Advanced Algorithms

Lecture 4
Greedy Algorithms

Bin Yang
byang@cs.aau.dk

ILO of Lecture 4

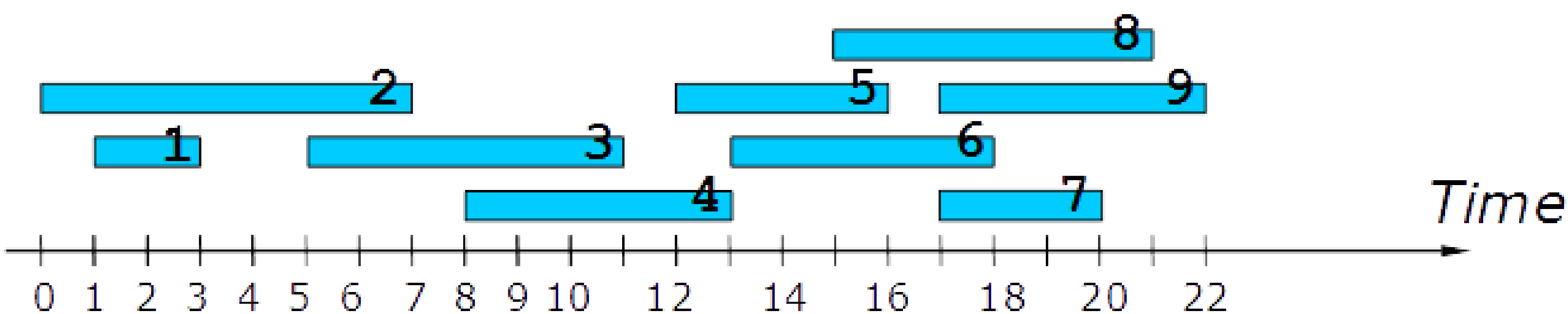
- Greedy algorithms
  - to understand the principles of the greedy algorithm design technique;
  - to understand two example greedy algorithms, for activity selection and Huffman coding, and to be able to prove that these algorithms find optimal solutions (correctness proof);
  - to be able to apply the greedy algorithm design technique.

Agenda

- Activity selection
- Huffman coding
- Principles of greedy algorithms
- Revisit some graph algorithms that use the greedy strategy

Activity Selection

- Input:
  - A set of n activities, each with start and end times: s_i and f_i. The i-th activity lasts during the period $[s_i, f_i)$.
- Output:
  - The ***largest*** subset of mutually *compatible* activities.
  - Activities are compatible if their intervals do not intersect.



- *Activities 1 and 2 are not compatible.*
- *Activities 2 and 4 are compatible.*

Activity Selection – Some Definitions

- Sort activities in A on the end time (for simplicity assume also "sentinel" activities a_0 and a_{n+1}.

0	i	1	2	3	4	5	6	7	8	9	10	11	12
-100	s_i	1	3	0	5	3	5	6	8	8	2	12	100
-100	f_i	4	5	6	7	9	9	10	11	12	14	16	100

- $S_{i,j}$: a set of activities that start after activity a_i finishes and that finish before activity a_j starts.
  - $S_{2,11=}$ $\{a_4, a_6, a_7, a_8, a_9\}$
    - Start after a_2.f=5 and finish before a_{11}.s=12
  - $S_{0,12=}$ $\{a_1, a_2, a_3, a_4, a_5 , a_6, a_7, a_8, a_9, a_{10}, a_{11}\}$
    - Start after a_0.f=-100 and finish before a_{12}.s=100
- $M_{i,j}$: a maximum set of mutually compatible activities in $S_{i,j}$.
- $C_{i,j}$: the cardinality of $M_{i,j}$
- Activity Selection: identify $C_{0,n+1}$ (and $M_{0,n+1}$)

Activity selection – DP solution

- Choose an activity a_k in $S_{i,j,}$ which splits $S_{i,j}$ into $S_{i,k}$ and $S_{k,j}$

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

  - $S_{2,11=}$ {a_4, a_6, a_7, a_8, a_9}
  - a_8, $S_{2,8=}$ {a_4} $S_{8,11=}$ {}
- The maximum number of compatible activities in $S_{i,j}$ *is the maximum of the sum of the following, over all possible a_k*
  - *maximum number of* compatible activities in $S_{i,k,}$ i.e., $C_{i,k}$
  - *maximum number of* compatible activities in $S_{k,j\ ,}$ i.e., $C_{k,j}$
  - *1, i.e., a_k itself*
- *Trivial sub-problems: 0 if* $S_{i,k}$ is empty.

1. Overlapping sub-problems.
2. Optimal sub-structures.

$$c[i,j]=\begin{cases}0 & \text{if } S_{ij}=\emptyset\,,\\ \max\limits_{a_k\in S_{ij}}\{c[i,k]+c[k,j]+1\} & \text{if } S_{ij}\neq\emptyset\,.\end{cases}$$

Algorithm, bottom-up

	0	1	2	3	4
0	**0**	**0**	c[0,2]	c[0,3]	c[0,4]
1	x	0	0	c[1,3]	c[1,4]
2	x	x	0	0	c[2,4]
3	x	x	x	0	0
4	x	x	x	x	0

How many sub-problems are there and how many choices do you need to consider for solving each sub-problem?

Recall Exercise 1 of Lecture 2.

Greedy strategy

- Given $S_{i,j}$, DP needs to consider ***every activity a_k*** in $S_{i,j}$ in order to identify the optimal solution.

$$c[i,j]=\begin{cases}0 & \text{if } S_{ij}=\emptyset,\\ \max\limits_{a_k\in S_{ij}}\{c[i,k]+c[k,j]+1\} & \text{if } S_{ij}\neq\emptyset.\end{cases}$$

  - For different $S_{i,j}$, there are different numbers of ***a_k*** in $S_{i,j}$

- Greedy strategy: what if we only considers ***"the best" (as of now)*** activity and be sure that it belongs to an optimal solution.
- Choose the activity that ***finishes first*** in $S_{i,j}$
  - Intuition: leave as much time as possible for other activities.
  - Then, solve *only one* sub-problem for the remaining compatible activities.

