Analysis of Algorithms
CECS 528

Midterm 2. Review

Framework of the exam

- Out of 50 points, 90 min long
- Starts at 9:50am Oct. 21 after a review and consultations
- Covers the following topics:
  - Dynamic programming – concepts
  - Dynamic programming – applications (Rod cutting, Longest common subsequence, 0-1 Knapsack problem, Assembly line scheduling)
  - Greedy algorithms – concepts
  - Greedy algorithms – applications (Activity selection, Minimum spanning trees, Huffman coding, Fractional knapsack)
- Solutions and explanations

Dynamic programming – concepts

Summary of concept

$$v_n(s_n) = \text{Max}\{f_n(d_n, s_n) + v_{n-1}[t_n(d_n, s_n)]\},$$

$$d_n \in D_n.$$

A formal statement of the *principal of optimality*:

"..an optimal sequence of decisions for a multistage problem has the property that, regardless of the current decision *dn* and current state *sn*, all subsequent decisions must be optimal, given the state *sn*−1 resulting from the current decision. "

Example 1

```
CUT-ROD(p, n)
1  if n == 0
2      return 0
3  q = -∞
4  for i = 1 to n
5      q = max(q, p[i] + CUT-ROD(p, n - i))
6  return q
```

Example 2

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise.} \end{cases}$$

Example 3

Let $f_i[j]$ = fastest time to get through $S_{i,j}$, $i = 1, 2$ and $j = 1, \ldots, n$.

Goal: fastest time to get all the way through = f^*.

$$
\begin{aligned}
f^* &= \min(f_1[n] + x_1,\ f_2[n] + x_2) \\
f_1[1] &= e_1 + a_{1,1} \\
f_2[1] &= e_2 + a_{2,1}
\end{aligned}
$$

For $j = 2, \ldots, n$:

$$
\begin{aligned}
f_1[j] &= \min(f_1[j-1] + a_{1,j},\ f_2[j-1] + t_{2,j-1} + a_{1,j}) \\
f_2[j] &= \min(f_2[j-1] + a_{2,j},\ f_1[j-1] + t_{1,j-1} + a_{2,j})
\end{aligned}
$$

$f_i[j]$ gives the *value* of an optimal solution. What if we want to *construct* an optimal solution?

Dynamic Programming – general characteristic

- *N* stage optimization problem
- Development of a *recursive optimization procedure*
- First solving a one-stage problem and sequentially including one stage at a time
- Solving one-stage problems until the overall optimum has been found
- Based on a *backward induction process:*
  - the first stage to be analyzed is the final stage of the problem and
  - problems are solved moving back one stage at a time until all stages are included

Dynamic Programming – general characteristic

- Alternatively, the recursive procedure can be based on a *forward induction process:*
  - the first stage to be solved is the initial stage of the problem
  - problems are solved moving forward one stage at a time, until all stages are included.
- In certain problem settings, only one of these induction processes can be applied (e.g., only backward induction is allowed in most problems involving uncertainties).

Principle of optimality

An optimal policy has the following property:

Whatever the current state and decision, the remaining decisions must constitute an optimal policy with regard to the state resulting from the current decision.

Proposed by Richard Bellman in 1957.

Applied to control theory, mathematical economics, machine learning, and others.

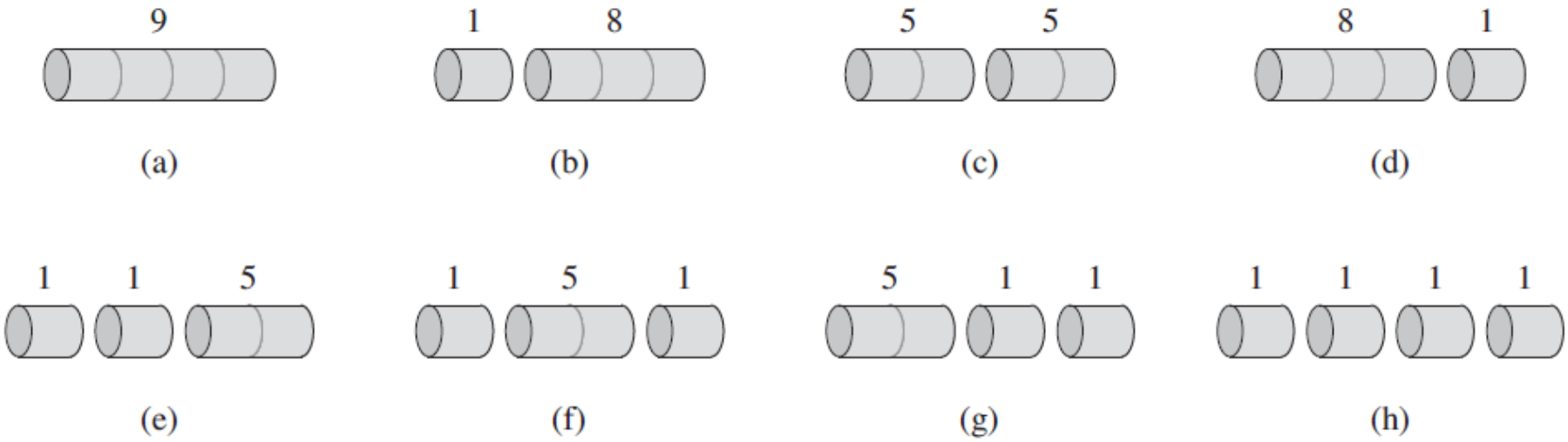
Dynamic programming – applications

Rod-cutting problem

Given a rod of length n inches and a table of prices p_i for $i=1, 2, \ldots, n$, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces. Note that if the price p_n for a rod of length n is large enough, an optimal solution may require no cutting at all.

Example

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30



The 8 possible ways of cutting up a rod of length 4. Above each piece is the value of that piece. The optimal strategy is part (c)—cutting the rod into two pieces of length 2—which has total value 10.

Recursive top-down implemenation

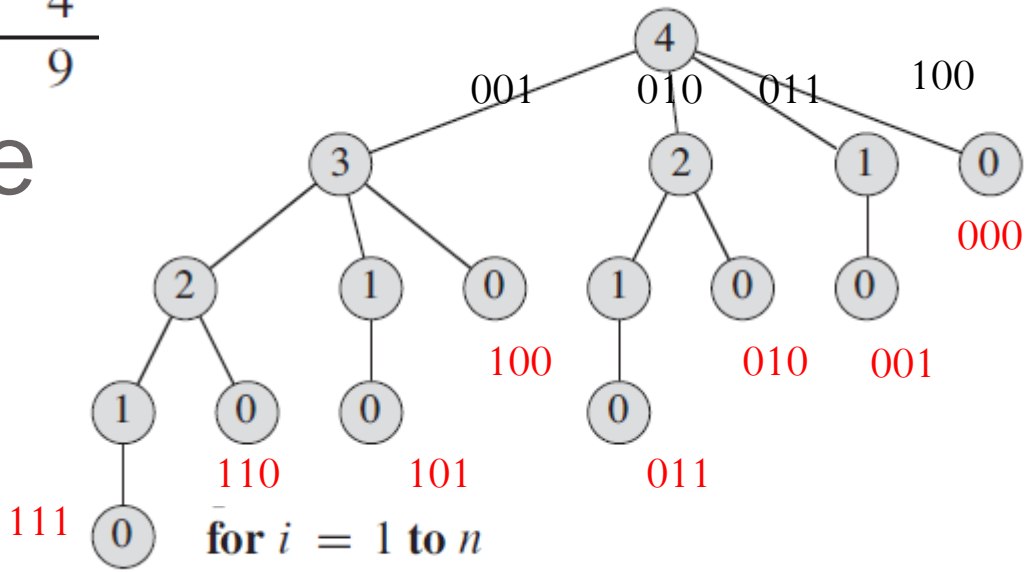
```
CUT-ROD(p, n)
1  if n == 0
2      return 0
3  q = -∞
4  for i = 1 to n
5      q = max(q, p[i] + CUT-ROD(p, n - i))
6  return q
```

