Some Problems that can be solved using graphs

1. Suppose you run a day care for an office building and there are seven children A, B, C, D, E, F. we need to assign a locker where each child's parent can put the child's food. The children come and leave so they are not all there at the same time. You have 1 hour time slots starting7:00 a.m. to 9:00 am.A star in the table means a child is present at that time. What is the minimum number of lockers necessary? Show how you would assign the lockers

	A	B	C	D	E	F
7:00	*	-	*	-	*	-
8:00	-	*	*	-	-	-
9:00	-	-	-	*	*	*

2.We want to schedule a training programs with following training modules for employees: A,B,C,D,E,F,G,H
Following pairs of training modules have no common employee:
A-B,B-C,A-D,A-E,B-E,E-F,A-F,B-F,D-F, A-G,A-H,B-H,C-H
How many minimum training slots are needed?

3. Map coloring:
Color the states so that no two adjacent states have same color. What is the minimum number of colors to be used?

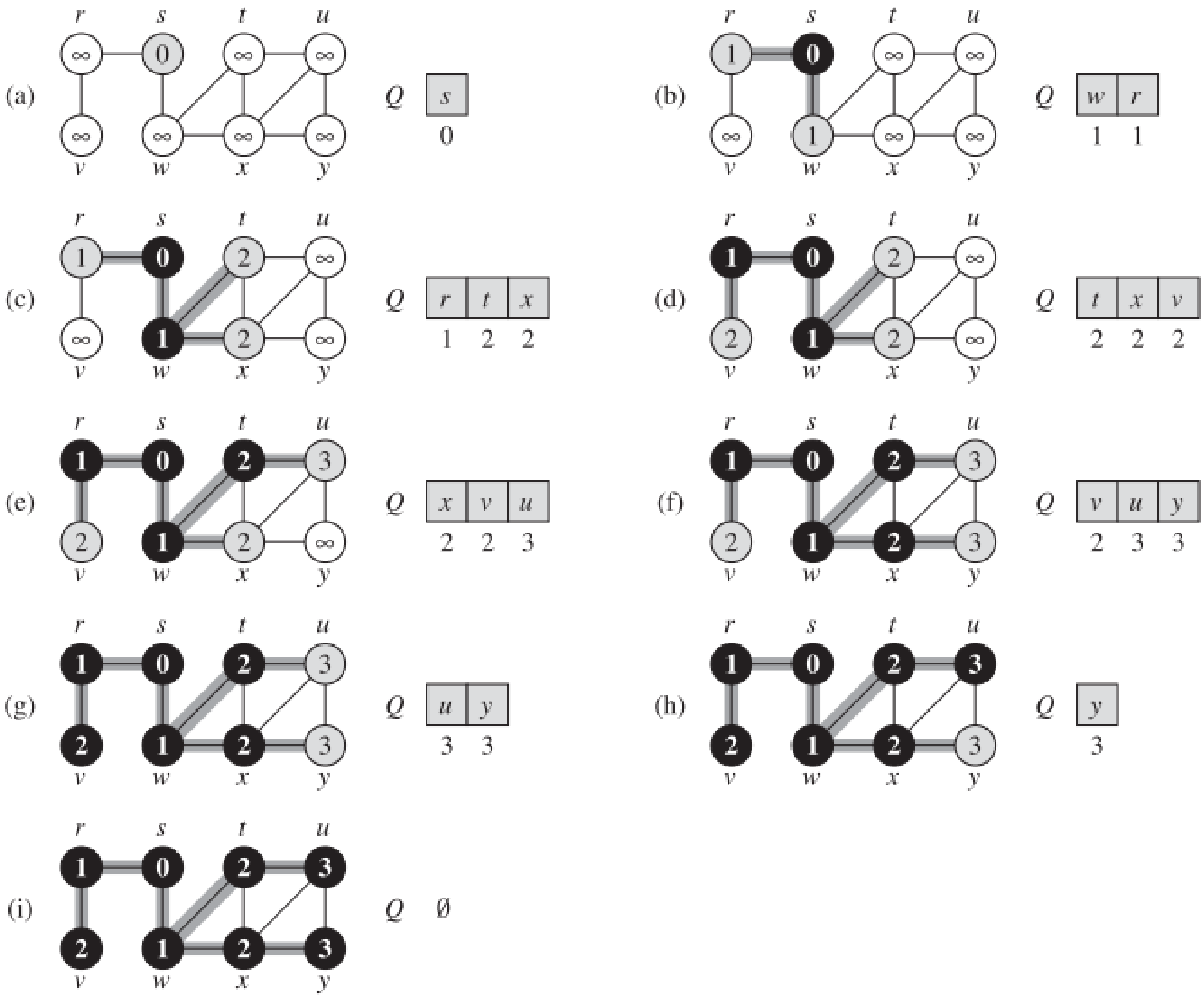
Graph Algorithms: Breadth First Search

$\text{BFS}(V, E, s)$

for each $u \in V - \{s\}$
  do $d[u] \leftarrow \infty$
$d[s] \leftarrow 0$
$Q \leftarrow \emptyset$
$\text{Enqueue}(Q, s)$
while $Q \neq \emptyset$
  do $u \leftarrow \text{Dequeue}(Q)$
    for each $v \in Adj[u]$
      do if $d[v] = \infty$
        then $d[v] \leftarrow d[u] + 1$
          $\text{Enqueue}(Q, v)$

Complexity = O(|V| +|E|)

BFS in action



Graph Algorithms: Depth First Search

DFS(V, E)

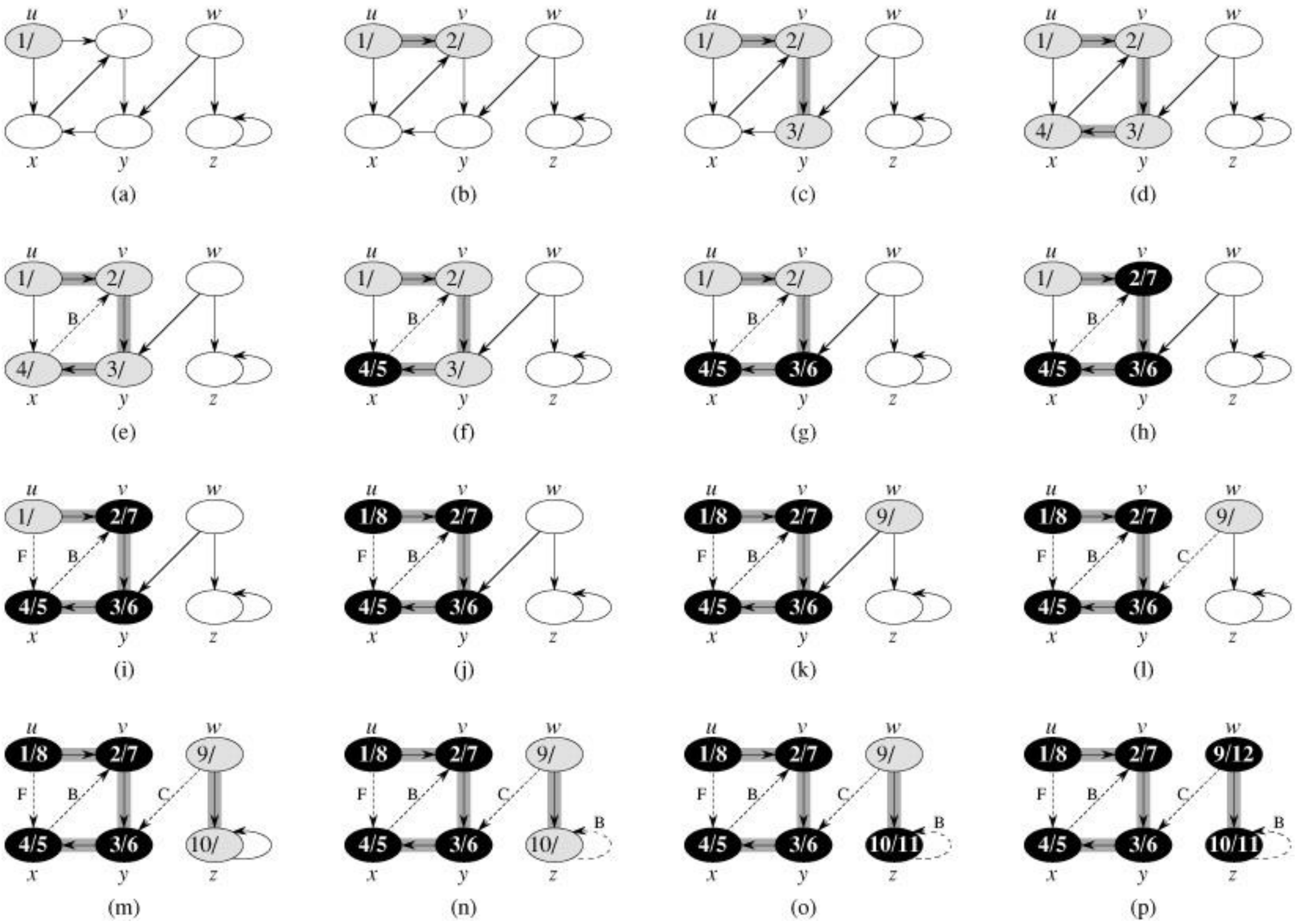
for each $u \in V$
    do $color[u] \leftarrow$ WHITE
$time \leftarrow 0$
for each $u \in V$
    do if $color[u] =$ WHITE
        then DFS-VISIT(u)