Greedy algorithm

- MaxN(A, i)
  - Assume that we have n activities in total.
  - Return the maximum-size set of mutually compatible activities in $S_{i,n+1}$
  - In the beginning, we call MaxN(A, 0) that returns the maximum-size set of mutually compatible activities in $S_{0,n+1}$

A[m] is the activity in $S_{i,n+1}$ that finishes first.

```
MaxN(A, i)
m ← i + 1
while m ≤ n and A[m].s < A[i].f do
    m ← m + 1
if m ≤ n then return {A[m]} ∪ MaxN(A, m)
          else return ∅
```

The found activity a_m that finishes first must belong to the maximum-size set of mutually compatible activities. Then, we only need to consider activities in $S_{m,n+1}$.

Example

A[0]	A[1]	A[2]	A[3]	*A[4]*
0	1	2	5	*10*
0	3	4	6	*10*

```
MaxN(A, i)
m ← i + 1
while m ≤ n and A[m].s < A[i].f do
    m ← m + 1
if m ≤ n then return {A[m]} ∪ MaxN(A, m)
              else return ∅
```

- MaxN(A, 0), A[1] is chosen, so $\{a_1\}$.
  - A[1] is the activity finishes the first from $S_{0, 4}$
- MaxN(A, 1), A[3] is chosen, so $\{a_1, a_3\}$.
  - A[3] is the activity finishes the first from $S_{1, 4}$
- MaxN(A, 3), nothing is chosen, so still $\{a_1, a_3\}$.
- {a1, a3} is the maximum-size set of mutually compatible activities.

Correctness?

- Why the activity that finishes first must be in the maximum-size set of mutually compatible activities?
  - Consider any nonempty sub-problem S_{ij}, and let a_x be an activity in S_{ij} with the earliest finish time.
  - Let M_{ij} be a maximum-size set of mutually compatible activities in S_{ij}. Let a_y be the activity in M_{ij} with the earliest finish time.
  - Lucky: if $a_x=a_y$, we have proved that a_x belongs to a maximum-size set of mutually compatible activities.
  - Unlucky: If not, by replacing a_y by a_x, M_{ij} is still a maximum-size set of mutually compatible activities.
    - $a_x.f <= a_y.f$

A[0]	A[1]	A[2]	A[3]	***A[4]***
0	1	2	5	***10***
0	3	4	6	***10***

  - $M_{0,4}$={a2, a3}, replacing a2 by a1, all activities in {a1, a3} are still compatible, and thus it is still a maximum-size set.

Greedy exchange

- It is a different proof technique compared to contradiction or induction.
- Greedy exchange is often used in proving the correctness of greedy algorithms.
- Assume that we already have an optimal solution that is produced by any other optimal algorithm.
  - M_{ij} in our previous proof.
- We show that it is possible to incrementally modify the optimal solution into the solution produced by our greedy algorithm in such a way that does not worsen the solution’s quality.
  - Replace a_y by a_x, still compatible and with the same cardinality.
- Thus, the quality of our greedy solution is at least as small as that of any other optimal solution.

Greedy choice property

- We can assemble a globally optimal solution by making locally optimal (greedy) choices.
  - We need to prove that there is always an optimal solution to the original problem that includes the greedy choice, so that the greedy choice is always safe.

- The challenge is to choose the right interpretation of “the best choice”:
  - Mini quiz: counter-example or proof
  - How about the activity that starts first?
  - The shortest activity?
  - The activity that overlaps the smallest number of the remaining activities?

Greedy choice property

- How about the activity that starts first?

a1	a2	a3
1	2	4
10	3	6

  - {a2, a3}, but not a1 that starts first.

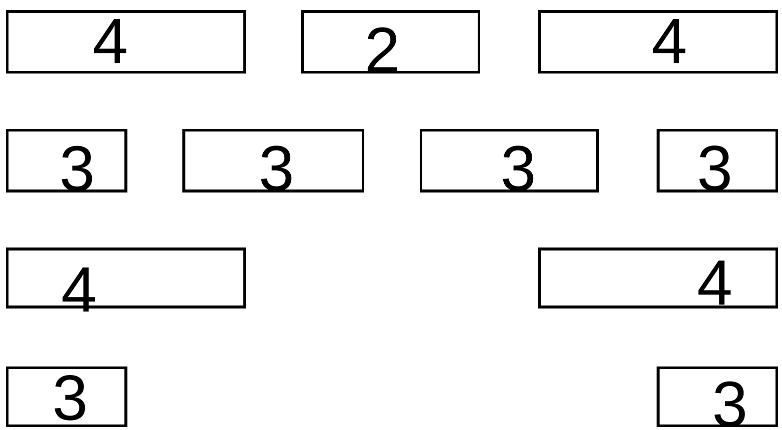
- The shortest activity?

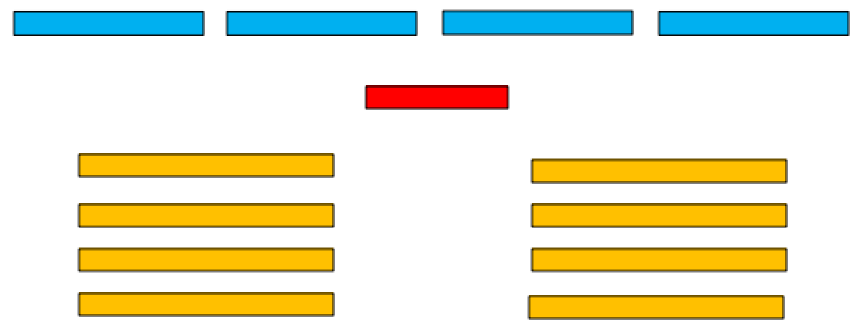
a1	a2	a3	a4
1	11	21	9
10	20	30	12

  - {a1, a2, a3}, but not a4 that is the shortest activity.

Greedy choice property

- The activity that overlaps the smallest number of the remaining activities?





- The second row gives the maximum-size set of mutually compatible activities, but it does not include the activity with the smallest overlaps, i.e., the one with 2.

Run time of the greedy algorithm

```
MaxN(A, i)
m ← i + 1
while m ≤ n and A[m].s < A[i].f do
    m ← m + 1
if m ≤ n then return {A[m]} ∪ MaxN(A, m)
         else return ∅
```

Assume that the activities in A have been ordered according to the finishing time already.

Intuition: each activity is examined once, and thus $\theta(n)$.

Still remember the run time of DP?
Exercise 1 of lecture 2: $\theta(n^3)$.

First self-study exercises

- A self-study exercise session = 4 hours of exercises
  - You need to do it in groups.
  - Each group can submit to Simon/me one written solution ***no later than a week*** of the session.
  - Simon/Bin will give written feedback for each of the submitted solutions.
  - 1st and 2nd self-study exercises: to Simon.
  - 3rd self-study exercises: to Bin.

