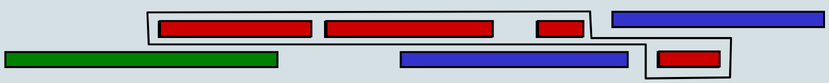
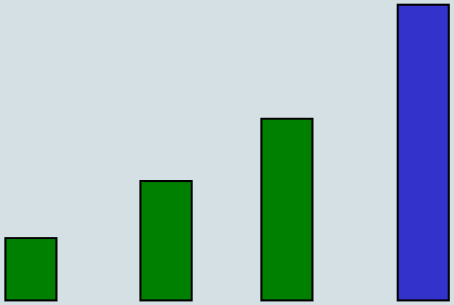
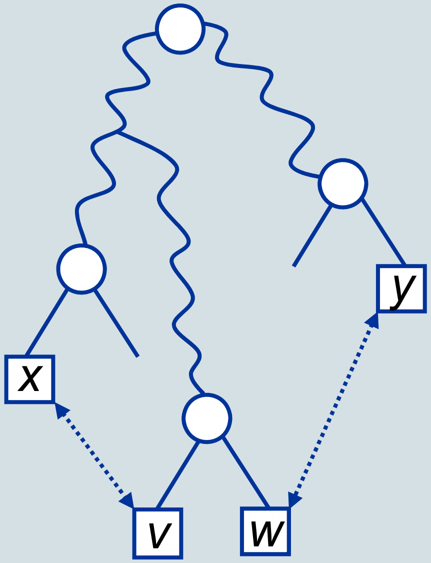
Algorithms (2IL15) – Lecture 2

THE GREEDY METHOD







Optimization problems

- for each instance there are (possibly) multiple valid solutions
- goal is to find an optimal solution

• minimization problem:

associate cost to every solution, find min-cost solution

• maximization problem:

associate profit to every solution, find max-profit solution

Techniques for optimization

optimization problems typically involve making choices

backtracking: just try all solutions

- can be applied to almost all problems, but gives very slow algorithms
- try all options for first choice,
  for each option, recursively make other choices

greedy algorithms: construct solution iteratively, always make choice that seems best

- can be applied to few problems, but gives fast algorithms
- only try option that seems best for first choice (greedy choice),
  recursively make other choices

dynamic programming

- in between: not as fast as greedy, but works for more problems

Algorithms for optimization: how to improve on backtracking

for greedy algorithms

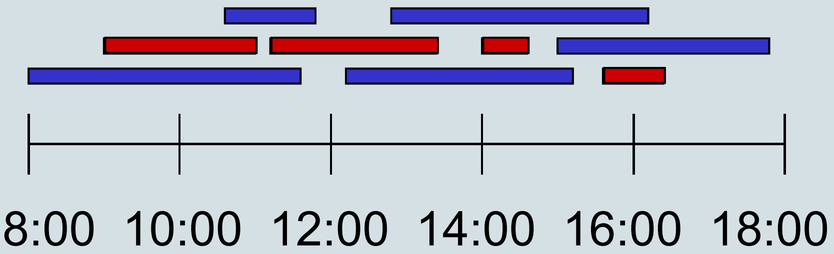
1. try to discover structure of optimal solutions: what properties do optimal solutions have ?
   - what are the choices that need to be made ?
   - do we have optimal substructure ?
     optimal solution = first choice + optimal solution for subproblem
   - do we have greedy-choice property for the first choice ?

2. prove that optimal solutions indeed have these properties
   - prove optimal substructure and greedy-choice property

