 are siblings in OPT we are done (T' is a subtree of OPT and thus the lemma holds for $i+1$ steps as well).
- Otherwise: T_2 must have a sibling subtree B. Exchange T_1 and B, as on the picture, yielding new tree OPT':



Note:

- Contribution of a leaf x to the weight of the tree T is $\text{depth}_T(x) \cdot f(x)$.
- Contribution of a subtree T_1 to the weight of the tree T is:

$$\sum_{x \text{ is a leaf in } T_1} \text{depth}_T(x) \cdot f(x) = d_1 \cdot f(T_1) + w(T_1)$$

Weight before (i.e., weight of OPT):

$$BEFORE = d_1 f(T_1) + w(T_1) + (d_2+1) f(B) + w(B) + (d_2+1) f(T_2) + w(T_2) + REST$$

Weight after (i.e., weight of OPT'):

$$AFTER = d_1 f(B) + w(B) + (d_2+1) f(T_1) + w(T_1) + (d_2+1) f(T_2) + w(T_2) + REST$$

Difference:

$$w(OPT') - w(OPT) = AFTER - BEFORE = (f(B) - f(T_1))(d_1 - (d_2+1))$$

Note:

- $T_1, \dots, T_k$ contain all leaves; therefore B is either one of $T_3, \dots, T_k$ or it contains one of them (because OPT contains $T_1, \dots, T_k$ as subtrees).
- Thus for some $j \geq 3$: $f(B) \geq f(T_j) \geq f(T_1)$
- Since T_2 was deeper or in the same depth in OPT than T_1, $d_2 + 1 \geq d_1$.
- Thus: $AFTER - BEFORE \leq 0$

If $AFTER - BEFORE < 0$, that would mean that cost of OPT' would be **better** than cost of OPT. That would contradict the assumption that OPT is the optimal tree. Therefore $AFTER = BEFORE$, the tree OPT' has the same cost as OPT and thus is an **optimal tree that contains** $(T', T_3, \dots, T_k)$ **as subtrees**, which shows that the lemma also holds after $i+1$ steps of the algorithm.

□

How long does it take? Depends on the implementation of the "forest" data structure:

- **list of trees:** $\Theta(m + n^2)$
- **priority queue:** $\Theta(m + n \log n)$

6.4 Aside: Text compression and Lempel-Ziv-Welch (LZW) compression algorithm

In compression, we do not want to create a specific output for a given input. Instead we want to provide a pair of algorithms:

- **Compression algorithm.** Encodes a given input string S into a compressed string $C(S)$, which is preferrably shorter than string S.
- **Decompression algorithm.** For every valid compressed string $C(S)$, recover the original string S.

Can we always compress a string to 80%? No!

- There are much fewer strings of length $0.8n$ than strings of length n. Therefore, if we compressed all the strings of length n to length $0.8n$, there would have to be some strings that share the same code, i.e. strings $S_1 \neq S_2$ for which $C(S_1) = C(S_2)$. But then, we cannot decompress those strings!
- If we could compress every string to 80%, then there is nothing to stop us to use the compression again on compressed strings, giving us even better compression rate of 64%. We can apply compression again and again, until we end up with a single bit...

Lempel-Ziv-Welch compression is based on replacing **variable length substrings** of the original string with **fixed length code words**. Typically, these code words are 12-bits long (i.e., numbers between 0 and 4095), and the compressed text is simply a sequence of these code words.

Note: If we simply designate a single 12-bit code word to each character of 7-bit ASCII code, and simply replace the characters with these code words, we would increase the size of the resulting file by more than 70%. Therefore, the algorithm relies on replacing longer and longer strings with single code word.

If we replace pair of characters with a single code, we have compressed this pair to $\approx 85\%$. However, there is space only for less than 1/4 of all pairs of characters. Similarly, replacing triple of characters with a single code gives us compression ratio $\approx 67\%$, however we can store only 0.2% of all triples in the dictionary. Therefore, choosing a good dictionary is crucial for the performance.

In LZW compression, we are building the dictionary on a fly so that it can be reconstructed even if we only have the compressed string. The algorithm is heuristic—the strings which occur more often will eventually be included in the dictionary.