  - I recommend that each of you solves the problems individually first, and then you discuss, summarize, and hand in solutions per group.
  - In case you cannot agree with each other, you can hand in multiple solutions for one problem.

Agenda

- Activity selection
- Huffman coding
- Principles of greedy algorithms
- Revisit some graph algorithms that use the greedy strategy

Data coding and compression

- Suppose we have 100,000-character data file that we wish to store compactly, i.e., using the least space.
- The file only has 6 distinct characters.
- Each character has different frequencies.

	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

- Fixed-length codeword
  - (45K+13K+12K+16K+9K+5K)*3=300K bits
- Variable-length codeword
  - 45K*1+(13K+12K+16K)*3+(9K+5K)*4=224K bits
- 224/300≈75%, we can save 25% of space by using variable-length codeword.

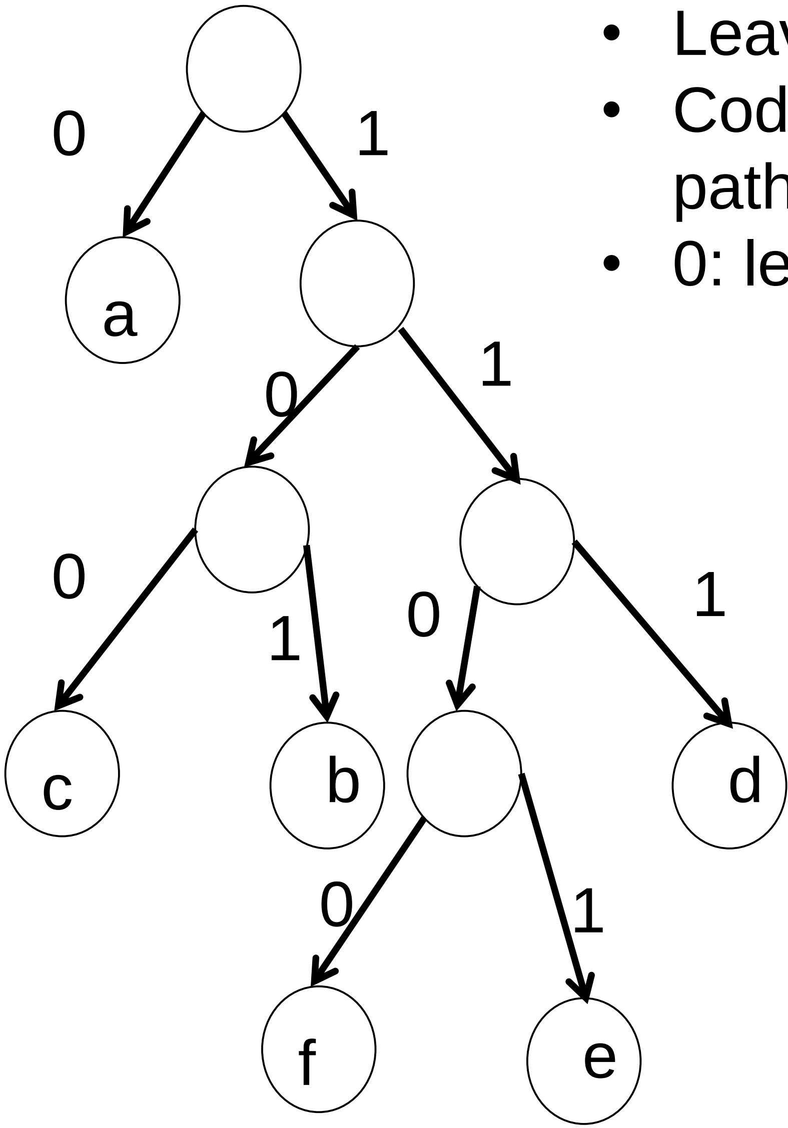
Prefix codes

- What is a prefix code?
  - No codeword is also a prefix of some other codeword.

	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

  - Prefix codes can always achieve the optimal data compression among any character code.
  - Prefix codes are desired because they simplify decoding.
  - From now on, we only consider prefix codes.
- Encoding: concatenate the codewords representing the characters in the file.
  - abc: 000001010 or 0101100
- Decoding:
  - 000000001100 = aabe
  - 001011101 = aabe

Decoding using a binary tree



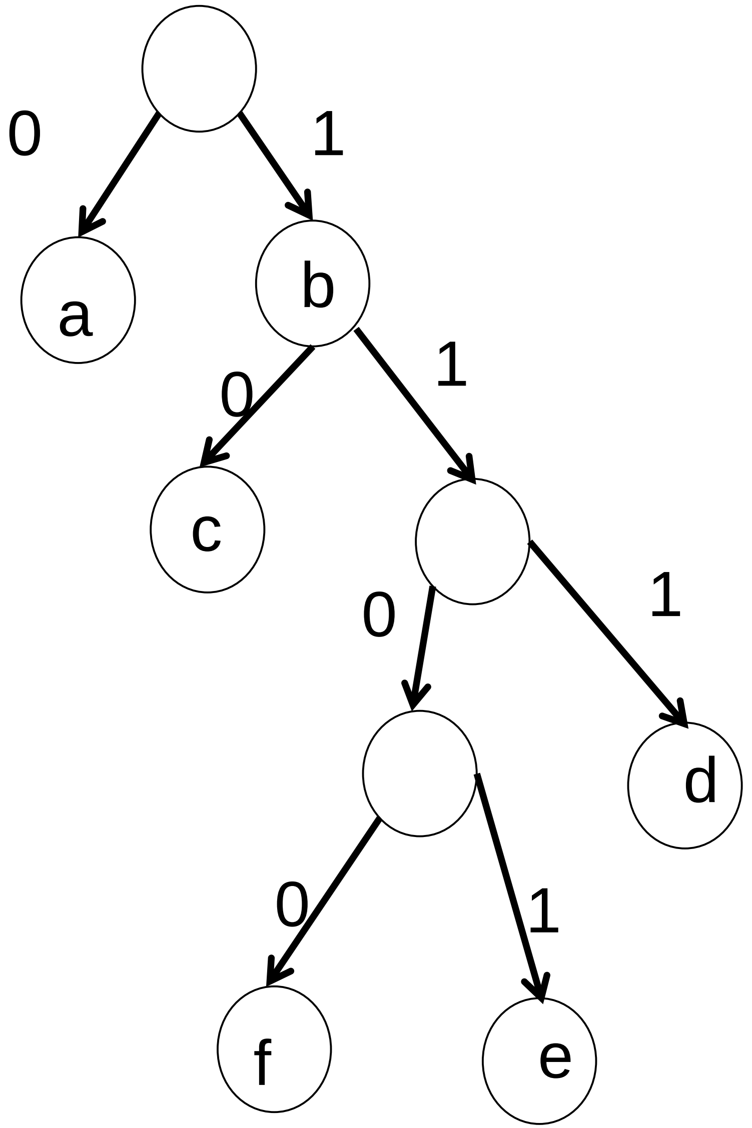
- Leaves represent characters.
- Codeword for a character is the simple path from the root to that character.
- 0: left 1:right

Decode: 001011101

aabe

Mini quiz: can you think about some non-prefix codes?

