1 ▸ Given two strings of characters $u = x_1...x_m$ and $v = y_1...y_n$ drawn from the alphabet $\{a,b,c\}$, we want to know the minimal cost of converting *u* to *v*, where $c_{ch} > 0$ is the cost of changing a symbol, $c_{ins} > 0$ is the cost of inserting a symbol, and $c_{del} > 0$ is the cost of deleting a symbol, and the cost of applying a sequence of operations is the sum of the costs of the operations that comprise the sequence. For example, if $c_{ch} = c_{ins} = c_{del} = 1$ and *u*=*abbaac* and *v*=*abcbc*, then the cost of converting *u* to *v* is 3 because of the sequence

$$abbaac \underset{c_{del}}{\to} abbac \underset{c_{ch}}{\to} abcac \underset{c_{ch}}{\to} abcbc \quad .$$

Write a dynamic programming algorithm to accept as input *u*, *v*, c_{ch}, c_{ins} and c_{del} and return the minimal cost of a sequence of operations to convert *u* to *v*.

<u>SOLUTION</u>: For $0 \le i \le m$, $0 \le j \le n$, we let $\delta(i, j)$ denote the minimal cost of a sequence of operations to convert $x_1...x_i$ to $y_1...y_j$. The answer to the problem is $\delta(m,n)$. To start the process, we note that $\delta(0,0) = 0$, $\delta(1,0) = c_{del}$ and $\delta(0,1) = c_{ins}$. For $1 \le i \le m \wedge 1 \le j \le n$, if $x_i=y_j$, then $\delta(i, j) = \delta(i-1, j-1)$. Otherwise, $\delta(i, j) = \min(\delta(i-1, j-1) + c_{ch}, \delta(i-1, j) + c_{del}, \delta(i, j-1) + c_{ins})$.

The corresponding program to fill in array $\delta[0..m, 0..n]$ is

$\delta[0,0] \leftarrow 0$
$\delta[1,0] = c_{del}$
$\delta[0,1] = c_{ins}$
for $i \leftarrow 1$ **to** m
  for $j \leftarrow 1$ **to** n
    if $x_i=y_j$ **then** $\delta[i, j] \leftarrow \delta[i-1, j-1]$
    else $\delta[i, j] \leftarrow \min(\delta[i-1, j-1] + c_{ch}, \delta[i-1, j] + c_{del}, \delta[i, j-1] + c_{ins})$
return $\delta[m,n]$

****

2 ▸ An instance of the PARTITION PROBLEM is a set U of integers, and the question is whether or not *U* can be partitioned into two sets *T* and *U**T* such that the sums of the integers in the two sets are equal. That is, does $\sum_{s \in T} s = \sum_{s \in U \setminus T} s = \frac{\sum_{s \in U} s}{2}$. For example, the instance $U = \{2,3,9,15,19\}$ admits the solution *U*={9,15}, and the instance $U = \{2,3,9,15,18\}$ does not admit a solution. Give a dynamic programming solution to the PARTITION PROBLEM, and analyze your solution.

<u>SOLUTION</u>: The instance $U = \{s_1, ..., s_n\}$ of the PARTITION PROBLEM admits a solution if and only if the instance *v*=*w*=$(s_1, ..., s_n)$, $W = \frac{\sum_{1 \le i \le n} s_i}{2}$ of the KNAPSACK PROBLEM admits a packing of value *W*. The algorithm takes time in $\Theta\left(n \sum_{1 \le i \le n} s_i\right)$.

****

3 ▸ Describe a $O(n^2)$ dynamic programming algorithm to find the length of the longest (not necessarily contiguous) increasing sequence of integers of $A[1..n]$. For example, if $A=(11,17,5,8,6,4,7,7,12,3)$, then the answer would be 4 because of the subsequence $(5,6,7,12)$.