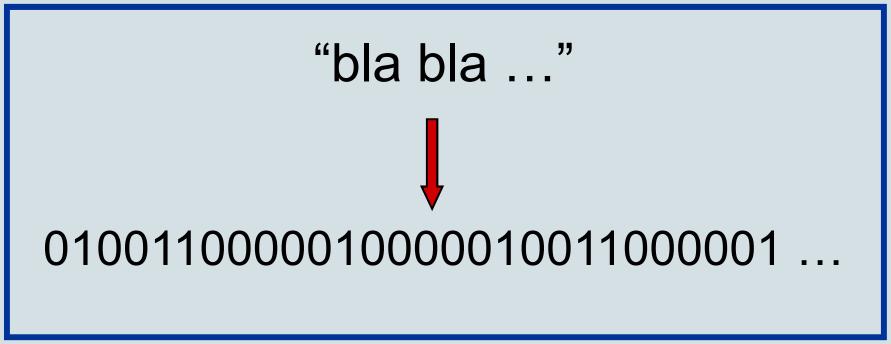
3. use these properties to design an algorithm and prove correctness
   - proof by induction (possible because optimal substructure)

Today: two examples of greedy algorithms

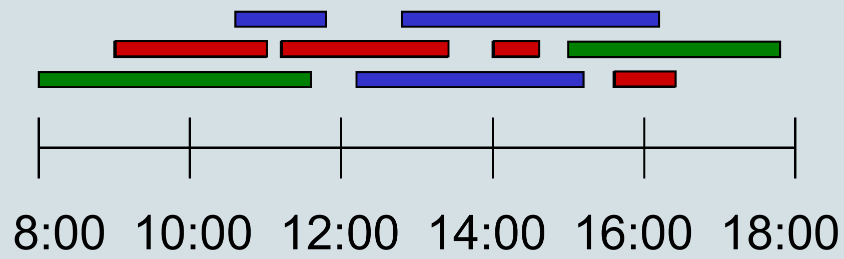
- Activity-Selection



- Optimal text encoding



Activity-Selection Problem

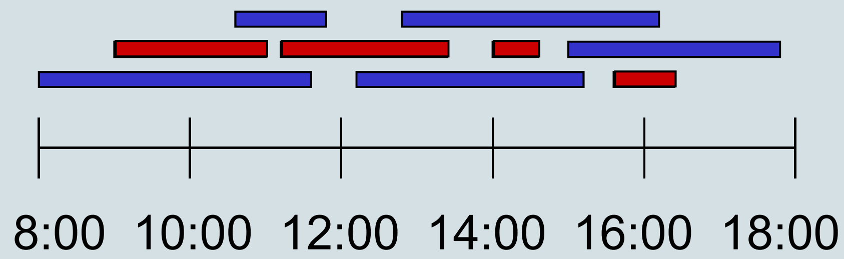


Input: set $A = \{a_1, \ldots, a_n\}$ of n activities
for each activity a_i: start time *start*(a_i), finishing time *end*(a_i)

Valid solution: any subset of non-overlapping activities

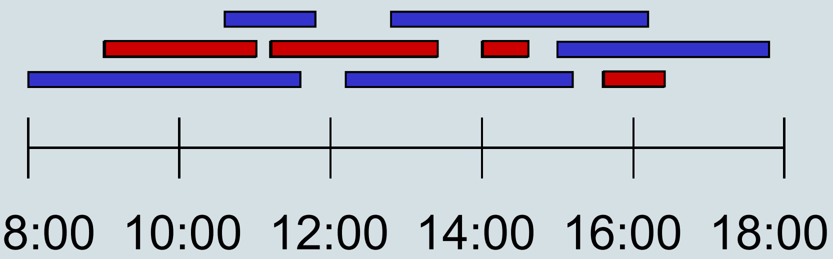
Optimal solution: valid solution with maximum number of activities

What are the choices ? What properties does optimal solution have?



- for each activity, do we select it or not?
  better to look at it differently …

What are the choices ? What properties does optimal solution have?



- what is first activity in optimal solution, what is second activity, etc.