Complexity
= (|V| +|E|)

DFS-VISIT(u)

$color[u] \leftarrow$ GRAY $\triangleright$ discover u
$time \leftarrow time + 1$
$d[u] \leftarrow time$
for each $v \in Adj[u]$ $\triangleright$ explore (u, v)
    do if $color[v] =$ WHITE
        then DFS-VISIT(v)
$color[u] \leftarrow$ BLACK
$time \leftarrow time + 1$
$f[u] \leftarrow time$ $\triangleright$ finish u

DFS in action



Graph Algorithms: Kruskal's MST

$G = (V, E)$ is a connected, undirected, weighted graph. $w : E \rightarrow \mathbf{R}$.

- Starts with each vertex being its own component.
- Repeatedly merges two components into one by choosing the light edge that connects them (i.e., the light edge crossing the cut between them).
- Scans the set of edges in monotonically increasing order by weight.
- Uses a disjoint-set data structure to determine whether an edge connects vertices in different components.

Graph Algorithms: Kruskal's MST (pseudoC)

```
MST-KRUSKAL(G, w)
A = Ø
for each vertex v ∈ G.V
    MAKE-SET(v)
sort the edges of G.E into nondecreasing order by weight w
for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
    if FIND-SET(u) ≠ FIND-SET(v)
        A = A ∪ {(u, v)}
        UNION(u, v)
return A
```

```
MAKE-SET(x)
x.p = x
x.rank = 0
```

```
UNION(x, y)
LINK(FIND-SET(x), FIND-SET(y))
```

```
LINK(x, y)
if x.rank > y.rank
    y.p = x
else x.p = y
    if x.rank == y.rank
        y.rank = y.rank + 1
```

```
FIND-SET(x)
if x ≠ x.p
    x.p = FIND-SET(x.p)
return x.p
```

Time complexity = O($m$log$n$)

|v| = n, |E|=m

See Chapter 21 in the book for Union and Find-Set operations

Graph Algorithms: Prim's MST

- Builds one tree, so A is always a tree.
- Starts from an arbitrary "root" r.
- At each step, find a light edge crossing cut $(V_A, V - V_A)$, where V_A = vertices that A is incident on. Add this edge to A.

