3 Greedy Algorithms

[BB chapter 6] with different examples or [Par chapter 2.3] with different examples or [CLR2 chapter 16] with different approach to greedy algorithms

3.1 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
```

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. $f_1 \le f_2 \le \ldots \le f_n$.
- 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* $0 \le l \le k$ *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:

- $s_{O_{l+2}} \ge f_{O_{l+1}}$ (because O must be a correct solution of the activity selection problem),
- $f_{G_{l+1}} \le f_{O_{l+1}}$ (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

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

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

3.2 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.

□

Points to take home:

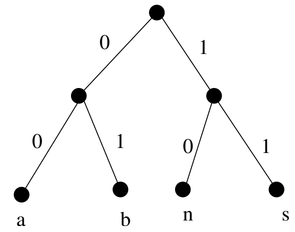
- Greedy algorithms are usually simple to describe and have fast running times ($\Theta(n)$ or $\Theta(n \log n)$).
- The hard part is demonstrating that the solution is optimal.
- This can be often done by induction: "change" any optimal solution to the greedy one without changing its cost.

3.3 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:

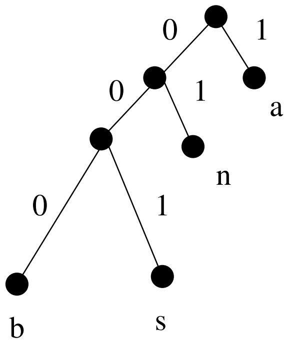
```
a 00
b 01
n 10
s 11

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 $S = s_1 s_2 \ldots s_m$ 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 $\mathrm{depth}_T(x)$ 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.

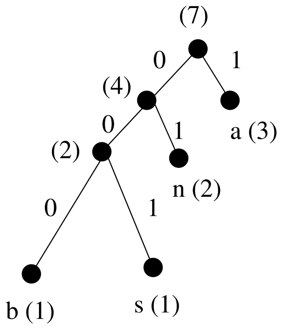
Greedy algorithm:

```
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.

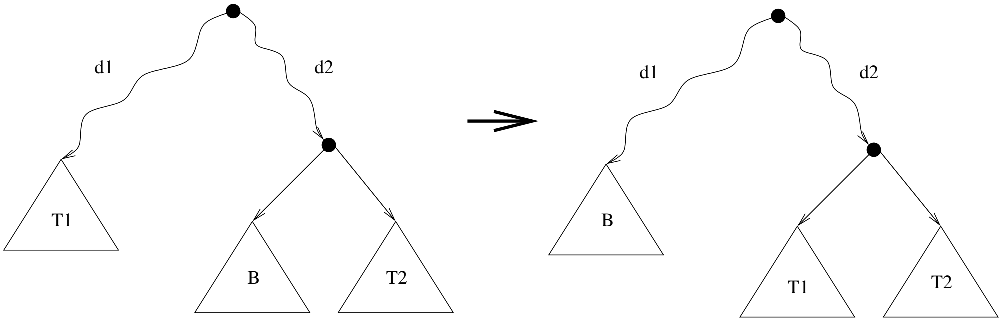
Proof. By induction on i.

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 the greedy algorithm in the step $i+1$ joins T_1 and T_2 to T', **and that T_2 is positioned deeper (or in the same depth) than T_1.**

  (Note the difference from the lecture presentation!)
- 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 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 $\mathrm{depth}_T(x) \cdot f(x)$.
- Contribution of a subtree T_1 to the weight of the tree T is:

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

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

$$\mathit{BEFORE} = d_1 f(T_1) + w(T_1) + (d_2+1) f(B) + w(B) + (d_2+1) f(T_2) + w(T_2) + \mathit{REST}$$

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

$$\mathit{AFTER} = d_1 f(B) + w(B) + (d_2+1) f(T_1) + w(T_1) + (d_2+1) f(T_2) + w(T_2) + \mathit{REST}$$

Difference:

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

Note:

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

- Thus OPT' is an optimal tree and it contains $(T', T_3, \ldots, T_k)$ as subtrees.

□

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

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