Procedure CUT-ROD takes as input an array $p[1..n]$ of prices and an integer n, and it returns the maximum revenue possible for a rod of length n. If $n = 0$, no revenue is possible, and so CUT-ROD returns 0 in line 2. Line 3 initializes the maximum revenue q to $-\infty$, so that the **for** loop in lines 4–5 correctly computes $q = \max_{1 \le i \le n}(p_i + \text{CUT-ROD}(p, n - i))$; line 6 then returns this value. A simple induction on n proves that this answer is equal to the desired answer r_n, using

$$r_n = \max_{1 \le i \le n} (p_i + r_{n-i}) \ .$$

length i	1	2	3	4
price p_i	1	5	8	9

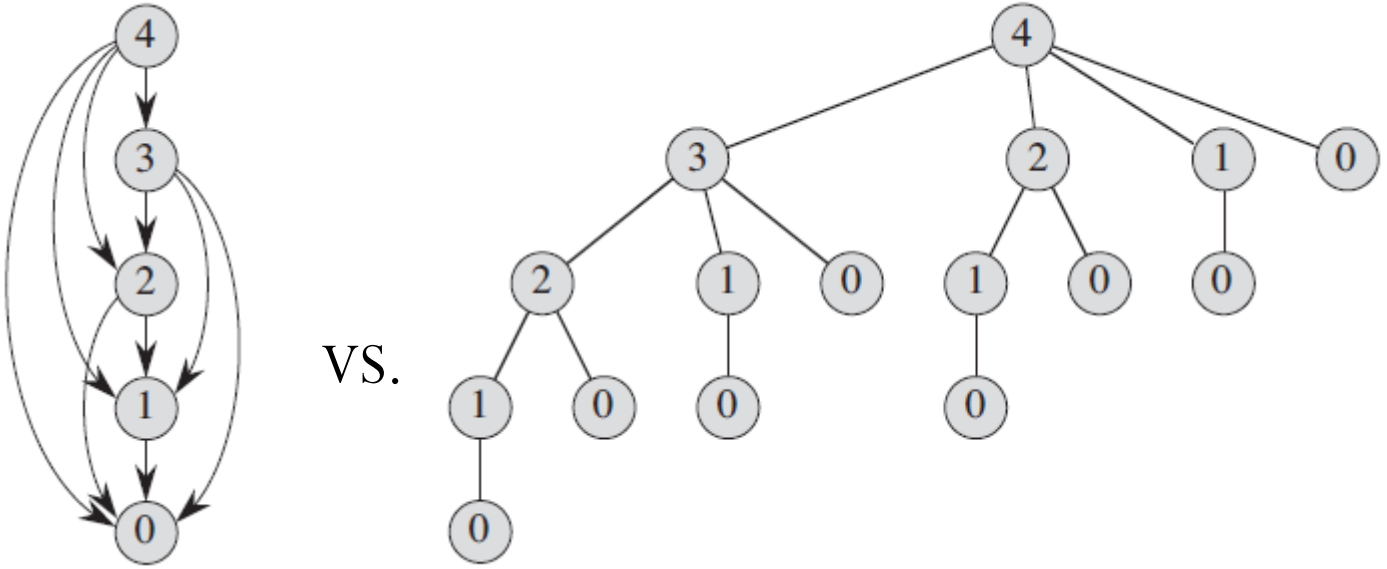
Recursion tree



The recursion tree showing recursive calls resulting from a call CUT-ROD(p, n) for n=4. Each node label gives the size n of the corresponding subproblem, so that an edge from a parent with label s to a child with label t corresponds to cutting off an initial piece of size s - t and leaving a remaining subproblem of size t . A path from the root to a leaf corresponds to one of the 2^{n-1} ways of cutting up a rod of length n.

In general, this recursion tree has 2^n nodes and 2^{n-1} leaves.

Collapse the nodes



The running time of procedure BOTTOM-UP-CUT-ROD is $\Theta(n^2)$, due to its doubly-nested loop structure.

Extended bottom-up cut

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
 1  let r[0..n] and s[0..n] be new arrays
 2  r[0] = 0
 3  for j = 1 to n
 4      q = -∞
 5      for i = 1 to j
 6          if q < p[i] + r[j - i]
 7              q = p[i] + r[j - i]
 8              s[j] = i
 9      r[j] = q
10  return r and s
```

Longest Common Subsequence Problem

- The diff program in Unix: what does it mean to say that we find the places where two files differ (including insertions and deletions)? Or, what does it mean to keep the "common" parts?
- Let it mean the longest subsequence present in both:

$$
\begin{aligned}
X &= a\ b\ \ c\ \ b\ \ d\ a\ b \\
Y &= \ \ b\ d\ c\ a\ b\ a\ b\ a \\
&\ \ \ \ \ \ b\ \ c\ \ b\ \ \ \ a
\end{aligned}
$$

- Running through all subsequences would take exponential time. There is a faster solution, recognizing that we only want to find some longest subsequence.
- Let $c[i,j]$ be the length of the longest common subsequence of the prefix of $X[1..i]$ and $Y[1..j]$. Recursion:

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1,j-1]+1 & \text{if } i,j > 0 \text{ and } x_i = y_j, \\ \max(c[i,j-1], c[i-1,j]) & \text{otherwise.} \end{cases}$$

We are not computing a function $C(X,Y,i,j)$ by naive recursion, but collect the values $c[i,j]$ as they are computed, in array c: $C(X,Y,i,j,c)$ checks whether $c[i,j]$ is defined. If yes, it just returns $c[i,j]$; else it uses the above recursion, and assigns $c[i,j]$ before returning the value.

$$\begin{array}{lccccccc} X = & a & b & c & b & d & a & b \\ Y = & b & d & c & a & b & a & b & a \\ & & b & c & b & & a \end{array}$$

Non-recursive implementation

- Compute the table $c[i, j]$ "bottom-up".
- Also, store the value $b[i, j] = \{\leftarrow\}$, $\{\uparrow\}$ or $\{\nwarrow\}$ depending on whether the optimum is $c[i, j-1]$, $c[i-1, j]$ or $c[i-1, j-1]+1$. (We also use the value $\{\leftarrow, \uparrow\}$ in case $c[i, j] = c[i-1, j] = c[i, j-1]$, though this is not important.)
- Find the longest common subsequence walking backwards on the arrows.

	j	1	2	3	4	5	6
i		b	d	c	a	b	a
1	a	0	0	0	1	1	1
2	b	1	1	1	1	2	2
3	c	1	1	2	2	2	2
4	b	1	1	2	2	3	3
5	d	1	2	2	2	3	3
6	a	1	2	2	3	3	4
7	b	1	2	2	3	4	4

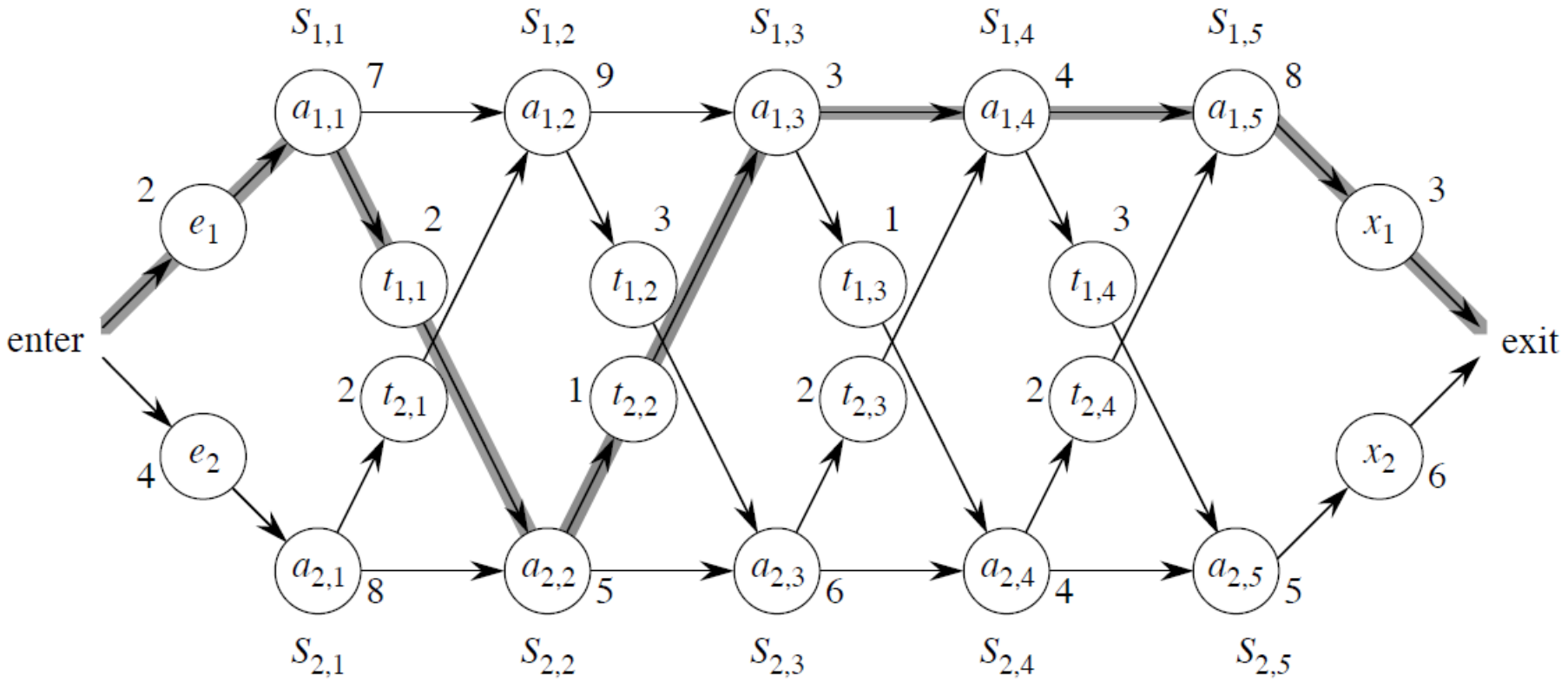
$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1,j-1]+1 & \text{if } i,j > 0 \text{ and } x_i = y_j, \\ \max(c[i,j-1], c[i-1,j]) & \text{otherwise.} \end{cases}$$

- Compute the table $c[i,j]$ "bottom-up".
- Also, store the value $b[i,j] = \{\leftarrow\}$, $\{\uparrow\}$ or $\{\nwarrow\}$ depending on whether the optimum is $c[i,j-1]$, $c[i-1,j]$ or $c[i-1,j-1]+1$. (We also use the value $\{\leftarrow,\uparrow\}$ in case $c[i,j] = c[i-1,j] = c[i,j-1]$, though this is not important.)
- Find the longest common subsequence walking backwards on the arrows.

LCS-length(X,Y)