$\boxed{\textbf{Compression algorithm.}}$

```
LEMPEL-ZIV-WELCH-COMPRESS:
  create empty dictionary D;
  // dictionary D will map strings to number 0,1,...,4095

  // initialize dictionary D with all characters from the alphabet
  dsize := 0;
  for all symbols s in alphabet
    D.insert(s,dsize);
    dsize := dsize + 1;

  // process the input
  while there is more characters on the input
    s := longest prefix from input
         such that s is in D  (*)
    output D.search(s);

    c := peek next character from input
    D.insert(s+c,dsize);
    dsize := dsize + 1;
```

Example:

```
COCOA_AND_BANANAS

Alphabet: _,A,B,C,D,N,O,S
```

```
          0 1 2 3 4 5 6 7
```

Longest prefix	Output code	Entry in dictionary	Dictionary number
C	3	CO	8
O	6	OC	9
CO	8	COA	10
A	1	A_	11
_	0	_A	12
A	1	AN	13
N	5	ND	14
D	4	D_	15
_	0	_B	16
B	2	BA	17
AN	13	ANA	18
ANA	18	ANAS	19
S	7	--	--

Therefore, the compressed string is: $3, 6, 8, 1, 0, 1, 5, 4, 0, 2, 13, 18, 7$

What do we do when we run out of codewords?

- stop adding new strings in the dictionary
- or: clear the dictionary and start building it anew

(but we must do the same thing in both compression and decompression algorithms)

Decompression algorithm. Decompression algorithm works similarly as the compression algorithm: it builds the same dictionary as the compression algorithm, and thus is able to decompress the string.

```
LEMPEL-ZIV-WELCH-DECOMPRESS: (bad version)
  create empty dictionary D

  // initialize dictionary D with all characters from
  // the alphabet
  dsize := 0;
  for all symbols s in alphabet
    D.insert(dsize,s);
    dsize := dsize + 1;

  // process the compressed sequence
  code := next code from the input
  s := D.search(code)
  output s

  while there are more codes on the input
    lasts := s
    code := next code from the input
    s := D.search(code);
    output s;
```

```
    // inv: lasts is the string corresponding to the last code word
    //      s is the string corresponding to the current code word

    // build dictionary
    D.insert(dsize,lasts+s[1]);
    dsize := dsize + 1;
```

Example:

```
Alphabet: _,A,B,C,D,N,O,S
          0 1 2 3 4 5 6 7

3 6 8 1 0 1 5 4 0 2 13 18 7

      Decoded   Entry in    Dictionary
Code  string    dictionary  number
3     C         --          --
6     O         CO          8
8     CO        OC          9
1     A         COA         10
0     _         A_          11
1     A         _A          12
5     N         AN          13
4     D         ND          14
0     _         D_          15
2     B         _B          16
13    AN        BA          17
18    ????
```

We have got number 18, but it is not in the dictionary yet! How can we decode it?

Special case: Let as assume, that word Kw (where K is a character, and w is some string) is in the dictionary, and we have to compress word $KwKwK$. **Then and only then** during the compression:

- we insert KwK into the dictionary
- we use it immediately in the next step of the compression algorithm

However, you can observe from the examples, that even though the decompression algorithm is building the same dictionary, as the compression algorithm, this process is **delayed by one step** in decompression, and therefore the code of word KwK is not available during decompression when it is needed.

Since we know, that this is the only such example, this problem is easily solved: if we encounter an unknown code, and the previously used code word expanded to the string of the form Kw, then the next code word must expand to the string of the form KwK.

```
LEMPEL-ZIV-WELCH-DECOMPRESS: (correct version)
  create empty dictionary D

  // initialize dictionary D with all characters from
  // the alphabet
  dsize := 0;
  for all symbols s in alphabet
```

```
    D.insert(dsize,s);
    dsize := dsize + 1;

  // process the compressed sequence
  code := next code from the input
  s := D.search(code)
  output s

  while there are more codes on the input
    lasts := s
    code := next code from the input
**  if code = dsize then s := lasts + lasts[1];
**  else s := D.search(code);
    output s;

    // build dictionary
    D.insert(dsize,lasts+s[1]);
    dsize := dsize + 1;
```

Now we can finish the example:

```
Alphabet: _,A,B,C,D,N,O,S
          0 1 2 3 4 5 6 7

3 6 8 1 0 1 5 4 0 2 13 18 7

      Decoded   Entry in    Dictionary
Code  string    dictionary  number
3     C         --          --
6     O         CO          8
8     CO        OC          9
1     A         COA         10
0     _         A_          11
1     A         _A          12
5     N         AN          13
4     D         ND          14
0     _         D_          15
2     B         _B          16
13    AN        BA          17
18    ANA       ANA         18   !!!
7     S         ANAS        19
```

Implementing LZW compression.