Decoding using a binary tree



Can you think about some non-prefix codes?

Can you decode: 010?

Should it be aba or ac?

Optimal code

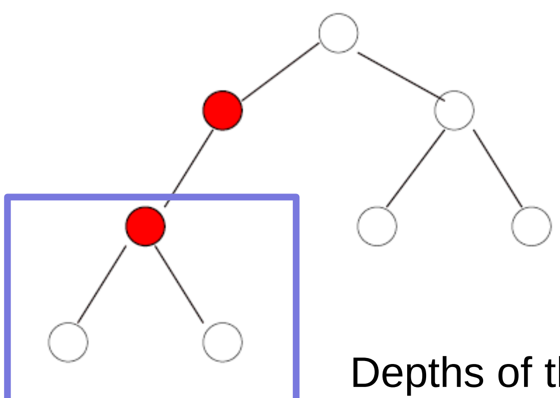
- The number of bits required to encode a file is

$$B(T) = \sum_{c \in C} \boxed{c.freq} \cdot \boxed{d_T(c)}$$

Depth of character c's corresponding leaf node in the binary tree.

Frequency of character c

- ***Optimal code*** achieves minimal B(T).
- Optimal code is always represented by a ***full*** binary tree.
  - Every non-leaf node must have two children.



If a non-leaf node has only one child, we can replace the non-leaf node with its unique child. This would decrease the total bits of the encoding.

Depths of the two characters in this sub-tree decrease.

Huffman code

- Huffman code is an optimal prefix code.
- Basic idea
  - Initially, one separate node for each character.
  - In each step, join two nodes with the least frequencies, and merge into a new node whose frequency is the sum of the corresponding two nodes.
  - Repeat until all nodes are joined into one tree.

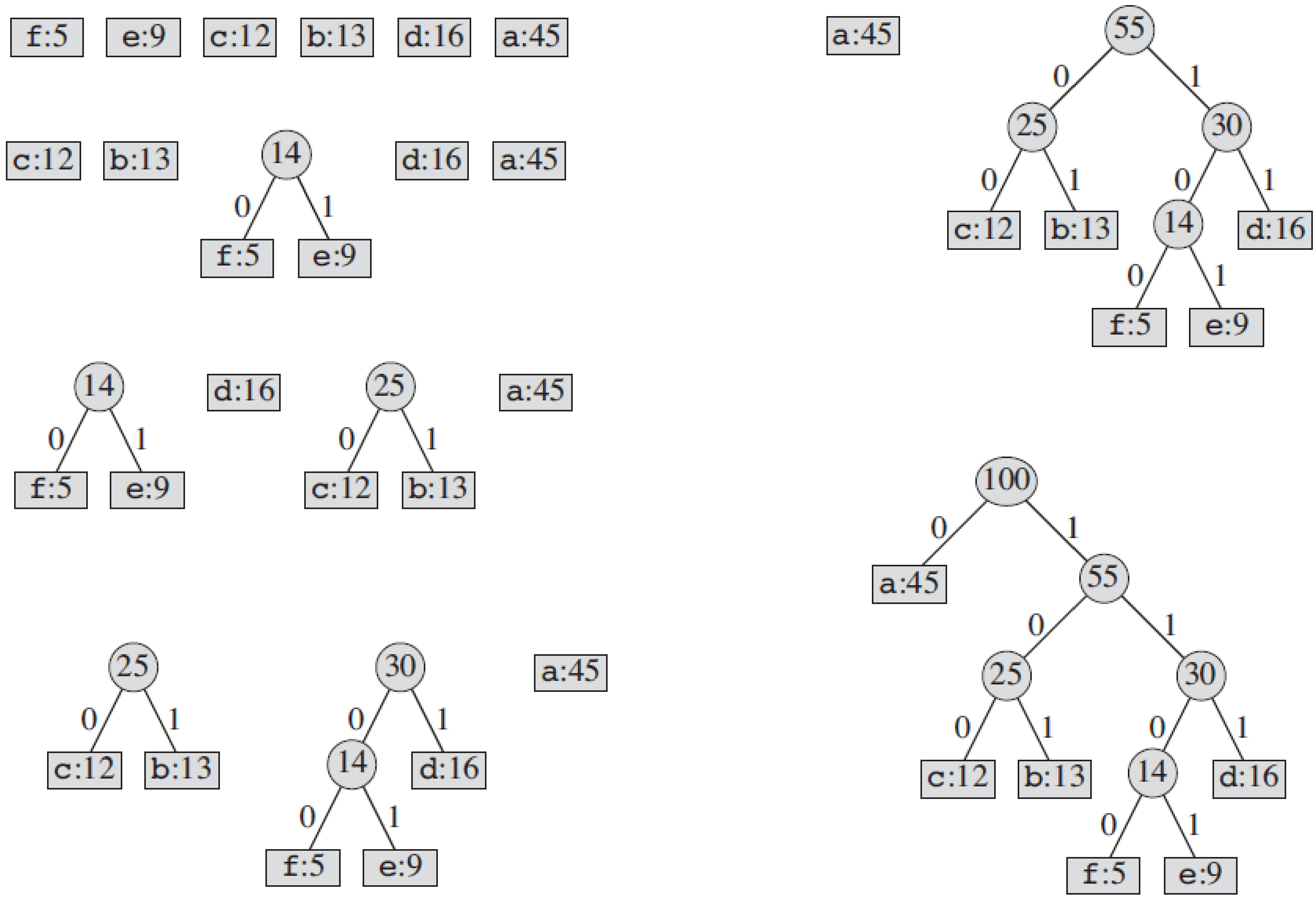
```
HUFFMAN(C)
n = |C|
Q = C
for i = 1 to n - 1
    allocate a new node z
    z.left = x = EXTRACT-MIN(Q)
    z.right = y = EXTRACT-MIN(Q)
    z.freq = x.freq + y.freq
    INSERT(Q, z)
return EXTRACT-MIN(Q)    // return the root of the tree
```

Input C is a set of characters, each character c∈ C is with an attribute c.freq that shows the frequency of c.