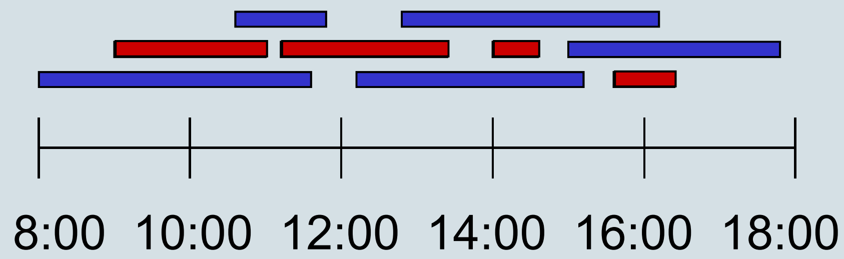
  do we have optimal substructure?

  optimal solution = first choice + optimal solution for subproblem ?

  yes!

  optimal solution = first activity + optimal selection from activities that do not overlap first activity

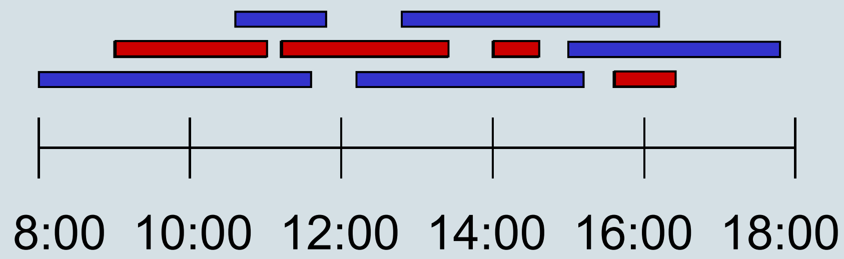
proof of optimal substructure



Lemma: Let a_i be the first activity in an optimal solution OPT for A.
Let B be the set of activities in A that do not overlap a_i.
Let S be an optimal solution for the set B.
Then S U $\{a_i\}$ is an optimal solution for A.

Proof. First note that S U $\{a_i\}$ is a valid solution for A. Second, note that OPT \ $\{a_i\}$ is a subset of non-overlapping activities from B. Hence, by definition of S we have size(S) ≥ *size* (OPT \ $\{a_i\}$), which implies that S U $\{a_i\}$ is an optimal solution for A. ■

What are the choices ? What properties does optimal solution have?



- do we have greedy-choice property:
  can we select first activity "greedily" and still get optimal solution?

  yes!

  first activity = $\underbrace{\text{activity that ends first}}_{\text{"greedy choice"}}$

$A = \{a_1, \ldots, a_n\}$: set of n activities

Lemma: Let a_i be an activity in A that ends first. Then there is an optimal solution to the Activity-Selection Problem for A that includes a_i.

Proof. General structure of all proofs for greedy-choice property:

- take optimal solution
- if OPT contains greedy choice, then done
- otherwise modify OPT so that it contains greedy choice, without decreasing the quality of the solution

Lemma: Let a_i be an activity in A that ends first. Then there is an optimal solution to the Activity-Selection Problem for A that includes a_i.

Proof. Let OPT be an optimal solution for A. If OPT includes a_i then the lemma obviously holds, so assume OPT does not include a_i.
We will show how to modify OPT into a solution OPT* such that

(i) OPT* is a valid solution
(ii) OPT* includes a_i
(iii) size(OPT*) ≥ size(OPT)
quality OPT* ≥ quality OPT

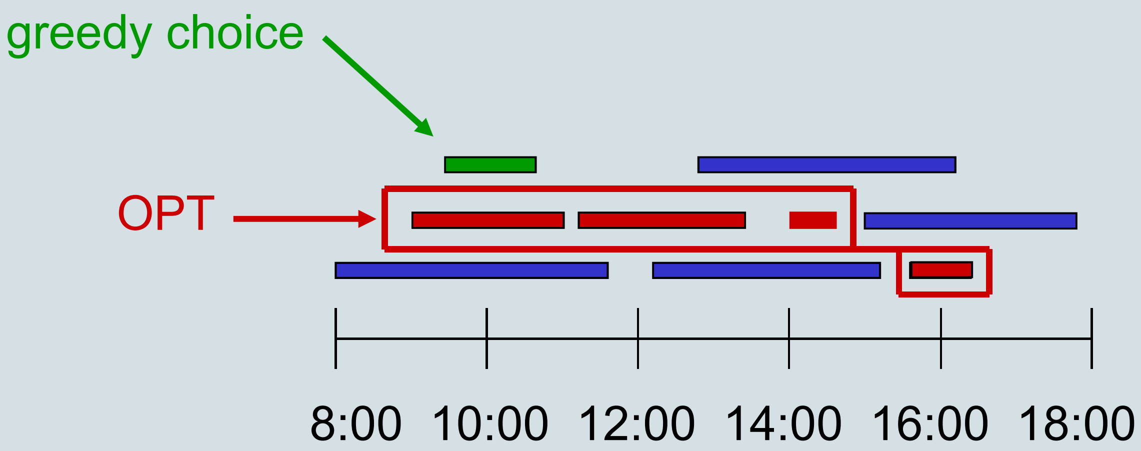
standard text you can basically use in proof for any greedy-choice property