SOLUTION: For every i, we compute *Length_of_longest*[i], the length of the longest increasing subsequence in $A[1..i]$ whose rightmost member is $A[i]$. The dynamic programming formulation is

$$Length\_of\_longest[i] = \max_{\substack{1\le j<i \\ A[j]<A[i]}} \{1, Length\_of\_longest[j]+1\}$$

This translates to the program

$Length\_of\_longest[1] \leftarrow 1$
for $i \leftarrow 2$ **to** n **do**
  $Length\_of\_longest[i] \leftarrow 1$
  for $j \leftarrow 1$ **to** i-1 **do**
    if $A[j] < A[i]$ **and** *Length_of_longest*[j]+1> *Length_of_longest*[i]
    then *Length_of_longest*[i]← *Length_of_longest*[j]+1
return $\max_{1\le i\le n}\{Length\_of\_longest[i]\}$

****

4 ▸ We are given $A=(a_1,...,a_n)$, where $a_i \in \mathbb{Z}^+$, $1\le i\le n$ and n>3. We define a set S of elements of A to be *independent* if $a_j, a_l \in S$ implies that $|j-i|>1$. That is, adjacent elements of A do not belong to S. We seek to compute an independent set of A with maximum sum. For example, if A=(11, 2, 16, 18, 3, 2), then S={11, 18, 2}. Find an algorithm polynomial in n (not in the individual values a_i) to solve the problem.

SOLUTION: Letting $f(i)$, $1\le i\le n$, denote the optimal value of a set S_i for $A=(a_1,...,a_i)$ where S_i **must include** a_i, we derive a dynamic programming solution for this problem. First we note that for i>3, S_i cannot include a_{i-1} and must include exactly one of a_{i-2} and a_{i-3}.

$f(1) \leftarrow a_1$
$f(2) \leftarrow a_2$
$f(3) \leftarrow a_1 + a_3$
for $i \leftarrow 4$ **to** n **do** $f(i) \leftarrow a_i + \max(f(i-2), f(i-3))$
return $\max(f(n-1), f(n))$

The time complexity of this algorithm is in $\Theta(n)$.

****

5 ▸ Consider the array $V[0..n, 0..W]$ computed by the dynamic programming algorithm to solve the 0/1-KNAPSACK PROBLEM. Either prove or give a counter example to each of the following.

a CONJECTURE 1: For any instance of the problem and any j, $0\le j\le n$, and any x,y, $0\le x<y\le W$, $V[j,x]\le V[j,y]$.

b CONJECTURE 2: For any instance of the problem and any j, k, $0\le j<k\le n$, and any x, $0\le x\le W$, $V[j,x]\le V[k,x]$.

SOLUTIONS: ***a*** The question asks if each row of V is weakly monotonically increasing. As a basis for a proof by induction, we note that the top row, j=0, is all 0s and hence is weakly monotonically increasing. If row j-1 is weakly monotonically increasing, then we have to show that this implies that $V[j,x] \geq V[j,x-1]$ for all $1 \leq x \leq W$. But $V[j,x] = \max\left(V[j-1,x], V\left[j-1, x-w[j]\right]+v[j]\right)$ and $V[j,x-1] = \max\left(V[j-1,x-1], V\left[j-1,x-1-w[j]\right]+v[j]\right)$. By the induction hypothesis, $V[j-1,x] \geq V[j-1,x-1]$ and $V\left[j-1,x-w[j]\right] \geq V\left[j-1,x-1-w[j]\right]$. Hence $V[j,x] \geq V[j,x-1]$.

b The question asks if each column of V is weakly monotonically increasing. As a basis for a proof by induction, we note that the leftmost column, x=0, is all 0s and hence is weakly monotonically increasing. If column x-1 is weakly monotonically increasing, then we have to show that this implies that $V[j,x] \geq V[j-1,x]$ for all $1 \leq j \leq n$. But $V[j,x] = \max\left(V[j-1,x], V\left[j-1,x-w[j]\right]+v[j]\right)$ and $V[j,x-1] = \max\left(V[j-1,x-1], V\left[j-1,x-1-w[j]\right]+v[j]\right)$. By the induction hypothesis, $V[j-1,x] \geq V[j-1,x-1]$ and $V\left[j-1,x-w[j]\right] \geq V\left[j-1,x-1-w[j]\right]$. Hence $V[j,x] \geq V[j,x-1]$.
****

6 ▸ Show how to solve the 0/1-KNAPSACK PROBLEM and return both the value of an optimal solution **and** the number of optimal solutions.