```
1  m ← X.length; n ← Y.length
2  b[1..m, 1..n], c[0..m, 0..n] ← new tables
3  for i = 1 to m do c[i, 0] ← 0
4  for j = 1 to n do c[0, j] ← 0
5  for i = 1 to m do
6      for j = 1 to n do
7          if x_i = y_j then
8              c[i, j] ← c[i − 1, j − 1] + 1
9              b[i, j] ← {↖}
10         else
11             c[i, j] ← max(c[i, j − 1], c[i − 1, j]); b[i, j] ← ∅
12             if c[i − 1, j] = c[i, j] then b[i, j] ← {↑}
13             if c[i, j − 1] = c[i, j] then b[i, j] ← b[i, j] ∪ {←}
14 return c, b
```

Assembly-line scheduling



Assembly-line scheduling

Automobile factory with two assembly lines.

- Each line has n stations: $S_{1,1}, \ldots, S_{1,n}$ and $S_{2,1}, \ldots, S_{2,n}$.
- Corresponding stations $S_{1,j}$ and $S_{2,j}$ perform the same function but can take different amounts of time $a_{1,j}$ and $a_{2,j}$.
- Entry times e_1 and e_2.
- Exit times x_1 and x_2.
- After going through a station, can either
  - stay on same line; no cost, or
  - transfer to other line; cost after $S_{i,j}$ is $t_{i,j}$. ($j = 1, \ldots, n-1$. No $t_{i,n}$, because the assembly line is done after $S_{i,n}$.)

Problem: Given all these costs (time = cost), what stations should be chosen from line 1 and from line 2 for fastest way through factory?

Try all possibilities?

- Each candidate is fully specified by which stations from line 1 are included. Looking for a subset of line 1 stations.
- Line 1 has n stations.
- 2^n subsets.
- Infeasible when n is large.

Structure of an optimal solution

Think about fastest way from entry through $S_{1,j}$.

- If $j = 1$, easy: just determine how long it takes to get through $S_{1,1}$.
- If $j \geq 2$, have two choices of how to get to $S_{1,j}$:
  - Through $S_{1,j-1}$, then directly to $S_{1,j}$.
  - Through $S_{2,j-1}$, then transfer over to $S_{1,j}$.

Suppose fastest way is through $S_{1,j-1}$.

Key observation

What is the fastest way from entry through $S_{1,j-1}$ in this solution?

If there were a faster way through $S_{1,j-1}$, we would use it instead to come up with a faster way through $S_{1,j}$.

Now suppose a fastest way is through $S_{2,j-1}$. Again, we must have taken a fastest way through $S_{2,j-1}$. Otherwise use some faster way through $S_{2,j-1}$ to give a faster way through $S_{1,j}$

Optimal structure

Generally: An optimal solution to a problem (fastest way through $S_{1,j}$) contains within it an optimal solution to subproblems (fastest way through $S_{1,j-1}$ or $S_{2,j-1}$).

This is optimal substructure.

Use optimal substructure to construct optimal solution to problem from optimal solutions to subproblems.

Fastest way through $S_{1,j-1}$ is either

- fastest way through $S_{1,j-1}$ then directly through $S_{1,j}$, or
- fastest way through $S_{2,j-1}$, transfer from line 2 to line 1, then through $S_{1,j}$.

Optimal structure

Symmetrically:

Fastest way through $S_{2,j}$ is either

- fastest way through $S_{2,j-1}$ then directly through $S_{2,j}$, or
- fastest way through $S_{1,j-1}$, transfer from line 1 to line 2, then through $S_{2,j}$.

Therefore, to solve problems of finding a fastest way through S1, j and S2, j , solve subproblems of finding a fastest way through $S_{1,j-1}$ and $S_{2,j-1}$.

Recursive solution

Let $f_i[j]$ = fastest time to get through $S_{i,j}$, $i = 1, 2$ and $j = 1, \ldots, n$.

Goal: fastest time to get all the way through = f^*.

$$
\begin{aligned}
f^* &= \min(f_1[n] + x_1,\ f_2[n] + x_2) \\
f_1[1] &= e_1 + a_{1,1} \\
f_2[1] &= e_2 + a_{2,1}
\end{aligned}
$$

For $j = 2, \ldots, n$:

$$
\begin{aligned}
f_1[j] &= \min(f_1[j-1] + a_{1,j},\ f_2[j-1] + t_{2,j-1} + a_{1,j}) \\
f_2[j] &= \min(f_2[j-1] + a_{2,j},\ f_1[j-1] + t_{1,j-1} + a_{2,j})
\end{aligned}
$$

$f_i[j]$ gives the *value* of an optimal solution. What if we want to *construct* an optimal solution?

Recursive solution

- $l_i[j]$ = line # (1 or 2) whose station $j-1$ is used in fastest way through $S_{i,j}$.
- In other words $S_{l_i[j],j-1}$ precedes $S_{i,j}$.
- Defined for $i = 1, 2$ and $j = 2, \ldots, n$.
- l^* = line # whose station n is used.

For example:

j	1	2	3	4	5
$f_1[j]$	9	18	20	24	32
$f_2[j]$	12	16	22	25	30

$f^* = 35$

j	2	3	4	5
$l_1[j]$	1	2	1	1
$l_2[j]$	1	2	1	2

$l^* = 1$

Go through optimal way given by l values

Compute an optimal solution

Could just write a recursive algorithm based on above recurrences.

- Let $r_i(j) =$ # of references made to $f_i[j]$.
- $r_1(n) = r_2(n) = 1$.
- $r_1(j) = r_2(j) = r_1(j+1) + r_2(j+1)$ for $j = 1, \ldots, n-1$.

Claim

$r_i(j) = 2^{n-j}$.

Proof Induction on j, down from n.

Basis: $j = n$. $2^{n-j} = 2^0 = 1 = r_i(n)$.

Inductive step: Assume $r_i(j+1) = 2^{n-(j+1)}$.

$$\begin{aligned} \text{Then } r_i(j) &= r_i(j+1) + r_2(j+1) \\ &= 2^{n-(j+1)} + 2^{n-(j+1)} \\ &= 2^{n-(j+1)+1} \\ &= 2^{n-j} \ . \end{aligned}$$

■ (claim)

Therefore, $f_1[1]$ alone is referenced 2^{n-1} times!

So top down isn't a good way to compute $f_i[j]$.

Observation

$f_i[\,j\,]$ depends only on $f_1[\,j-1\,]$ and $f_2[\,j-1\,]$

for $j \geq 2$.

So compute in order of increasing j .

0-1 Knapsack problem

Recursive Formula

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w], B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

- The best subset of S_k that has the total weight w, either contains item k or not.
- First case: $w_k > w$. Item k can't be part of the solution, since if it was, the total weight would be $> w$, which is unacceptable.
- Second case: $w_k \leq w$. Then the item k can be in the solution, and we choose the case with greater value.

0-1 Knapsack Algorithm