Graph Algorithms: Prim's MST (pseudoC)

PRIM(G, w, r)

$Q = \emptyset$
for each $u \in G.V$
  $u.key = \infty$
  $u.\pi =$ NIL
  INSERT(Q, u)
DECREASE-KEY($Q, r, 0$) // $r.key = 0$
while $Q \neq \emptyset$
  $u =$ EXTRACT-MIN(Q)
  for each $v \in G.Adj[u]$
    if $v \in Q$ **and** $w(u, v) < v.key$
      $v.\pi = u$
      DECREASE-KEY($Q, v, w(u, v)$)

Time Complexity (using binary Min-Heap):

$O(|E|\log|V| + |V|\log|V|)$

Single Source Shortest Path

INIT-SINGLE-SOURCE(V, s)

for each $v \in V$
  do $d[v] \leftarrow \infty$
    $\pi[v] \leftarrow$ NIL
$d[s] \leftarrow 0$

RELAX(u, v, w)

if $d[v] > d[u] + w(u, v)$
  then $d[v] \leftarrow d[u] + w(u, v)$
    $\pi[v] \leftarrow u$

Single Source Shortest Path: Bellman-Ford

$\text{BELLMAN-FORD}(V, E, w, s)$

$\text{INIT-SINGLE-SOURCE}(V, s)$

for $i \leftarrow 1$ to $|V| - 1$

    do for each edge $(u, v) \in E$

        do $\text{RELAX}(u, v, w)$

for each edge $(u, v) \in E$

    do if $d[v] > d[u] + w(u, v)$

        then return FALSE

return TRUE

Core: The first **for** loop relaxes all edges $|V| - 1$ times

Time: $\Theta(VE)$.

Single Source Shortest Path: Dijkstra

Not applicable to graph with –ve weights

DIJKSTRA(V, E, w, s)

INIT-SINGLE-SOURCE(V, s)

$S \leftarrow \emptyset$

$Q \leftarrow V$ $\qquad \triangleright$ i.e., insert all vertices into Q

while $Q \neq \emptyset$

  do $u \leftarrow$ EXTRACT-MIN(Q)

    $S \leftarrow S \cup \{u\}$

    for each vertex $v \in Adj[u]$

      do RELAX(u, v, w)

Dynamic Programming: Rod cutting

You are given a rod of length $n \geq 0$ (n in inches)

A rod of length i inches will be sold for p_i dollars

Cutting is free (simplifying assumption)

Problem: given a table of prices p_i determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces.

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

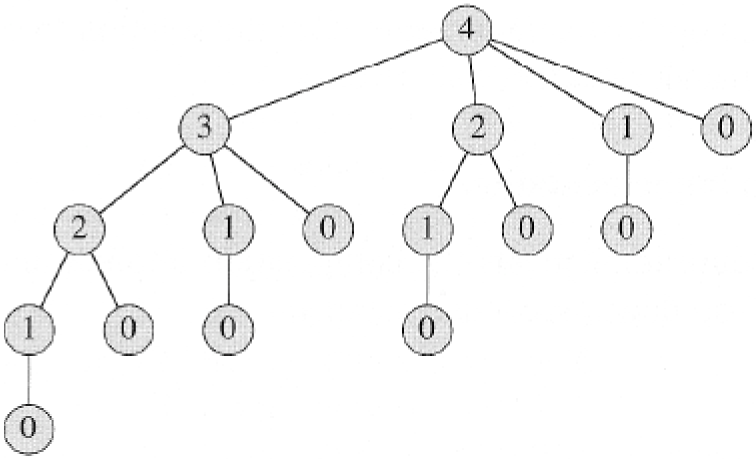
Dynamic Programming: Rod cutting

In ALL cases we have the recursion

$$r_n = \max(p_n, r_1 + r_{n\text{-}1}, r_2 + r_{n\text{-}2}, \ldots, r_{n\text{-}1} + r_1)$$