Thus OPT* is an optimal solution including a_i, and so the lemma holds. To modify OPT we proceed as follows.

here comes the modification, which is problem-specific

■

How to modify OPT?



replace first activity in OPT by greedy choice

Lemma: Let a_i be an activity in A that ends first. Then there is an optimal solution to the Activity-Selection Problem for A that includes a_i.

Proof. […] We show how to modify OPT into a solution OPT* such that

(i) OPT* is a valid solution
(ii) OPT* includes a_i
(iii) size(OPT*) ≥ size(OPT)

[…] To modify OPT we proceed as follows.

Let a_k be activity in OPT ending first, and let OPT* = (OPT \ $\{a_k\}$) U $\{a_i\}$.
Then OPT* includes a_i and size(OPT*) = size(OPT).
We have end(a_i) ≤ end(a_k) by definition of a_i, so a_i cannot overlap any activities in OPT \ $\{a_k\}$. Hence, OPT* is a valid solution. ■

And now the algorithm:

Algorithm *Greedy-Activity-Selection (A)*

1. **if** A is empty
2. **then return** A
3. **else** $a_i \leftarrow$ an activity from A ends first
4. $B \leftarrow$ all activities from A that do not overlap a_i
5. **return** $\{a_i\}$ U *Greedy-Activity-Selection (B)*

Correctness:

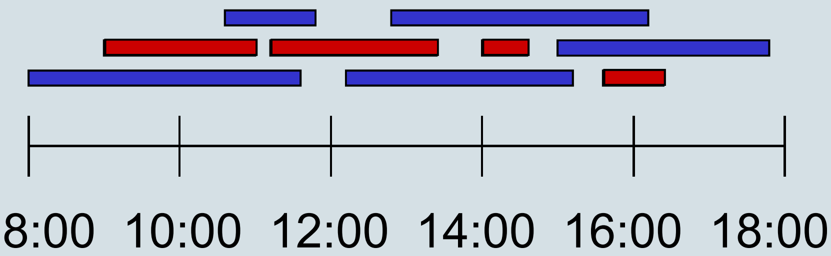
- by induction, using optimal substructure and greedy-choice property

Running time:

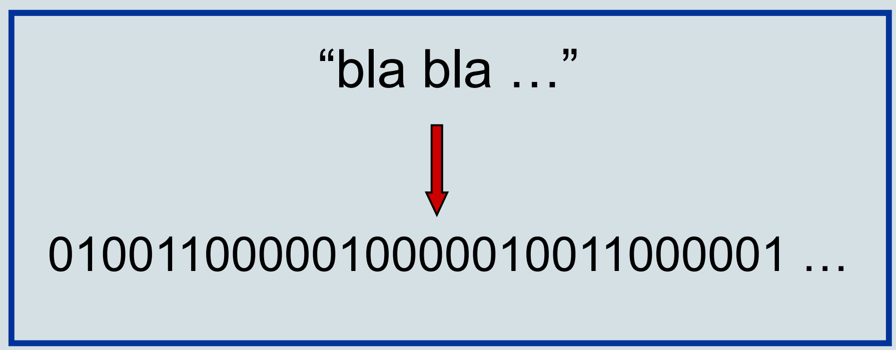
- $O(n^2)$ if implemented naively
- $O(n)$ after sorting on finishing time, if implemented more cleverly

Today: two examples of greedy algorithms

- Activity-Selection



- Optimal text encoding



Optimal text encoding

“bla□bla …”

↓

0100110000010000010011000001 …

Standard text encoding schemes: fixed number of bits per character

- ASCII: 7 bits (extended versions 8 bits)
- UCS-2 (Unicode): 16 bits

Can we do better using variable-length encoding?