```
for w = 0 to W
    B[0,w] = 0
for i = 1 to n
    B[i,0] = 0
for i = 1 to n
    for w = 0 to W
        if w_i <= w // item i can be part of the solution
            if b_i + B[i-1,w-w_i] > B[i-1,w]
                B[i,w] = b_i + B[i-1,w- w_i]
            else
                B[i,w] = B[i-1,w]
        else B[i,w] = B[i-1,w]  // w_i > w
```

0-1 Knapsack problem

Running time

```
for w = 0 to W                    O(W)
   B[0,w] = 0
for i = 1 to n
   B[i,0] = 0
for i = 1 to n                    Repeat n times
   for w = 0 to W                 O(W)
      < the rest of the code >
```

What is the running time of this algorithm?

O(n*W)

Remember that the brute-force algorithm takes $O(2^n)$

Example

Let's run our algorithm on the following data:

n = 4 (# of elements)
W = 5 (max weight)
Elements (weight, benefit):
(2,3), (3,4), (4,5), (5,6)

Example

Example (2)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1						
2						
3						
4						

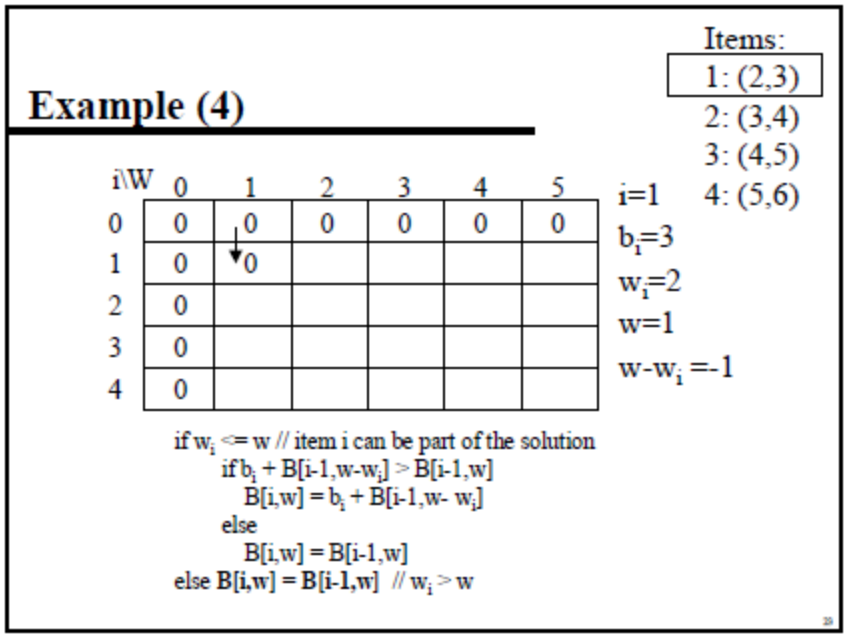
for w = 0 to W
    B[0,w] = 0

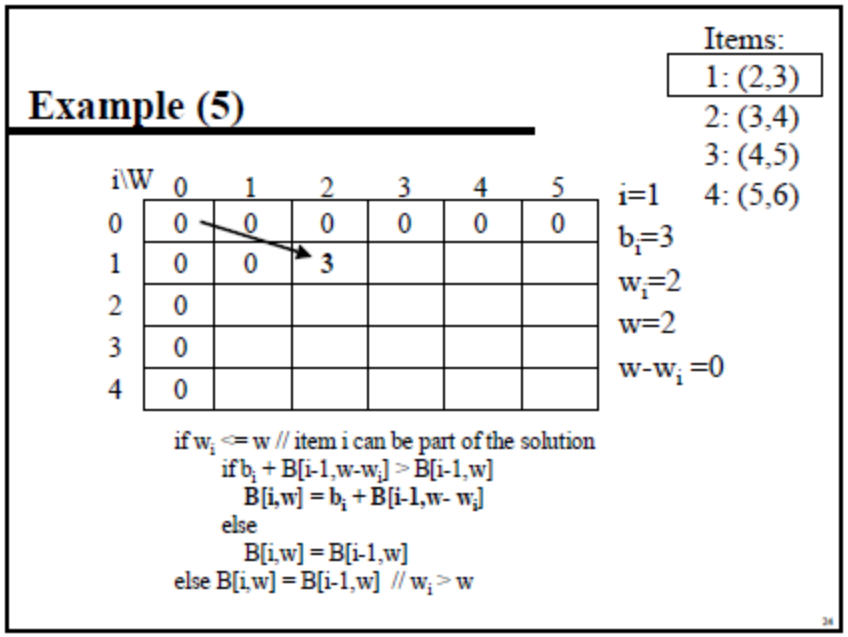
Example (3)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

for i = 1 to n
    B[i,0] = 0

Example


Example (4)
Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)
i\W 0 1 2 3 4 5
0 0 0 0 0 0 0
1 0 0
2 0
3 0
4 0
i=1
b_i=3
w_i=2
w=1
w-w_i =-1
if w_i <= w // item i can be part of the solution
if b_i + B[i-1,w-w_i] > B[i-1,w]
B[i,w] = b_i + B[i-1,w- w_i]
else
B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // w_i > w


Example (5)
Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)
i\W 0 1 2 3 4 5
0 0 0 0 0 0 0
1 0 0 3
2 0
3 0
4 0
i=1
b_i=3
w_i=2
w=2
w-w_i =0
if w_i <= w // item i can be part of the solution
if b_i + B[i-1,w-w_i] > B[i-1,w]
B[i,w] = b_i + B[i-1,w- w_i]
else
B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // w_i > w

Example

Example (14)

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4		
4	0					

$i=3$
$b_i=5$
$w_i=4$
$w= 1..3$

if w_i <= w // item i can be part of the solution
  if b_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = b_i + B[i-1,w- w_i]
  else
    B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // w_i > w

Example (15)

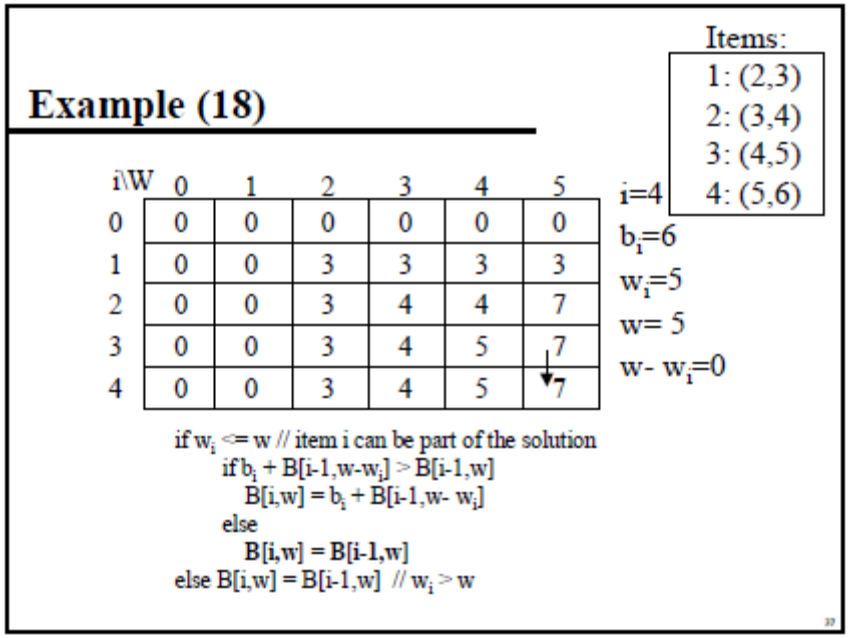
Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	
4	0					

$i=3$
$b_i=5$
$w_i=4$
$w= 4$
$w- w_i=0$

if w_i <= w // item i can be part of the solution
  if b_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = b_i + B[i-1,w- w_i]
  else
    B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // w_i > w

Example


Example (18)
Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)
i\W 0 1 2 3 4 5
0 0 0 0 0 0 0
1 0 0 3 3 3 3
2 0 0 3 4 4 7
3 0 0 3 4 5 7
4 0 0 3 4 5 7
i=4
b_i=6
w_i=5
w= 5
w- w_i=0
if w_i <= w // item i can be part of the solution
if b_i + B[i-1,w-w_i] > B[i-1,w]
B[i,w] = b_i + B[i-1,w- w_i]
else
B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // w_i > w

Greedy algorithms – concepts

Greedy algorithms

GA are characterized by the following two properties:

- the algorithm works in stages, and during each stage a selection is made which is locally optimal;
- the sum totality of all the locally optimal choices produces a globally optimal solution.

A greedy method does not always lead to a globally optimal solution.

They are referred to as will refer to it as a heuristic, or a greedy heuristic.

These are algorithms that provide a "short cut" to a solution, but not necessarily to an optimal solution.

The term "greedy algorithm" is used for a greedy method that always produces a correct/optimal solution.

Problems

Some problems that can be solved using a greedy algorithm:

- Minimum Spanning Tree: finding a spanning tree for a graph whose edges have minimal weight.
- Fractional Knapsack: selecting what goods to load into a shipping container.
- Activity Selection: finding a maximum set of activities (each with start and end times) which can be hosted in an auditorium.
- Huffman Coding: finding a code for a set of words which minimizes the expected code-length.
- Optimal List Storage.

Greedy algorithms – applications

Activity selection

Given a set of *activities A* of length *n*

$A = \langle a_1, a_2, ..., a_n \rangle$

with *starting times*

$S = \langle s_1, s_2, ..., s_n \rangle$

and *finishing times*

$F = \langle f_1, f_2, ..., f_n \rangle$

such that $0 \leq s_i < f_i < \infty$, we define two activities a_i and a_j to be *compatible* if $f_i \leq s_j$ **or** $f_j \leq s_i$

i.e. one activity ends before the other begins so they do not overlap.

Task

Find a *maximal* set of compatible activities, e.g. scheduling the most activities in a lecture hall.

Note that we want to find the maximum *number* of activities, **not** necessarily the maximum *use* of the resource.