- **Without Dynamic Programming**

```
CUT-ROD(p, n)
  if n == 0
      return 0
  q = −∞
  for i = 1 to n
      q = max(q, p[i] + CUT-ROD(p, n − i))
  return q
```



The number of nodes for a tree corresponding to a rod of size n is:

$$T(0) = 1, \quad T(n) = 1 + \sum_{j=0}^{n-1} T(j) = 2^n, \; n \geq 1.$$

Dynamic Programming; Rod Cutting

- **With Dynamic Programming**

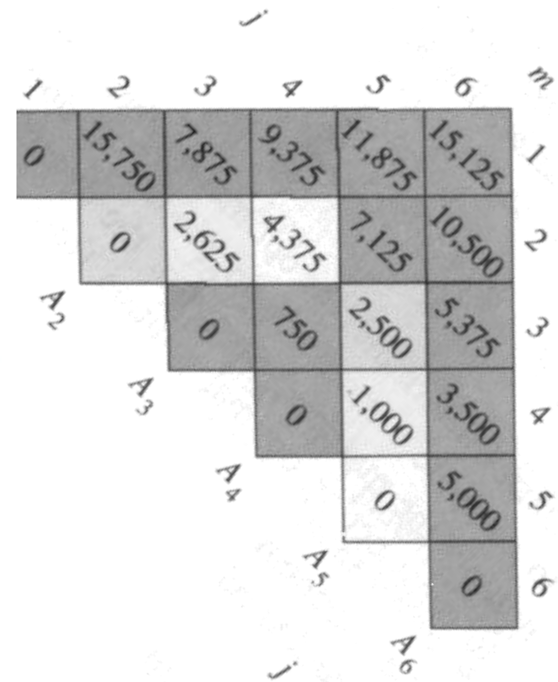
```
MEMOIZED-CUT-ROD(p, n)
let r[0..n] be a new array
for i = 0 to n
    r[i] = -∞
return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
if r[n] ≥ 0
    return r[n]
if n == 0
    q = 0
else q = -∞
    for i = 1 to n
        q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n - i, r))
r[n] = q
return q
```

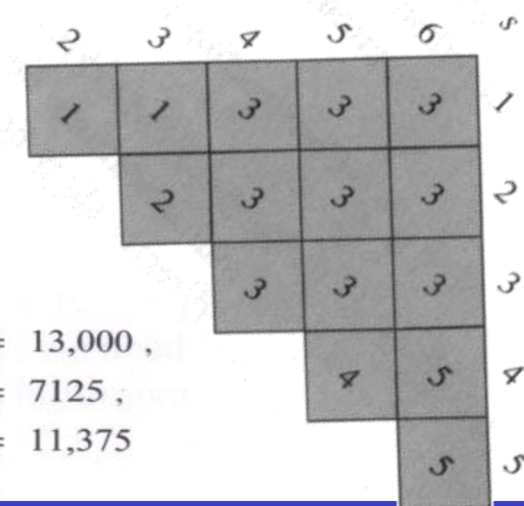
Dynamic Programming: Matrix Multiplication

```
LOOKUP-CHAIN(m, p, i, j)
if m[i, j] < ∞
    return m[i, j]
if i == j
    m[i, j] = 0
else for k = i to j − 1
        q = LOOKUP-CHAIN(m, p, i, k)
             + LOOKUP-CHAIN(m, p, k + 1, j) + p_{i−1} p_k p_j
        if q < m[i, j]
            m[i, j] = q
return m[i, j]
```

```
MATRIX-CHAIN(p)
n = p.length − 1
let m[1..n, 1..n] be a new table
for i = 1 to n
    for j = i to n
        m[i, j] = ∞
return LOOKUP-CHAIN(m, p, 1, n)
```



matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25



$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13{,}000\,, \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125\,, \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11{,}375 \end{cases}$$
$$= 7125\,.$$