Idea: give characters that occur frequently a short code and
give characters that do not occur frequently a longer code

The encoding problem

Input: set C of n characters $c_1, \dots c_n$; for each character c_i its frequency $f(c_i)$

Output: binary code for each character

~~code(c_1) = 01001, code (c_2) = 010,~~ … not a prefix-code

Variable length encoding: how do we know where characters end ?

text = 0100101100 … Does it start with c_1 = 01001 or c_2 = 010 or … ??

Use prefix-code: no character code is prefix of another character code

Variable-length prefix encoding: can it help?

Text: “een□voordeel”

Frequencies: f(e)=4, f(n)=1, f(v)=1, f(o)=2, f(r)=1, f(d)=1, f(l)=1, f(□)=1

fixed-length code:

e=000 n=001 v=010 0=011 r =100 d=101 l =110 □=111

length of encoded text: 12 x 3 = 36 bits

possible prefix code:

e=00 n=0110 v=0111 o=010 r =100 d=101 l=110 □=111

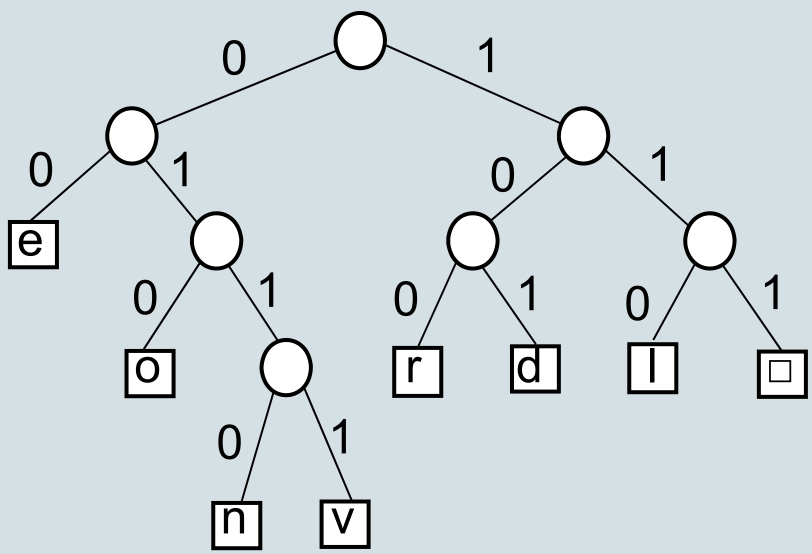
length of encoded text: 4x2 + 2x4 + 6x3 = 34 bits

Representing prefix codes

Text: “een□voordeel”

Frequencies: f(e)=4, f(n)=1, f(v)=1, f(o)=2, f(r)=1, f(d)=1, f(l)=1, f(□)=1

code: e=00 n=0110 v=0111 o=010 r =100 d=101 l=110 □=111



representation is binary tree *T*:

- one leaf for each character
- internal nodes always have two outgoing edges, labeled 0 and 1
- code of character: follow path to leaf and list bits