Greedy Algorithm Solution

To use the greedy approach, we must *prove* that the greedy choice produces an optimal solution (although not necessarily the *only* solution).

Consider any non-empty subproblem S_{ij} with activity a_m having the earliest finishing time, i.e.

$$f_{min} = \min\{f_k : a_k \in S_{ij}\}$$

then the following two conditions must hold

1. a_m is used in an optimal subset of S_{ij}
2. $S_{im} = \emptyset$ leaving S_{mj} as the only subproblem

meaning that the greedy solution produces an optimal solution.

Proof

Let A_{ij} be an optimal solution for S_{ij} and a_k be the first activity in A_{ij}

$\rightarrow$ If $a_k = a_m$ then the condition holds.

$\rightarrow$ If $a_k \neq a_m$ then construct $A_{ij}' = A_{ij} - \{a_k\} \cup \{a_m\}$.

Since $f_m \leq f_k \Rightarrow A_{ij}'$ is still optimal.

If S_{im} is non-empty $\Rightarrow a_k$ with

$f_i \leq s_k < f_k \leq s_m < f_m \Rightarrow f_k < f_m$ which contradicts the assumption that f_m is the minimum finishing time. Thus $S_{im} = \emptyset$.

Thus instead of having 2 subproblems each with n-j-1 choices per problem, we have reduced it to 1 subproblem with 1 choice.

Algorithm

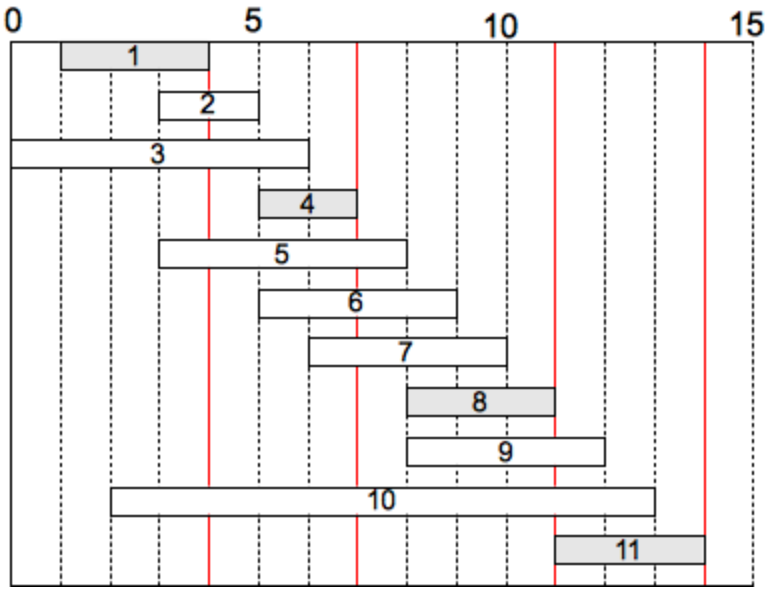
Always start by choosing the first activity (since it finishes first), then repeatedly choose the next compatible activity until none remain.

The algorithm can be implemented either recursively or iteratively in O(n) time (assuming the activities are sorted by finishing times) since each activity is examined only once.

Example

Consider the following set of activities represented graphically in non-decreasing order of finishing times

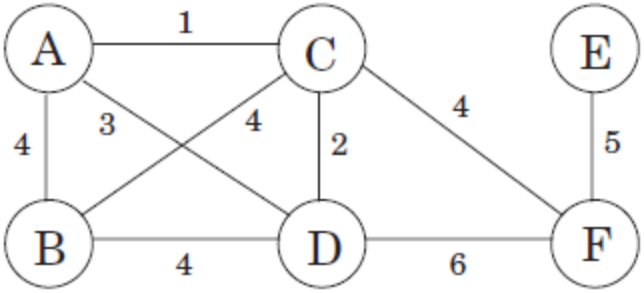
i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16



Minimum spanning trees

- Electronic circuit designs often need to make the pins of several components electrically equivalent by wiring them together.
- To interconnect a set of n pins, we can use an arrangement of n -1 wires, each connecting two pins.
- Of all such arrangements, the one that uses the least amount of wire is usually the most desirable.
- We can model this wiring problem with a connected, undirected graph $G = (V, E)$, where V is the set of pins, E is the set of possible interconnections between pairs of pins.
- For each edge $(v, u) \subseteq E$, we have a weight $w(u, v)$ specifying the cost (amount of wire needed) to connect u and v.

A graph



One immediate observation is that the optimal set of edges cannot contain a cycle, because removing an edge from this cycle would reduce the cost without compromising connectivity:

Removing a cycle edge cannot disconnect a graph.

So the solution must be connected and acyclic: undirected graphs of this kind are called *trees*.

The particular tree we want is the one with minimum total weight, known as the *minimum spanning tree*.

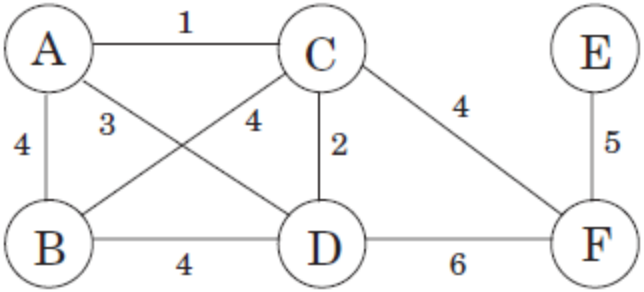
Formal definition

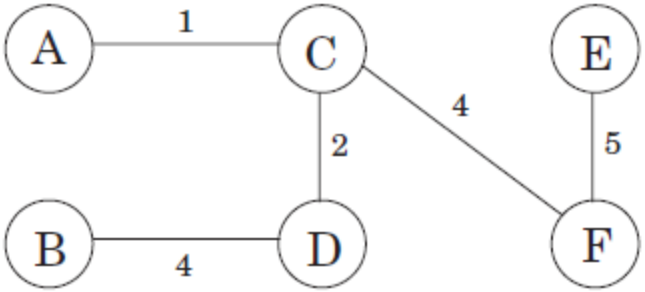
Input: An undirected graph $G = (V, E)$; edge weights w_e.

Output: A tree $T = (V, E)$, with $E' \subseteq E$, that minimizes

$$weight(T) = \sum_{e \in E'} w_e$$

The minimum spanning tree has a cost of 16:





This is not the only optimal solution. Can you spot another?

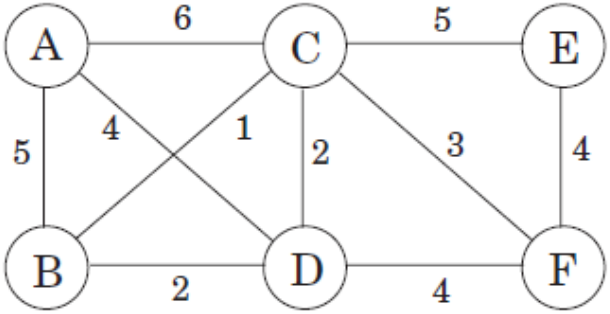
A greedy approach

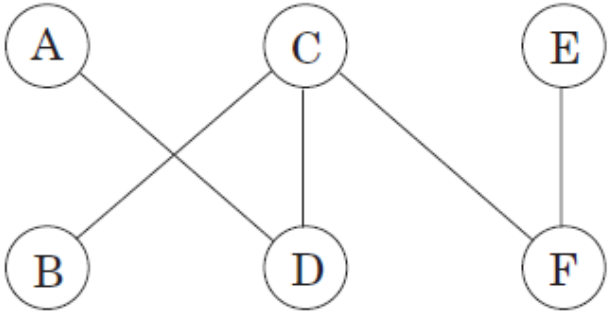
Kruskal's minimum spanning tree algorithm starts with the empty graph and then selects edges from E according to the following rule.

Repeatedly add the next lightest edge that doesn't produce a cycle.

It constructs the tree edge by edge and, apart from taking care to avoid cycles, simply picks whichever edge is cheapest at the moment.

This is a *greedy* algorithm: every decision it makes is the one with the most obvious immediate advantage.



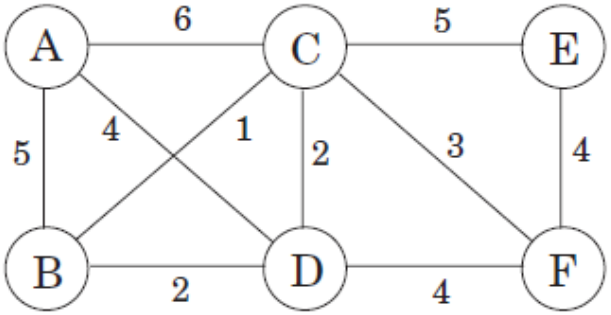


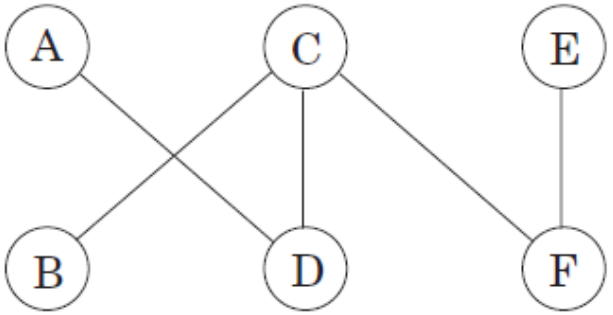
Example

- Start with an empty graph and then attempt to add edges in increasing order of weight (ties are broken arbitrarily):

$B-C,\ C-D,\ B-D,\ C-F,\ D-F,\ E-F,\ A-D,\ A-B,\ C-E,\ A-C.$

- The first two succeed, but the third, $B - D$, would produce a cycle if added.
- So we ignore it and move along. The final result is a tree with cost 14, the minimum possible.





Kruskal algorithm

- Model the algorithm's state as a collection of disjoint sets, each of which contains the nodes of a particular component. Initially each node is in a component by itself:

  makeset(x): *create a singleton set containing just x.*

- Repeatedly test pairs of nodes to see if they belong to the same set.

  find(x): *to which set does x belong?*

- Whenever an edge is added, two components are merged:

  union(x, y): *merge the sets containing x and y.*

Kruskal's minimum spanning tree algorithm