Insert all characters into a priority queue w.r.t. frequency

Find the two nodes with the least frequencies, and join them into one node.

Example



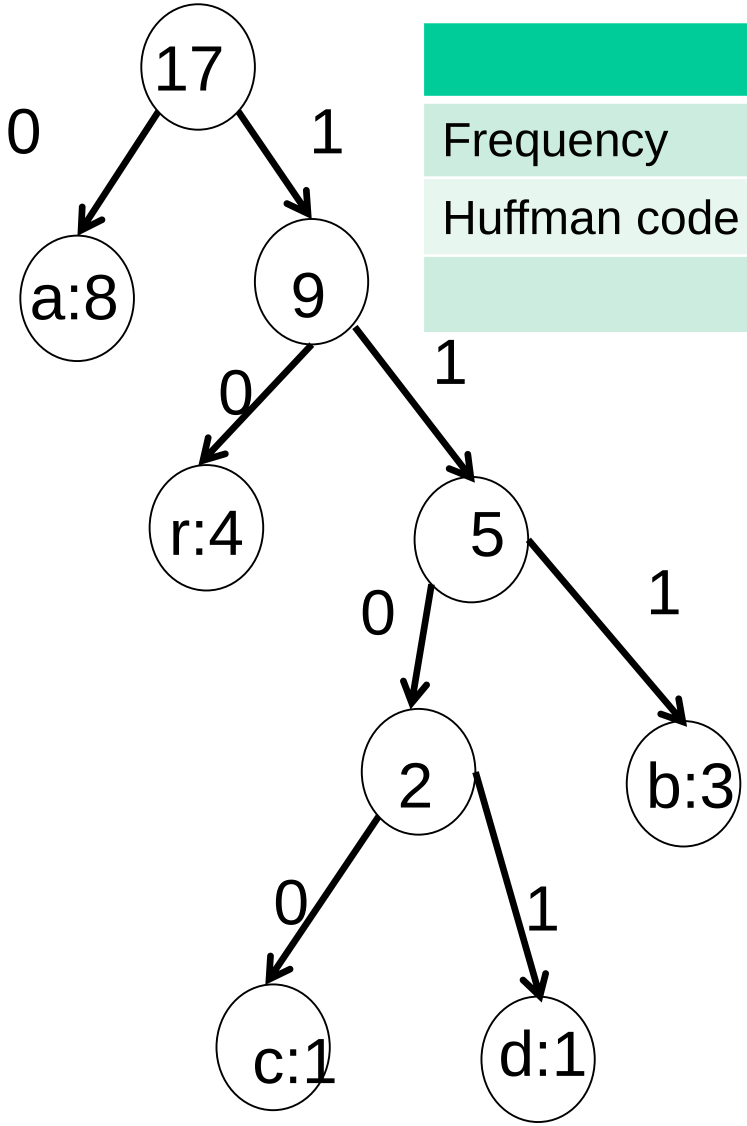
f:5 e:9 c:12 b:13 d:16 a:45
c:12 b:13 14 f:5 e:9 d:16 a:45
14 f:5 e:9 d:16 25 c:12 b:13 a:45
25 c:12 b:13 30 14 f:5 e:9 d:16 a:45
a:45 55 25 c:12 b:13 30 14 f:5 e:9 d:16
100 a:45 55 25 c:12 b:13 30 14 f:5 e:9 d:16
0
1

Mini quiz (also on Moodle)

- Identify the Huffman code for the following table using the algorithm we just saw.

	a	b	c	d	r
Frequency	8	3	1	1	4

- Then, write done the code for
  - ab
  - rc



	a	b	c	d	r
Frequency	8	3	1	1	4
Huffman code	0	111	1100	1101	10
	0	111	1101	1100	10

ab: 0111

rc: 101100 or 101101

Run time

Assuming that we use a binary heap to implement the priority queue Q here.

```
HUFFMAN(C)
    n = |C|
    Q = C
    for i = 1 to n − 1
        allocate a new node z
        z.left = x = EXTRACT-MIN(Q)
        z.right = y = EXTRACT-MIN(Q)
        z.freq = x.freq + y.freq
        INSERT(Q, z)
    return EXTRACT-MIN(Q)    // return the root of the tree
```

Initialize a priority queue with n elements: O(n)

All operations here in a priority queue is O(lgn)

In total, n-1 iterations.
O(nlgn)

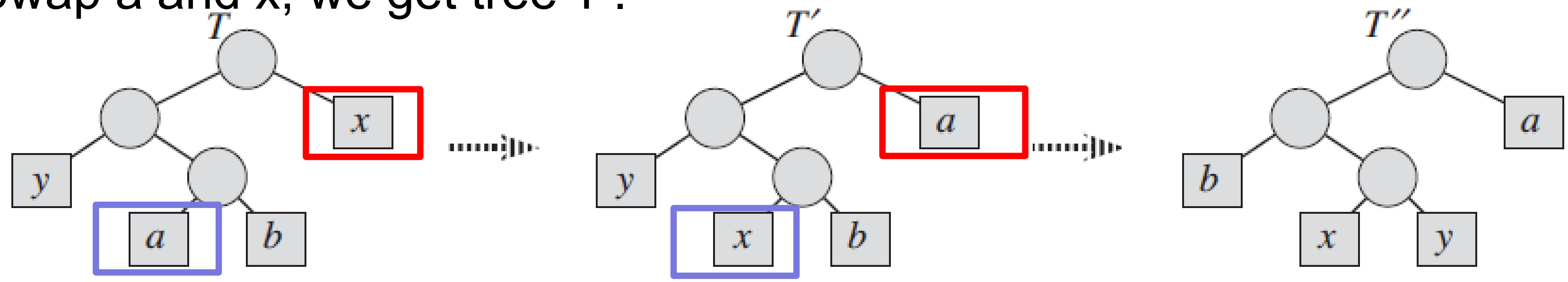
What if we use an ordered/unordered linked list to implement the priority Q? What is the run time then?