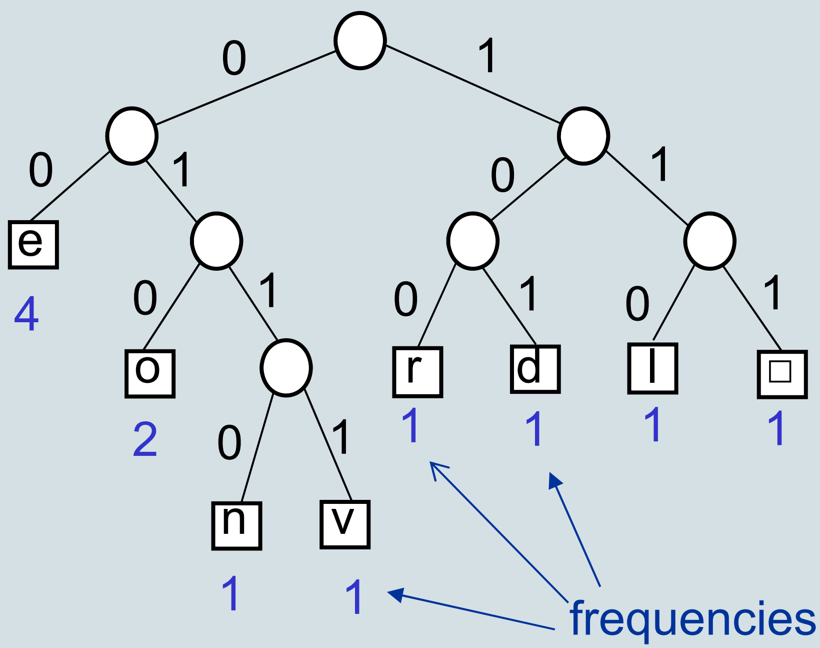
codes represented by such trees are exactly the “non-redundant” prefix codes

Representing prefix codes

Text: “een□voordeel”

Frequencies: f(e)=4, f(n)=1, f(v)=1, f(o)=2, f(r)=1, f(d)=1, f(l)=1, f(□)=1

code: e=00 n=0110 v=0111 o=010 r =100 d=101 l=110 □=111



cost of encoding represented by *T*:

$$\sum_i f(c_i) \cdot \text{depth}(c_i)$$

Designing greedy algorithms

1. try to discover structure of optimal solutions: what properties do optimal solutions have ?
   - what are the choices that need to be made ?
   - do we have optimal substructure ?

     optimal solution = first choice + optimal solution for subproblem
   - do we have greedy-choice property for the first choice ?

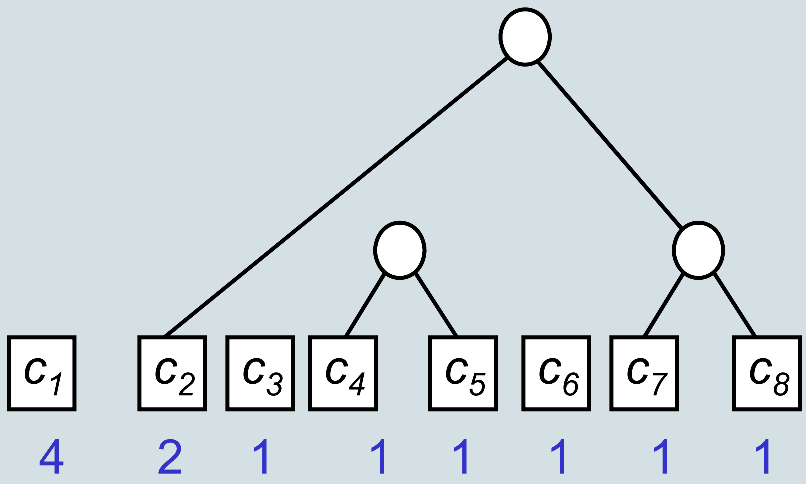
2. prove that optimal solutions indeed have these properties
   - prove optimal substructure and greedy-choice property

3. use these properties to design an algorithm and prove correctness
   - proof by induction (possible because optimal substructure)

Bottom-up contruction of tree:

start with separate leaves, and then “merge” *n-1* times until we have the tree

choices: which subtrees to merge at every step

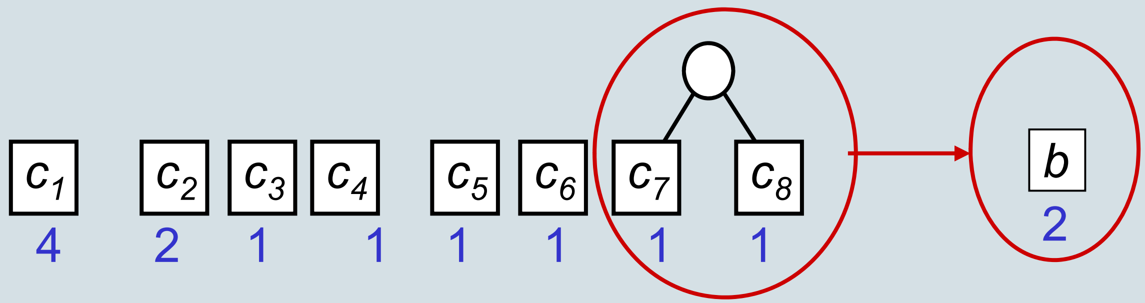


we do not have to merge adjacent leaves

Bottom-up contruction of tree:

start with separate leaves, and then “merge” *n-1* times until we have the tree

choices: which subtrees to merge at every step



Do we have optimal substructure?

Do we even have a problem of the same type?

Yes, we have a subproblem of the same type: after merging, replace merged leaves c_i, c_k by a single leaf b with $f(b) = f(c_i) + f(c_k)$

(other way of looking at it: problem is about merging weighted subtrees)

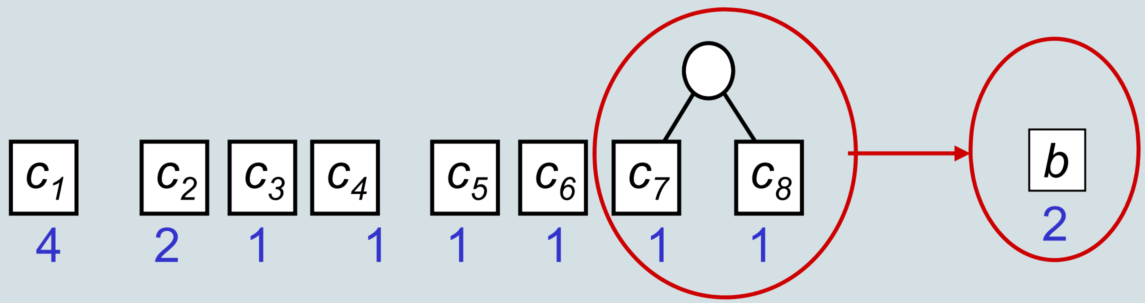
Lemma: Let c_i and c_k be siblings in an optimal tree for set C of characters.
Let $B = (C \setminus \{c_i , c_k \}) \cup \{b\}$, where $f(b) = f(c_i) + f(c_k)$.
Let T_B be an optimal tree for B.
Then replacing the leaf for b in T_B by an internal node with c_i , c_k as children results in an optimal tree for C.

Proof. Do yourself. ■

Bottom-up contruction of tree:

start with separate leaves, and then “merge” *n-1* times until we have the tree

choices: which subtrees to merge at every step



Do we have a greedy-choice property?

Which leaves should we merge first?

Greedy choice: first merge two leaves with smallest character frequency

Lemma: Let c_i, c_k be two characters with the lowest frequency in C.
Then there is an optimal tree T_{OPT} for C where c_i, c_k are siblings.

Proof. Let OPT be an optimal tree T_{OPT} for C. If c_i, c_k are siblings in T_{OPT} then the lemma obviously holds, so assume this is not the case.
We will show how to modify T_{OPT} into a tree T^* such that

(i) T^* is a valid tree
(ii) c_i, c_k are siblings in T^*
(iii) cost(T^*) ≤ cost(T_{OPT})