```
procedure kruskal (G, w)
Input:  A connected undirected graph G = (V, E) with edge weights w_e
output: A minimum spanning three defined by the edges X

for all u ∈ V:
   makeset (u)

X = {}
sort the edges E by weight
for all edges {u, v} ∈ E, in increasing order of weight:
   if find(u) ≠ find(v):
      add edge {u, v} to X
      union(u, v)
```

A data structure for disjoint sets

Union by rank:

One way to store a set is as a directed tree.

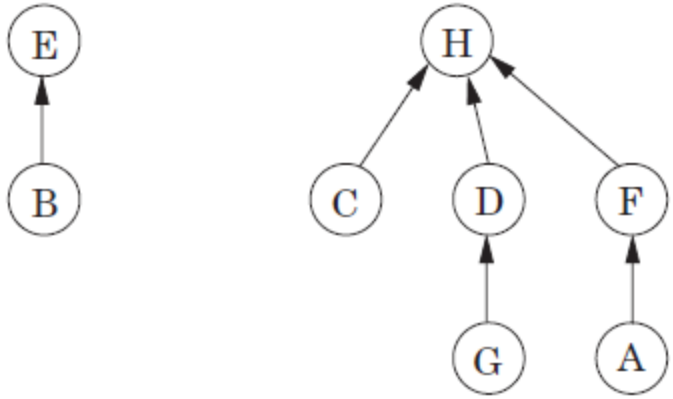
Nodes of the tree are elements of the set, arranged in no particular order, and each has parent pointers that eventually lead up to the root of the tree.

This root element is a convenient representative, or name, for the set.

It is distinguished from the other elements by the fact that its parent pointer is a self-loop.

A data structure for disjoint sets

A directed-tree representation of two sets $\{B, E\}$ *and* $\{A, C, D, F, G, H\}$:



Union by rank

In addition to a parent pointer π, each node also has a rank that, for the time being, should be interpreted as the height of the subtree hanging from that node.

```
procedure makeset(x)
π(x) = x
rank(x) = 0

function find(x)
while x ≠ π(x): x = π(x)
return x
```

Makeset

makeset is a constant-time operation.

find follows parent pointers to the root of the tree and therefore takes time proportional to the height of the tree.

The tree actually gets built via the third operation, *union*, and so this procedure should keep trees shallow.

Merging two sets

Make the root of one point to the root of the other.

If the representatives (roots) of the sets are r_x and r_y, should we make

r_x point to r_y or the other way around?

A good strategy is to make the root of the shorter tree point to the root of the taller tree.

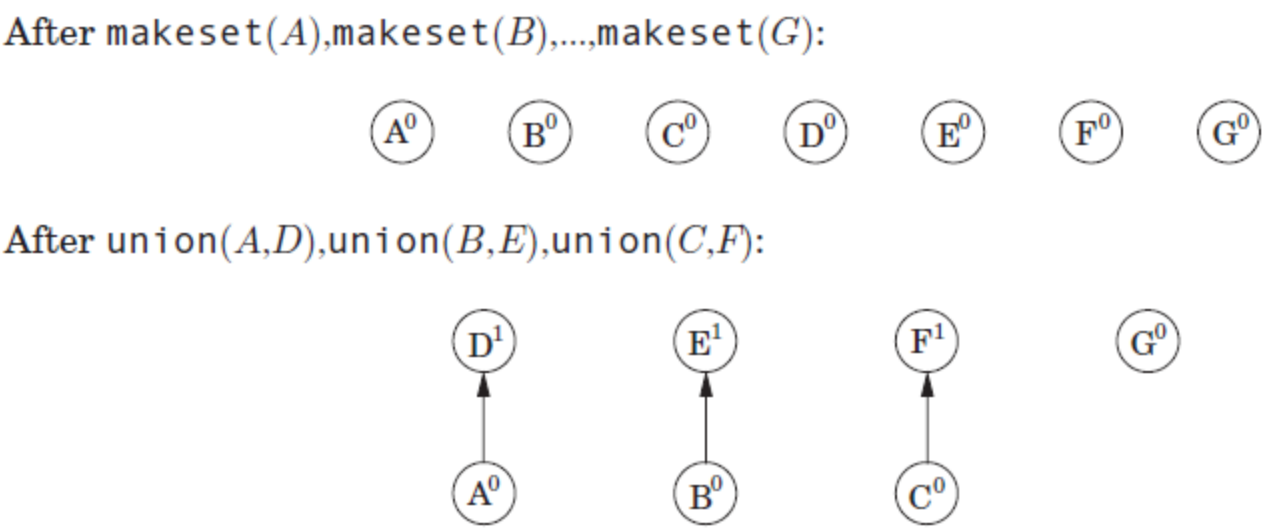
This way, the overall height increases only if the two trees being merged are equally tall. Instead of explicitly computing heights of trees, we will use the rank numbers of their root nodes—which is why this scheme is called *union by rank*.

Union by rank

