Dynamic Prog.

```
CUT-ROD$(p, n)$
if $n == 0$
    return 0
$q = -\infty$
for $i = 1$ to $n$
    $q = \max(q, p[i] + \text{CUT-ROD}(p, n - i))$
return $q$
```

```
BOTTOM-UP-CUT-ROD$(p, n)$
let $r[0 \,..\, n]$ be a new array
$r[0] = 0$
for $j = 1$ to $n$
    $q = -\infty$
    for $i = 1$ to $j$
        $q = \max(q, p[i] + r[j - i])$
    $r[j] = q$
return $r[n]$
```

```
EXTENDED-BOTTOM-UP-CUT-ROD$(p, n)$
let $r[0 \,..\, n]$ and $s[0 \,..\, n]$ be new arrays
$r[0] = 0$
for $j = 1$ to $n$
    $q = -\infty$
    for $i = 1$ to $j$
        if $q < p[i] + r[j - i]$
            $q = p[i] + r[j - i]$
            $s[j] = i$
    $r[j] = q$
return $r$ and $s$
```

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

```
MEMOIZED-CUT-ROD-AUX$(p, n, r)$
if $r[n] \geq 0$
    return $r[n]$
if $n == 0$
    $q = 0$
else $q = -\infty$
    for $i = 1$ to $n$
        $q = \max(q, p[i] + \text{MEMOIZED-CUT-ROD-AUX}(p, n - i, r))$
$r[n] = q$
return $q$
```

```
MATRIX-MULTIPLY$(A, B)$
if $A.columns \neq B.rows$
    error "incompatible dimensions"
else let $C$ be a new $A.rows \times B.columns$ matrix
    for $i = 1$ to $A.rows$
        for $j = 1$ to $B.columns$
            $c_{ij} = 0$
            for $k = 1$ to $A.columns$
                $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$
    return $C$
```

The pseudocode that follows takes as inputs the probabilities $p_1, \ldots, p_n$ and $q_0, \ldots, q_n$ and the size n, and it returns the tables e and *root*.

```
OPTIMAL-BST$(p, q, n)$
let $e[1 \,..\, n+1, 0 \,..\, n]$, $w[1 \,..\, n+1, 0 \,..\, n]$,
        and $root[1 \,..\, n, 1 \,..\, n]$ be new tables
for $i = 1$ to $n + 1$
    $e[i, i-1] = q_{i-1}$
    $w[i, i-1] = q_{i-1}$
for $l = 1$ to $n$
    for $i = 1$ to $n - l + 1$
        $j = i + l - 1$
        $e[i, j] = \infty$
        $w[i, j] = w[i, j-1] + p_j + q_j$
        for $r = i$ to $j$
            $t = e[i, r-1] + e[r+1, j] + w[i, j]$
            if $t < e[i, j]$
                $e[i, j] = t$
                $root[i, j] = r$
return $e$ and $root$
```

```
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 $x_i == y_j$
            $c[i, j] = c[i-1, j-1] + 1$
            $b[i, j] =$ "$\nwarrow$"
        elseif $c[i-1, j] \geq c[i, j-1]$
            $c[i, j] = c[i-1, j]$
            $b[i, j] =$ "$\uparrow$"
        else $c[i, j] = c[i, j-1]$
            $b[i, j] =$ "$\leftarrow$"
return $c$ and $b$
```

Greedy

```
GREEDY-ACTIVITY-SELECTOR$(s, f)$
$n = s.length$
$A = \{a_1\}$
$k = 1$
for $m = 2$ to $n$
    if $s[m] \geq f[k]$
        $A = A \cup \{a_m\}$
        $k = m$
return $A$
```

```
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
```

```
RECURSIVE-ACTIVITY-SELECTOR$(s, f, k, n)$
$m = k + 1$
while $m \leq n$ and $s[m] < f[k]$        // find the first activity in $S_k$ to finish
    $m = m + 1$
if $m \leq n$
    return $\{a_m\} \cup$ RECURSIVE-ACTIVITY-SELECTOR$(s, f, m, n)$
else return $\emptyset$
```

Disjoint sets

```
CONNECTED-COMPONENTS$(G)$
for each vertex $v \in G.V$
    MAKE-SET$(v)$
for each edge $(u, v) \in G.E$
    if FIND-SET$(u) \neq$ FIND-SET$(v)$
        UNION$(u, v)$
```

```
SAME-COMPONENT$(u, v)$
if FIND-SET$(u) ==$ FIND-SET$(v)$
    return TRUE
else return FALSE
```

-Forests-

```
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 $x.rank == y.rank$
        $y.rank = y.rank + 1$
```