Matrix Multiplication: Bottom up, iterative

```
MATRIX-CHAIN-ORDER(p)
n ← length[p] − 1
for i ← 1 to n
    do m[i, i] ← 0
for l ← 2 to n                ▷ l is the chain length.
    do for i ← 1 to n − l + 1
           do j ← i + l − 1
              m[i, j] ← ∞
              for k ← i to j − 1
                  do q ← m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
                     if q < m[i, j]
                        then m[i, j] ← q
                             s[i, j] ← k
return m and s
```

Dynamic Programming: LCS

```
LCS-LENGTH(X, Y)
m = X.length
n = Y.length
let b[1..m, 1..n] and c[0..m, 0..n] be new tables
for i = 1 to m
    c[i, 0] = 0
for j = 0 to n
    c[0, j] = 0
for i = 1 to m
    for j = 1 to n
        if x_i == y_j
            c[i, j] = c[i − 1, j − 1] + 1
            b[i, j] = "↖"
        elseif c[i − 1, j] ≥ c[i, j − 1]
            c[i, j] = c[i − 1, j]
            b[i, j] = "↑"
        else c[i, j] = c[i, j − 1]
            b[i, j] = "←"
return c and b
```

i \ j		0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑0	↑0	↑0	↖1	←1	↖1
2	B	0	↖1	←1	←1	↑1	↖2	←2
3	C	0	↑1	↑1	↖2	←2	↑2	↑2
4	B	0	↖1	↑1	↑2	↑2	↖3	←3
5	D	0	↑1	↖2	↑2	↑2	↑3	↑3
6	A	0	↑1	↑2	↑2	↖3	↑3	↖4
7	B	0	↖1	↑2	↑2	↑3	↖4	↑4

```
PRINT-LCS(b, X, i, j)
if i == 0 or j == 0
    return
if b[i, j] == "↖"
    PRINT-LCS(b, X, i − 1, j − 1)
    print x_i
elseif b[i, j] == "↑"
    PRINT-LCS(b, X, i − 1, j)
else PRINT-LCS(b, X, i, j − 1)
```

Approximation Algorithm: Subset Sum

Algorithm 1: EXACT-SUBSET-SUM(S, t)

$n \longleftarrow |S|$
$L_0 \longleftarrow \langle 0 \rangle$
for i = 1 to n **do**
  $L_i \longleftarrow MergeLists(L_{i-1}, L_{i-1} + x_i)$
  remove from L_i every element greater than t
return the largest element in L_n

Approximation Algorithm

The solution returned is within a factor of $1 + \epsilon$ of the optimal solution.

The running time is polynomial in both n and $1/\epsilon$

Algorithm 2: TRIM(L, δ)

$m \longleftarrow |L|$
$L' \longleftarrow \langle 0 \rangle$
$last \longleftarrow y_1$
for i = 2 to n **do**
  if $y_i > last.(1 + \delta)$ **then**
    append y_i onto the end of L'
    $last \longleftarrow y_i$
return L'

Algorithm 3: APPROX-SUBSET-SUM(S, t, ϵ)

$n \longleftarrow |S|$
$L_0 \longleftarrow \langle 0 \rangle$
for i = 1 to n **do**
  $L_i \longleftarrow MergeLists(L_{i-1}, L_{i-1} + x_i)$
  $L_i \longleftarrow Trim(L_i, \epsilon/2n)$
  remove from L_i every element greater than t;
return the largest element in L_n

Approximation Algorithm: Setcover

GREEDY-SET-COVER($X, \mathcal{F}$)

```
U ← X
C ← ∅
while U ≠ ∅
      do select an S ∈ F that maximizes |S ∩ U|
           U ← U − S
           C ← C ∪ {S}
return C
```

Note: TSP is also covered in the class, in addition we discussed, P,NP, NP complete and NP hard, cook Levin theorem, and idea of reduction