```
procedure union(x, y)
r_x = find(x)
r_y = find(y)
if r_x = r_y: return
if rank(r_x) > rank(r_y):
    π(r_y) = r_x
else:
    π(r_x) = r_y
    if rank(r_x) = rank(r_y): rank(r_y) = rank(r_y) + 1
```

Example

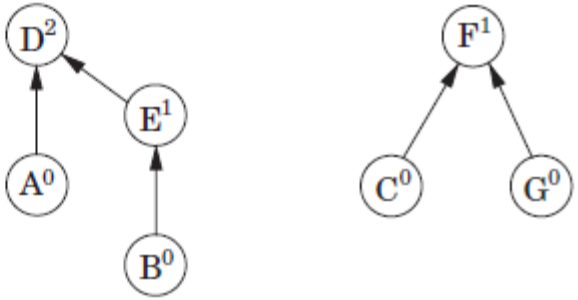
A sequence of disjoint-set operations. Superscripts denote rank.



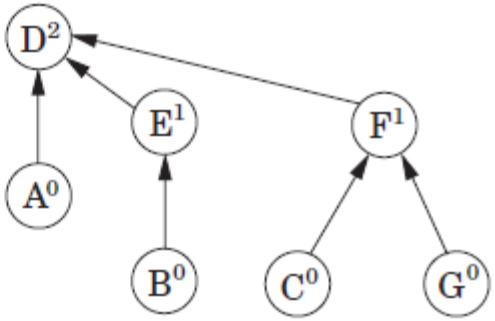
Example

A sequence of disjoint-set operations. Superscripts denote rank.

After union(C,G),union(E,A):



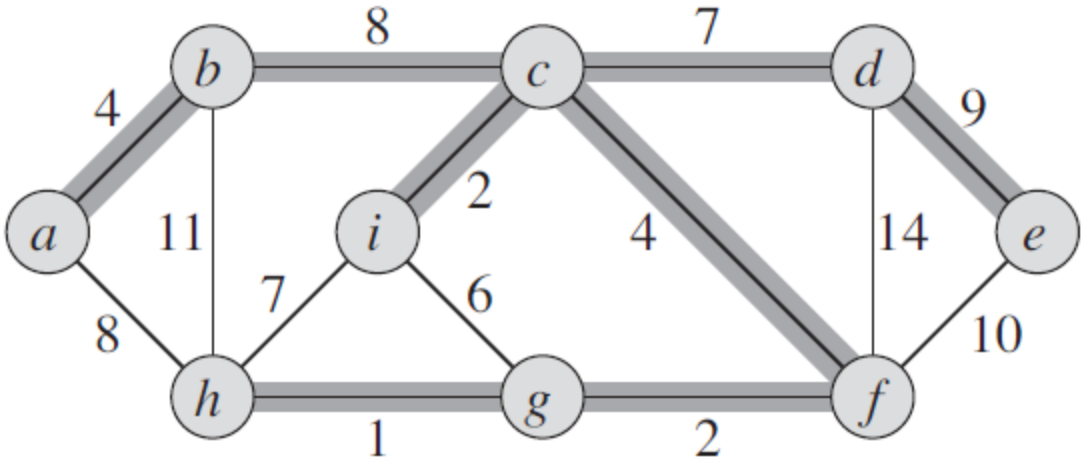
After union(B,G):



How the Kruskal works

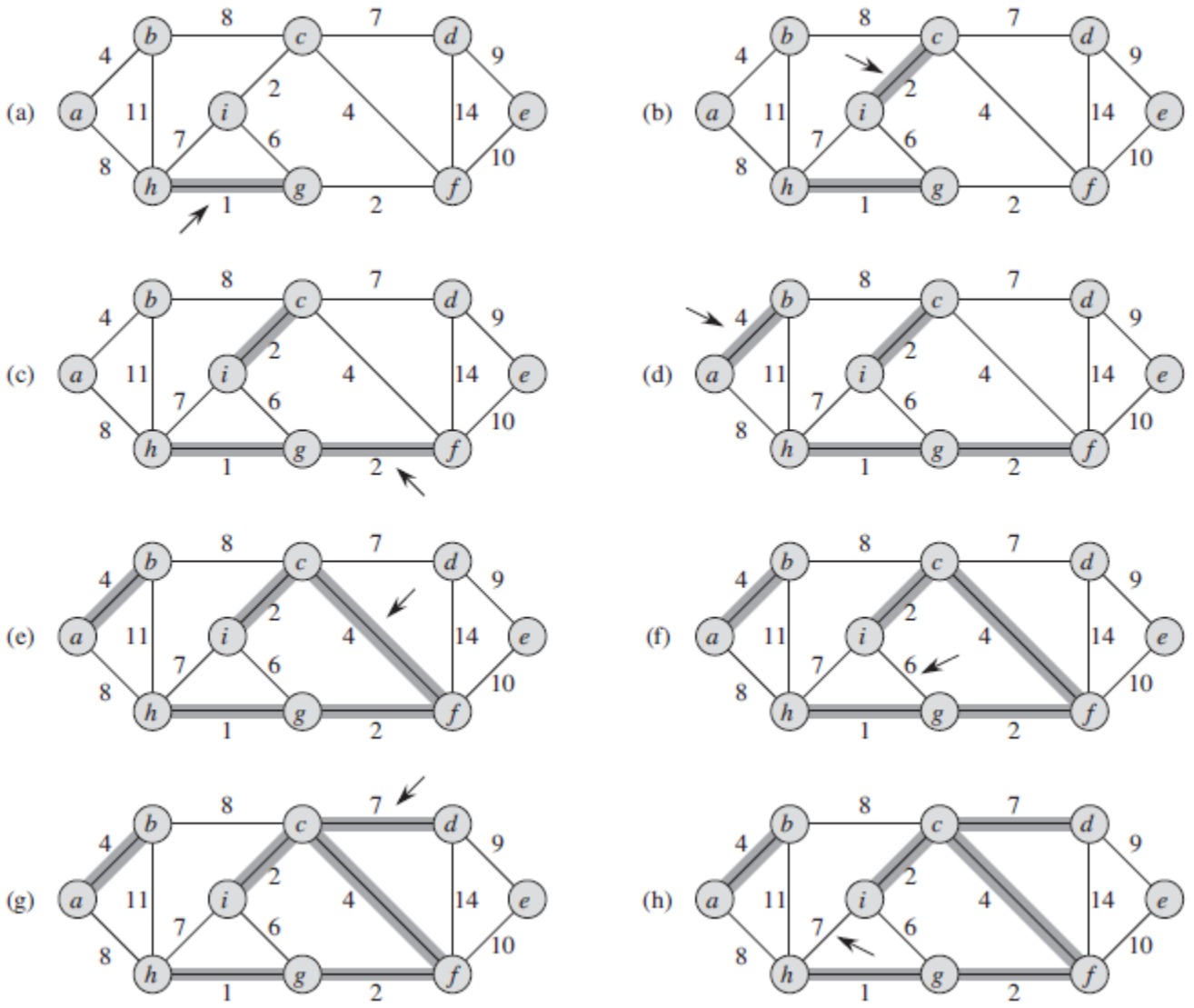
Let's consider minimum spanning tree for a connected graph.

The weights on edges are shown, and the edges in a minimum spanning tree are shaded. The total weight of the tree is 37. This minimum spanning tree is not unique: removing the edge (b, c) and replacing it with the edge (a, h) yields another spanning tree with weight 37.



How the Kruskal works

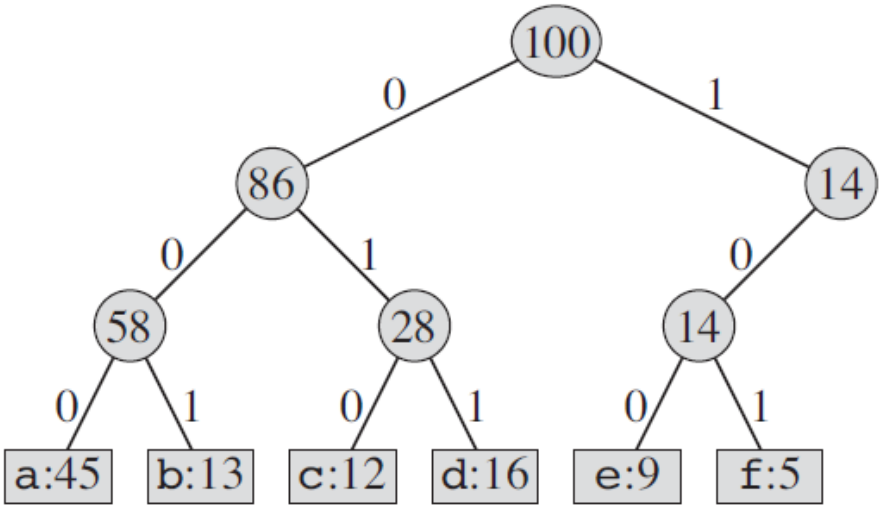
- The algorithm considers each edge in sorted order by weight.
- An arrow points to the edge under consideration at each step of the algorithm.
- If the edge joins two distinct trees in the forest, it is added to the forest, thereby merging the two trees.


(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
a
b
c
d
e
f
g
h
i
4
8
11
8
7
2
4
7
6
9
14
10
1
2

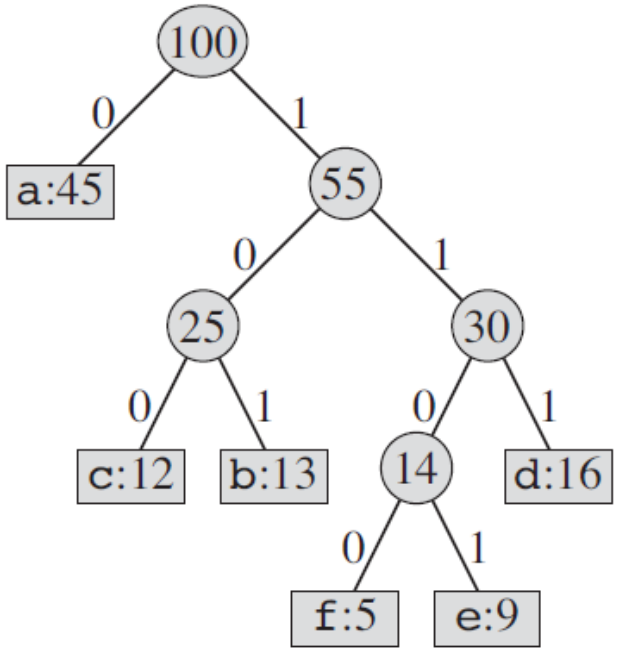
Huffman encoding:

(a) The tree corresponding to the fixed-length code **a** = 000, . . . , **f** = 101.

(b) The tree corresponding to the optimal prefix code **a** = 0, **b** = 101, . . . , **f** = 1100.



(a)

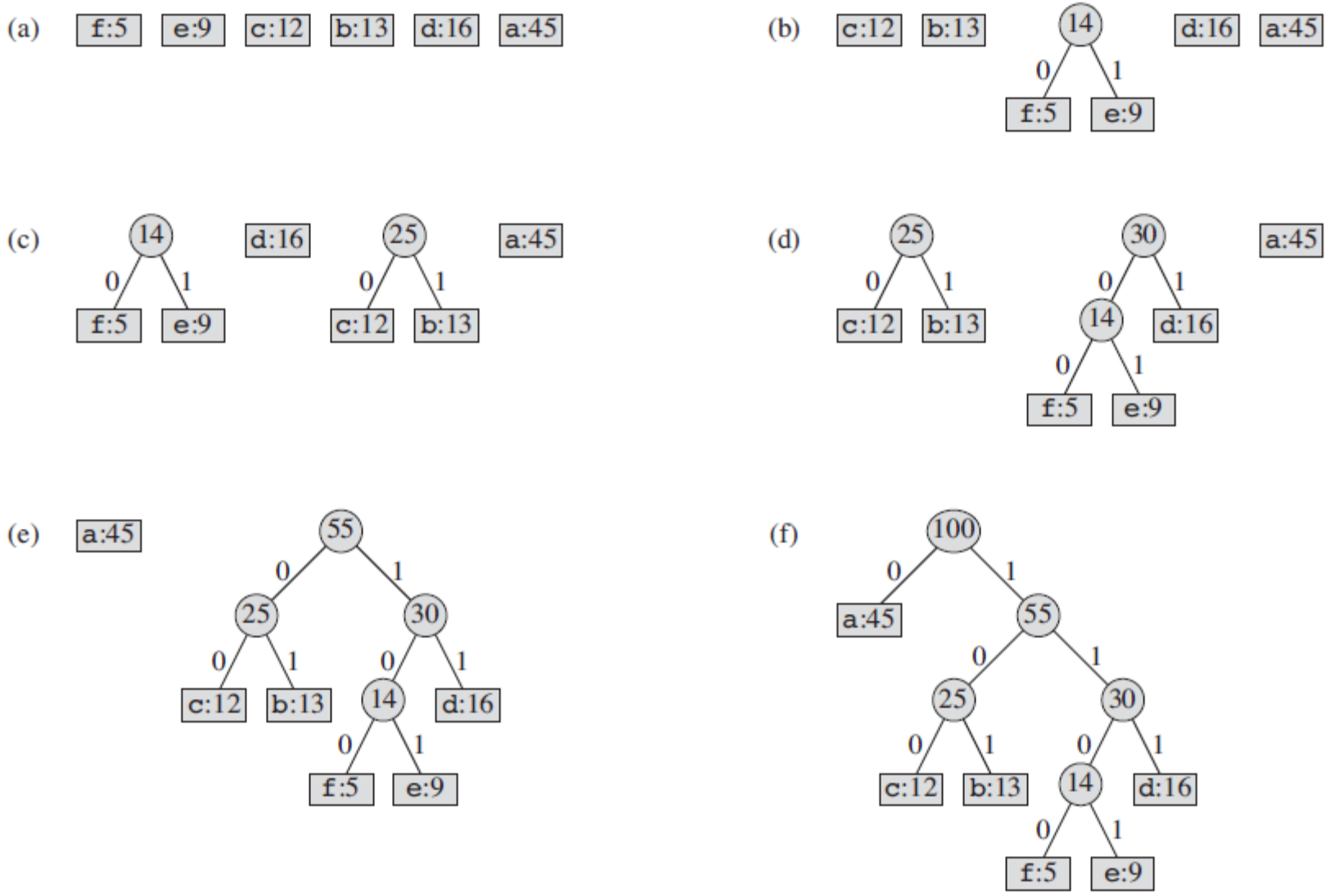


(b)

Constructing the Huffman code

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

The steps of Huffman code



Horn clauses

- In order to display human-level intelligence, a computer must be able to perform at least some modicum of logical reasoning.