- for decompression: there is nothing special (use a simple array with indexes [0..max] for D)

  Running time: can be done in $O(n)$

- for compression: we need to extend dictionary to support quickly the following operation:

```
    s := longest prefix from input
         such that s is in D  (*)
```

Typical solution: hash table with hash function that allows adding one character at a time (similar to shifting in Rabin-Karp)

Less typical solution: tries (usually requires more memory)

Running time: can be done in O(n) (with tries)

Patent issues. The LZW algorithm not only played important role in computer science, in both software (compression of data on file systems) and hardware (increasing speed of data transmission by compression of data flow), but it also supplies an important story in intellectual property management (you can see more details about this story at `http://www.kyz.uklinux.net/giflzw.php`):

- 1984: Scientific publication in IEEE Computer magazine Terry Welch: A Technique for High-Performance Data Compression. US patent awarded in 1985 (so called Unisys patent)
- Shortly after scientific publication, the use of the algorithm quickly spreads:
  - Unix compress (1984)
  - GIF interchanged format (1987)
  - compression in modems (1989) (patent license)
  - postscript compression (1989) (Adobe pays license for their tools)
- Unisys waits until 1994 until GIF gets popular and then says: any developers who write software that creates or reads GIF file format has to buy licenses of the patent from Unisys.

  Developers react by creating patent-free PNG format, most users abandon use of GIFs, since developers are removing support for the format.
- In 1995: Unisys granted royalty free license for non-commercial non-profit software. But this license was retracted later (in 1999).
- Patent coverage finally expires (2003 in USA, 2004 in the rest of the world). Now everybody can use the LZW compression legally again.

According to US constitution, the main purpose of patents is to "promote the progress of science and useful arts", and "securing for limited times to authors and inventors the exclusive right to their respective writings and discoveries" is only means of achieving this goal. Do you think that Unisys patent contributed to progress in computer science?

6.5 Coin changing

Problem 1: Coin changing in European coin system. Assume that we have unlimited amount of 1c, 2c, 5c, 10c, 20c, 50c, 1 euro, and 2 euro coins. Pay out a given sum S with the smallest number of coins possible.

Solution: Greedy algorithm again – try the largest coin first.

```
while S>0 do
  c:=value of the largest coin no larger than S;
  num:=S div c;
  pay out num coins of value c;
  S:=S-num*c;
```

Time: $\Theta(m)$, where m is the number of coins.

Proof of correctness:

Lemma 3. *No optimal solution will contain more than one 1c coin, two 2c coins, one 5c coin, one 10c coins, two 20c coin, one 50c coin, and one 1 euro coin.*

Proof. If we have two 1c, 5c, 10c, 50c, or 1 euro coins, we can always replace it with one coin of higher denomination. Three 2c coins can be replaced by 5c+1c, three 10c coins can be replaced by 20c+10c. Thus, each of these cases would result in a better solution. □

Lemma 4. *Suppose the greedy algorithm gave a solution $G = (g_1, \ldots, g_k)$ (where $g_1 \geq g_2 \geq \ldots \geq g_k$). Then for any $l \leq k$ there exists an optimal solution of the form $OPT = (g_1, \ldots, g_l, o_{l+1}, \ldots, o_m)$ (where $g_l \geq o_{l+1} \geq \ldots \geq o_m$).*

Proof. By induction on l.

Base case. For $l = 0$ the claim is trivial.

Induction step. Assume that the claim holds for l. Thus we may assume that there exists an optimal solution of the form $OPT = (g_1, \ldots, g_l, o_{l+1}, \ldots, o_m)$. We claim that $o_{l+1} = g_{l+1}$.

Assume to the contrary that $o_{l+1} \neq g_{l+1}$. Since the greedy algorithm chose the largest coin which could be paid out at the moment, all coins $o_{l+1}, \ldots, o_m$ are smaller than g_{l+1}. If we did not use the coin g_{l+1}, we would have to cover this sum by smaller coins.

Case 1. $g_{l+1} = 2$ The coins $o_{l+1}, \ldots, o_m$ must sum to at least 2 euros. However, it follows from Lemma 3 that the optimal solution must then contain 3 quarters, 2 dimes, and 1 nickel – but these can be replaced by loonie and the number of coins would decrease. **Contradiction.**

Case 2. $g_{l+1} = 0.25$ Then the optimal solution must contain 2 dimes and 1 nickel – but these can be replaced by quarter and the number of coins would decrease. **Contradiction.**

Case 3. $g_{l+1} = 0.10$ The optimal solution can contain at most 1 nickel and 4 pennies but those would not make it for a dime. **Contradiction.**

Case 4. $g_{l+1} = 0.05$ The optimal solution can contain at most 4 pennies but those would not make it for a nickel. **Contradiction.** □

Problem 2: Paying postage. The Canadian postage stamp system currently has the following small stamps: all values 1c-5c, 9c, 10c, 25c, 48c. Pay out a given sum S with the smallest number of stamps possible.

The greedy algorithm does not work! Sum 18 would be paid as 10+5+3 by the greedy algorithm, but the two 9 cent stamps is a better solution.

How to solve the coin changing in general?