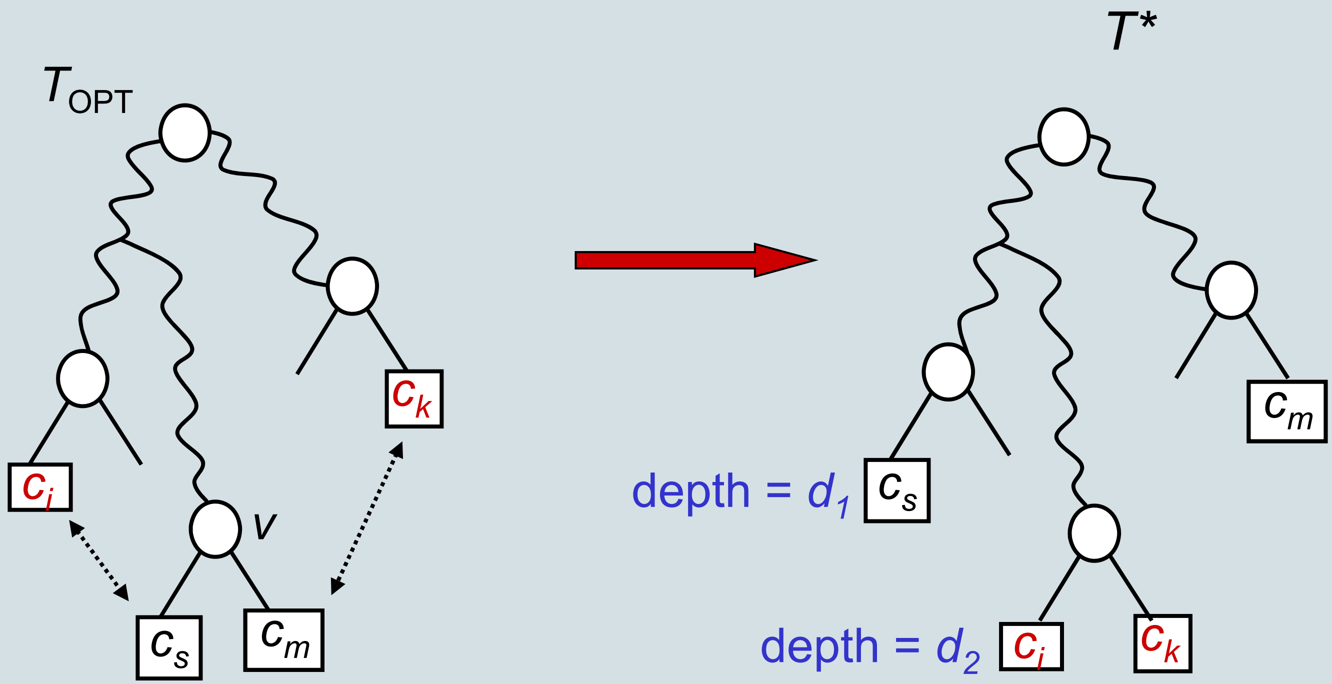
standard text you can basically use in proof for any greedy-choice property

Thus T^* is an optimal tree in which c_i, c_k are siblings, and so the lemma holds. To modify T_{OPT} we proceed as follows.

now we have to do the modification

■

How to modify T_{OPT} ?



- take a deepest internal node v
- make c_i, c_k children of v by swapping them with current children (if necessary)

change in cost due to swapping c_i and c_s

cost (T_{OPT}) – cost (T*)

$= \ f(c_s) \cdot (d_2 - d_1) + f(c_i) \cdot (d_1 - d_2)$

$= \ (f(c_s) - f(c_i)) \cdot (d_2 - d_1)$

$\geq \ 0$

Conclusion: T^* is valid tree where c_i, c_k are siblings and cost(T*) ≤ cost (T_{OPT}).

Algorithm *Construct-Huffman-Tree (C: set of n characters)*

```
if |C| = 1
    then return a tree consisting of single leaf, storing the character in C
    else  c_i , c_k ← two characters from C with lowest frequency
          Remove c_i , c_k from C, and replace them by a new character b
          with f(b) = f(c_i) + f(c_k). Let B denote the new set of characters.
          T_B ← Construct-Huffman-Tree(B)
          Replace leaf for b in T_B with internal node with c_i , c_k as children.
          Let T be the new tree.
          return T
```

Correctness:

- by induction, using optimal substructure and greedy-choice property

Running time:

- $O(n^2)$?!
- $O(n \log n)$ if implemented smartly (use heap)
- Sorting + $O(n)$ if implemented even smarter (hint: 2 queues)

Summary

- greedy algorithm: solves optimization problem by trying only one option for first choice (the greedy choice) and then solving subproblem recursively
- need: optimal substructure + greedy choice property
- proof of greedy-choice property: show that optimal solution can be modified such that it uses greedy choice