- Horn formulas are a particular framework for doing this, for expressing logical facts and deriving conclusions.
- The most primitive object in a Horn formula is a Boolean variable, taking value either true or false.

Horn clauses

A literal is either a variable **x** or its negation **x** ("NOT x"). In Horn formulas, knowledge about variables is represented by two kinds of clauses:

1. Implications, whose left-hand side is an AND of any number of positive literals and whose right-hand side is a single positive literal. These express statements of the form "if the conditions on the left hold, then the one on the right must also be true." $(z \wedge w) \Rightarrow u$. A degenerate type of implication is the singleton "⇒ x," meaning simply that x is true.
2. Pure negative clauses, consisting of an OR of any number of negative literals, as in (u ∨ v ∨ y)

Horn clause

- Given a set of clauses of these two types, the goal is to determine whether there is a consistent explanation: an assignment of true/false values to the variables that satisfies all the clauses. This is also called a *satisfying assignment*.
- The implications amounts to setting some of the variables to true, while the negative clauses encourage to make them false.
- The strategy for solving a Horn formula is this:
- Start with all variables false.
- Then proceed to set some of them to true, one by one, but very reluctantly, and only if we absolutely have to because an implication would otherwise be violated.
- Once we are done with this phase and all implications are satisfied, only then do we turn to the negative clauses and make sure they are all satisfied.

Algorithm

```
Input:  a Horn formula
Output: a satisfying assignment, if one exists

set all variables to false

while there is an implication that is not satisfied:
  set the right-hand variable of the implication to true

if all pure negative clauses are satisfied: return the
assignment
else: return “formula is not satisfiable”
```