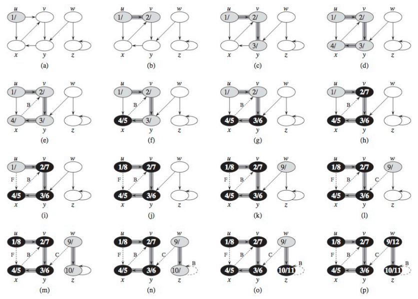
Graphs

```
BFS$(G, s)$
for each vertex $u \in G.V - \{s\}$
    $u.color =$ WHITE
    $u.d = \infty$
    $u.\pi =$ NIL
$s.color =$ GRAY
$s.d = 0$
$s.\pi =$ NIL
$Q = \emptyset$
ENQUEUE$(Q, s)$
while $Q \neq \emptyset$
    $u =$ DEQUEUE$(Q)$
    for each $v \in G.Adj[u]$
        if $v.color ==$ WHITE
            $v.color =$ GRAY
            $v.d = u.d + 1$
            $v.\pi = u$
            ENQUEUE$(Q, v)$
    $u.color =$ BLACK
```

```
PRINT-PATH$(G, s, v)$
if $v == s$
    print $s$
elseif $v.\pi ==$ NIL
    print "no path from" $s$ "to" $v$ "exists"
else PRINT-PATH$(G, s, v.\pi)$
    print $v$
```

```
DFS($G$)
for each vertex $u \in G.V$
    $u.color$ = WHITE
    $u.\pi$ = NIL
$time = 0$
for each vertex $u \in G.V$
    if $u.color$ == WHITE
        DFS-VISIT($G, u$)
```

```
DFS-VISIT($G, u$)
$time = time + 1$
$u.d = time$
$u.color$ = GRAY
for each $v \in G.Adj[u]$
    if $v.color$ == WHITE
        $v.\pi = u$
        DFS-VISIT($G, v$)
$u.color$ = BLACK
$time = time + 1$
$u.f = time$
```



the ***greedy-choice property:*** we can assemble a globally optimal solution by making locally optimal (greedy) choices.

Define the ***shortest-path distance*** delta(s, v) from s to v as the minimum number of edges in any path from vertex s to vertex v

a problem exhibits ***optimal substructure*** if an optimal solution to the problem contains within it optimal solutions to subproblems.

What do we mean by subproblems being independent? We mean that the solution to one subproblem does not affect the solution to another subproblem of the same problem.

Both problems examined in Sections 15.1 and 15.2 have independent subprob- lems. In matrix-chain multiplication, the subproblems are multiplying subchains Ai..Ak and Ak+1..Aj. These subchains are disjoint, so that no ma- trix could possibly be included in both of them.

When a recursive algorithm revisits the same problem repeatedly, we say that the optimization problem has ***overlapping subproblems.***

You will find yourself following a common pattern in discovering optimal sub- structure: 1) You show that a solution to the problem consists of making a choice, such as choosing an initial cut in a rod or choosing an index at which to split the matrix chain. Making this choice leaves one or more subproblems to be solved. 2) You suppose that for a given problem, you are given the choice that leads to an optimal solution. You do not concern yourself yet with how to determine this choice. You just assume that it has been given to you. 3) Given this choice, you determine which subproblems ensue and how to best characterize the resulting space of subproblems. 4) You show that the solutions to the subproblems used within an optimal solution to the problem must themselves be optimal by using a “cut-and-paste” tech- nique. You do so by supposing that each of the subproblem solutions is not optimal and then deriving a contradiction. In particular, by “cutting out” the nonoptimal solution to each subproblem and “pasting in” the optimal one, you show that you can get a better solution to the original problem, thus contradict- ing your supposition that you already had an optimal solution. If an optimal solution gives rise to more than one subproblem, they are typically so similar that you can modify the cut-and-paste argument for one to apply to the others with little effort.

we design greedy algorithms according to the following sequence of steps: 1) Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve. 2) Prove that there is always an optimal solution to the original problem that makes the greedy choice, so that the greedy choice is always safe. 3) Demonstrate optimal substructure by showing that, having made the greedy choice, what remains is a subproblem with the property that if we combine an optimal solution to the subproblem with the greedy choice we have made, we arrive at an optimal solution to the original problem.

the ***greedy-choice property:*** we can assemble a globally optimal solution by making locally optimal (greedy) choices.

How can we tell whether a greedy algorithm will solve a particular optimization problem? No way works all the time, but the greedy-choice property and optimal substructure are the two key ingredients.

MAKE-SET.x/ creates a new set whose only member (and thus representative) is x. Since the sets are disjoint, we require that x not already be in some other set.