Correctness of Huffman code

- Greedy choice property and optimal substructure
- Greedy choice property
  - Let x and y be the two characters with lowest frequencies.
  - We need to prove that there exists an optimal prefix code where the codewords for x and y have the *same length* and *differ only in the last bit*.
  - We need to prove this greedy choice property.
  - Still use the "greedy exchange" proof technique.
    - Assume that we already have an optimal solution, tree **T**, that is produced by any other optimal algorithm.
    - We show that it is possible to incrementally modify the optimal solution T into the solution produced by our greedy algorithm, tree **T''**, in such a way that does not worsen the solution's quality.

Greedy choice property

- Let x and y be the two characters with lowest frequencies.
- Let's assume that we have an optimal code tree T, where leaves a and b are two siblings of the maximum depth.
- Swap a and x, we get tree T'.



$$
\begin{aligned}
B(T) - B(T') \\
&= \sum_{c \in C} c.freq \cdot d_T(c) - \sum_{c \in C} c.freq \cdot d_{T'}(c) \\
&= x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot \boxed{d_{T'}(x)} - a.freq \cdot \boxed{d_{T'}(a)} \\
&= x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot \boxed{d_T(a)} - a.freq \cdot \boxed{d_T(x)} \\
&= \boxed{(a.freq - x.freq)}\boxed{(d_T(a) - d_T(x))} \\
&\geq 0 ,
\end{aligned}
$$

$B(T) \geq B(T')$

Since x and y are the two characters with lowest frequencies, we have **$x.freq \leq a.freq$**

In tree T, a and b are two siblings of maximum depth. Thus, we have **$d_T(a) \geq d_T(x)$**

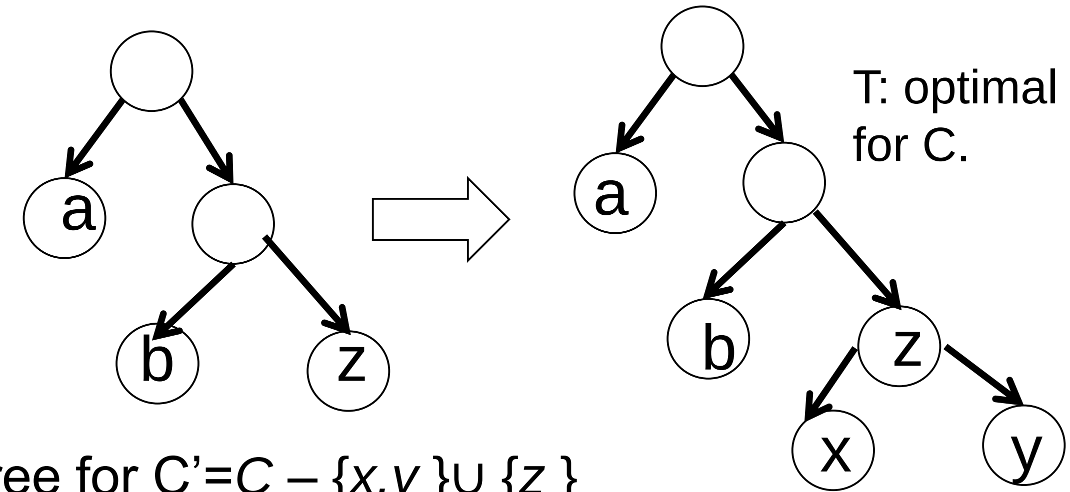
Greedy choice property

- Similarly, we can show B(T')≥ B(T'').
- Then, B(T) ≥ B(T') ≥ B(T'').
- Recall our assumption that T is an optimal code tree, i.e., B(T)≤B(T'').
- Then, B(T)=B(T'')
- Thus, T'' is also an optimal code tree.

Correctness of Huffman code (2)

Proof: Lemma 16.3 CLRS
Also slides on Moodle.

- Optimal-substructure property
  - What is the sub-problem?
  - Every time, we have one less character/node.
- Formally, we have
  - Let *x, y* – characters with minimum frequency
  - $C' = C - \{x,y\} \cup \{z\}$, such that $z.freq = x.freq + y.freq$
  - **Let *T'* be an optimal tree for *C'***
  - Replace leaf *z* in *T'* with internal node with two children *x* and *y*
  - The resulting tree *T* is an optimal tree for *C*

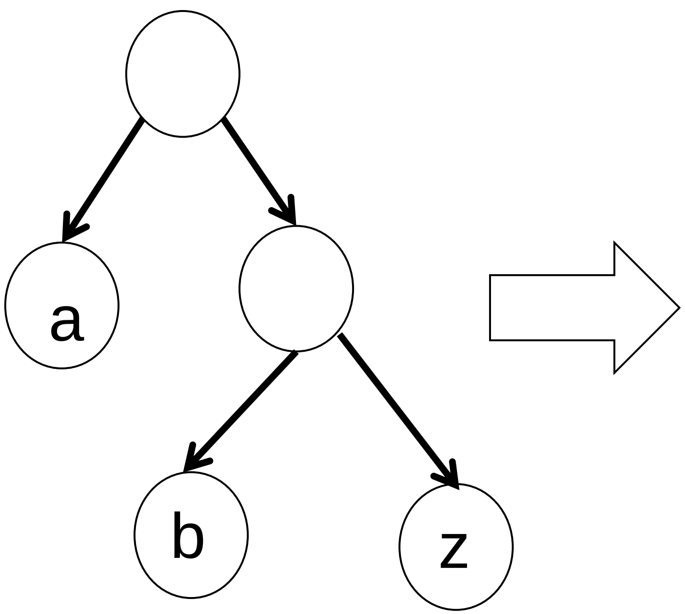


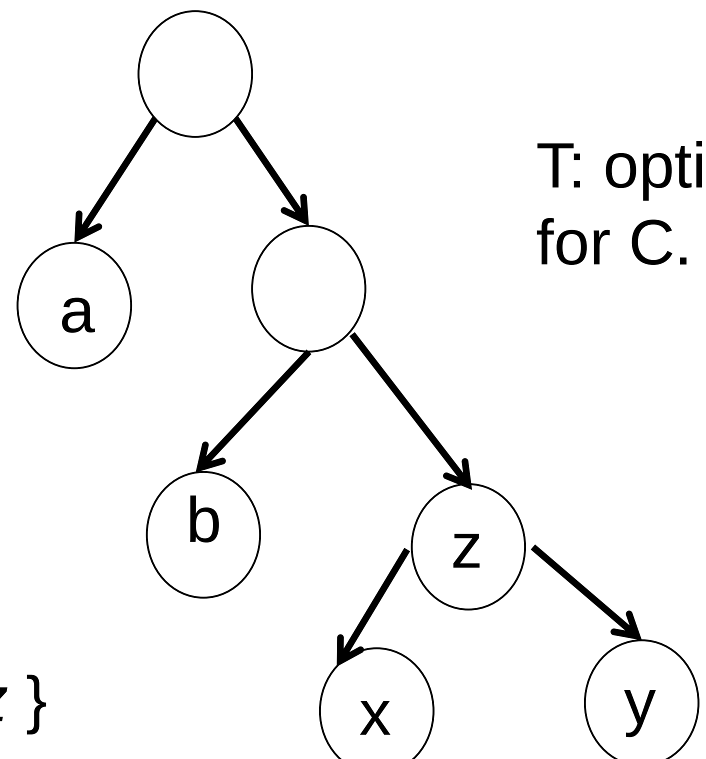
Solution to a sub-problem with character set C'.

T': Optimal tree for $C'=C - \{x,y\} \cup \{z\}$

Optimal-substructure property

- Let *T'* be an optimal tree for *C'*
- Replace leaf *z* in *T'* with internal node with two children *x* and *y* to get T.
- For each c ∈ *C*-{*x*,*y* }, we have $d_T(c)$=$d_{T'}(c)$, and thus
  - $c.freq * d_T(c) = c.freq * d_{T'}(c)$ //e.g., a, b
  - Let's call this **conclusion 1**.



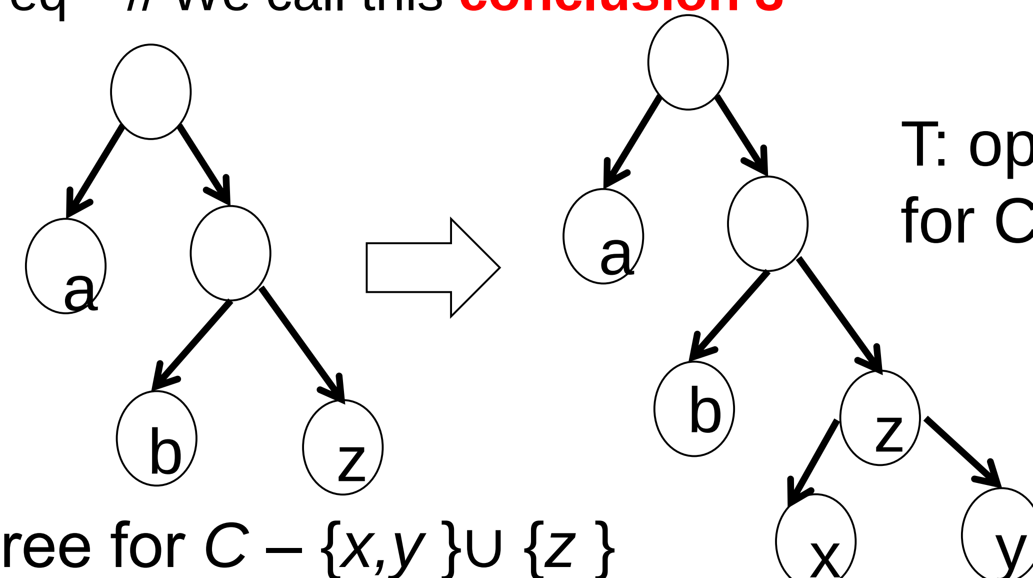


T: optimal tree for C.

T': Optimal tree for *C* – {*x*,*y* }∪ {*z* }

Optimal-substructure property

- For x, y, and z, we have $d_T(x)=d_T(y)=d_{T'}(z)+1$ and z.freq=x.freq+y.freq
  - $x.freq*d_T(x)+y.freq*d_T(y)=(x.freq+y.freq)*d_T(x)=(x.freq+y.freq)*(d_{T'}(z)+1)$
  - $=z.freq*d_{T'}(z)+(x.freq+y.freq)$ //We call this **conclusion 2**
- $B(T)=\sum_{c\in C} c.freq * d_T(c)$ // using the definition of B(T)
  - $=\sum_{c\in C-\{x,y\}} c.freq * d_T(c) + x.freq*d_T(x)+y.freq*d_T(y)$
  - $=\sum_{c\in C-\{x,y\}} c.freq * d_T(c) + z.freq*d_{T'}(z)+(x.freq+y.freq)$ // using conclusion 2
  - $=\sum_{c\in C-\{x,y\}} c.freq * d_{T'}(c) + z.freq*d_{T'}(z)+(x.freq+y.freq)$ // using conclusion 1
  - = B(T')+(x.freq+y.freq) // using the definition of B(T')
- B(T')=B(T)-x.freq-y.freq // We call this **conclusion 3**

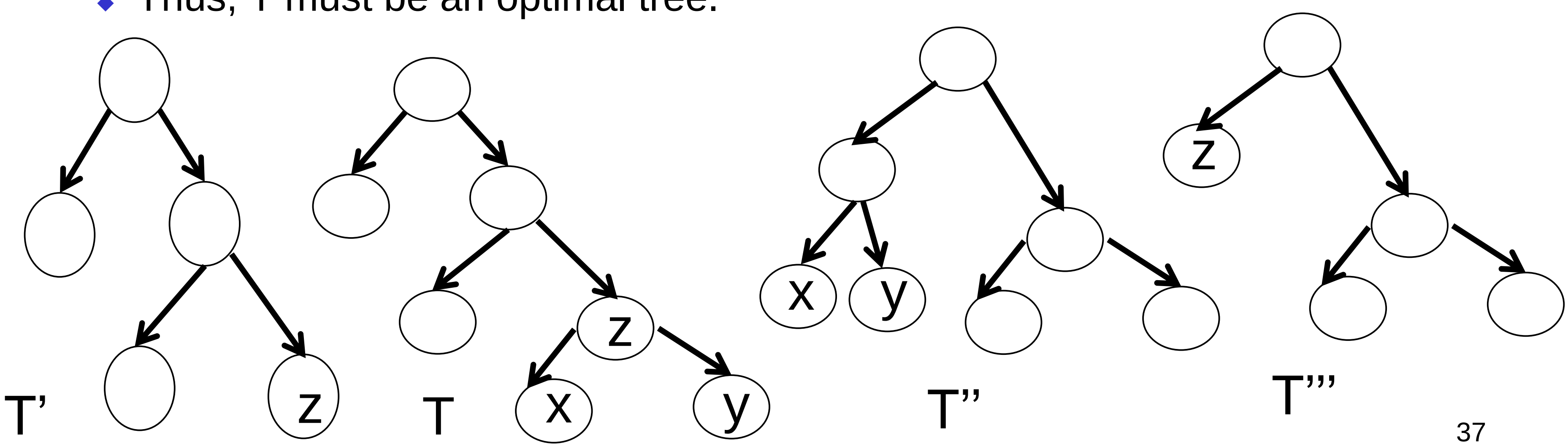


T: optimal tree for C.

T': Optimal tree for $C - \{x,y\} \cup \{z\}$

Optimal-substructure property

- Proof by contradiction:
  - Assume that T is not an optimal tree, we must have another tree T’’ that B(T’’)<B(T).
  - Previously, we have shown that an optimal tree T’’ has x and y as siblings.
  - Let T’’’ be the tree T’’ with the common parent of x and y replaced by a leaf z with z.freq=x.freq+y.freq, then
  - B(T’’’)=B(T’’)-x.freq-y.freq // using conclusion 3
  - <B(T) -x.freq-y.freq //due to the assumption B(T’’)<B(T)
  - =B(T’) // using conclusion 3 again.
  - B(T’’’)< B(T’) conflicts that T’ is an optimal tree for $C' = C - \{x, y\} \cup \{z\}$.
  - Thus, T must be an optimal tree.



Agenda

- Activity selection
- Huffman coding
- Principles of greedy algorithms
- Revisit some graph algorithms that use the greedy strategy

Principles of Greedy Algorithms

- Greedy algorithms are used for solving optimization problem
  - A number of choices have to be made to arrive at an optimal solution.
  - At each step, make the greedy “locally best” choice, without considering all possible choices and solutions to subproblems induced by these choices (compare to dynamic programming).
  - After the choice, only one sub-problem remains (smaller than the original).

- Greedy algorithms usually sort or use priority queues.

Principles of Greedy Algorithms

- First, we need to show the *optimal sub-structure* property
  - The same with DP.
- The main challenge is to decide the interpretation of “the best” so that it leads to a global optimal solution, i.e., proving the *greedy choice property*
  - Or you find counter-examples demonstrating that your greedy choice does not lead to a global optimal solution.
- *Greedy exchange* is a useful proof technique for proving the greedy choice property.

Agenda

- Activity selection
- Huffman coding
- Principles of greedy algorithms
- Revisit some graph algorithms that use the greedy strategy

Minimum spanning tree

- A spanning tree of a connected, undirected graph G is a sub-graph of G, which is
  - A tree (connected, undirected graph without cycles)
  - Contains all vertices of G.
- MST of graph G is a spanning tree T that minimizes w(T)=$\sum_{(u,v)\in T} w(u,v)$ for all possible spanning trees.
- It is an optimization problem:
  - There are many spanning trees
  - We want to find the MST that is a spanning tree with the least sum of weights of the edges in the spanning tree.
- Prim's algorithm and Kruskal's algorithm
- A generic algorithm

Prim’s algorithm

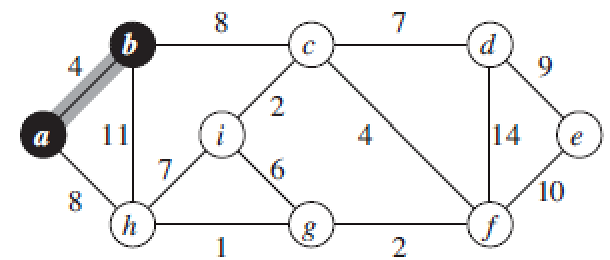
```
MST-Prim(G,r)
for each vertex u ∈ G.V()
    u.setkey(∞)
    u.setparent(NIL)
r.setkey(0)
Q.init(G.V())    // Q is a priority queue ADT
while not Q.isEmpty()
    u ← Q.extractMin()    // making u part of T
    for each v ∈ u.adjacent() do
        if v ∈ Q and G.w(u,v) < v.key() then
            v.setkey(G.w(u,v))
            Q.modifyKey(v)
            v.setparent(u)
```

Initialize all vertices

Update the keys and also maintain the priority queue according to the updated keys.

Greedy strategy for Prim's alg

- A weighted graph $G = (V, E)$ and a starting vertex s. Find a minimum spanning tree of G with root s.
- *Greedy choice*: Among all incident edges of *s*, choose an edge (*s, u*) with a minimum weight.
- Remaining sub-problem:
  - Consider a new graph G'=(V', E')
  - V'=V-{s, u}+{s'}
  - E'=E-{(s, u)}, but with all the edges incident on s or u made incident on s' (supervertex).
  - If there are both edges (*s, v*) and (*u, v*) in *E*, the weight of the corresponding new edge (*s', v*) in *E'* is $w(s', v) = \min(w(s, v), w(u, v))$.
  - Find minimum spanning tree of G' from *s'*.



ILO of Lecture 4

- Greedy algorithms
  - to understand the principles of the greedy algorithm design technique;
  - to understand the example greedy algorithms for activity selection and Huffman coding, to be able to prove that these algorithms find optimal solutions;
  - to be able to apply the greedy algorithm design technique.

Intended Learning Outcomes (ILO)

- After taking this course, you should acquire the following knowledge
  - Algorithm **design** techniques such as divide-and-conquer, greedy algorithms, dynamic programming, back-tracking, branch-and-bound algorithms, and plane-sweep algorithms;
  - Algorithm **analysis** techniques such as recursion, amortized analysis;
  - A collection of **core** algorithms and data structures to solve a number problems from various computer science areas: algorithms for external memory, multiple-threaded algorithms, advanced graph algorithms, heuristic search and geometric calculations;
  - There will also enter into one or more **optional subjects** in advanced algorithms, including, but not limited to: *approximate algorithms*, randomized algorithms, search for text, linear programming and number theoretic algorithms such as cryptosystems.

Lecture 5

- Amortized analysis
  - to understand what is amortized analysis, when is it used, and how it differs from the average-case analysis;
  - to be able to apply the techniques of the aggregate analysis, the accounting method, and the potential method to analyze operations on simple data structures.