SOLUTION: Let $V[j,x]$, $0 \leq j \leq n, 0 \leq x \leq W$, be value of optimal packing of knapsack of capacity x using only objects $\subseteq \{1,\ldots,j\}$, and let $N[j,x]$, $0 \leq j \leq n, 0 \leq x \leq W$, be the number of optimal packings of knapsack of capacity x using only objects $\subseteq \{1,\ldots,j\}$

for $j \leftarrow 0$ **to** n **do** { $V[j,0] \leftarrow 0$ ▸ W=0, can't carry any weight
$N[j,0] \leftarrow 1$} ▸ There's one empty packing
for $x \leftarrow 0$ **to** n **do** { $V[0,x] \leftarrow 0$ ▸ j=0, can't carry any objects
$N[0,x] \leftarrow 1$} ▸ There's one empty packing
for $j \leftarrow 1$ **to** n **do**
  for $x \leftarrow 1$ **to** W **do**
    if (x - $w[j]$ >= 0)
    then { $V[j,x] \leftarrow \max\left(V[j-1,x], V\left[j-1,x-w[j]\right]+v[j]\right)$
      if $V[j-1,x] = V\left[j-1,x-w[j]\right]+v[j]$
        then $N[j,x] \leftarrow N[j-1,x] + N\left[j-1,x-w[j]\right]$
      else if $N[j-1,x] > N\left[j-1,x-w[j]\right]+v[j]$
          then $N[j,x] \leftarrow N[j-1,x]$
          else $N[j,x] \leftarrow N\left[j-1,x-w[j]\right]$
    else { $V[j,x] \leftarrow V[j-1,x]$, $N[j,x] \leftarrow N[j-1,x]$ }
    return $V[n,W], N[n,W]$
****

7 ▸ A typical dynamic programming algorithm provides the cost of a solution or establishes the existence of a solution without actually constructing the solution. To see how to construct a solution by using an efficient mechanism which tests for the existence of a solution, solve the following:

You are given a boolean function *BlackBox* of two inputs:

-a list of integers $x_1,...,x_n$,

-an integer q,

and you are told that, in time $O(1)$, *BlackBox* will return true if there is some subset of $x_1,...,x_n$ whose sum is q and false otherwise. Design an algorithm (a program is not needed) with the same input which will return an actual subset of $x_1,...,x_n$ whose sum is q, if such a subset exists, or else it should return "failure".

For example, *BlackBox*((23,27,41,72,-4,6), 29) would return "true", but your algorithm with the same input would return (27,-4,6) or (23,6). Your algorithm may call *BlackBox* as often as it wishes and it should work in time $O(n)$.

SOLUTION:

$S \leftarrow (x_1,x_2,...,x_n)$

if not *BlackBox* (S, q) **then return** ("failure") $O(1)$

for $i \leftarrow 1$ **to** n **do** n times

$S \leftarrow S - x_i$

if not *BlackBox* (S, q) ▹ we really need x_i ; put it back

$S \leftarrow S + x_i$;

return S ; $O(1)$

Since the body of the loop is executed in $O(1)$ time, the time to execute the loop (and the program) is $O(n)$.

****

8 ▸ Let a country's currency be coins worth c_1¢, c_2¢,…, c_n¢. We seek an algorithm which accepts as input $(c_1,...,c_n;x)$ and which gives as output a **minimal** number of coins, drawn from $(c_1,...,c_n)$, such that the sum of the values of the coins is x¢. So, for example, for $(1,5,10,25,50;156)$ the answer would be $(50,50,50,5,1)$.

a One algorithm is

GREED$(c_1,...,c_n,x)$

if x>0 **then** {let c_i be $\max(c_1,...,c_n)$ such that $c_i \le x$

give c_i

GREED$(c_1,...,c_n,x-c_i)$

GREED works for $(1,5,10,25,50;x)$ for any x. Give an instance of the problem, $(c_1,...,c_n;x)$, for which GREED does not work.

b Give an algorithm which works for any $(c_1,...,c_n;x)$. The time complexity of your algorithm should be in $O(nx)$.

SOLUTION: ***a*** For $(1,4,6;8)$ GREED will return $(6,1,1)$ although the answer is $(4,4)$.

b For $0 \le m \le x$ we let $\kappa(m)$ denote the minimum number of coins to give m¢. If this minimum number of coins includes a c_i¢ coin, then, by the Optimality Principle, we use $\kappa(m-c_i)$ coins to give $(m-c_i)$¢.

$$\kappa(m) = 1 + \min_{1\le i\le n}\{\kappa(m-c_i)\}$$

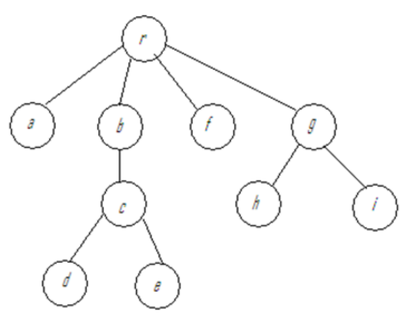
In the dynamic programming formulation, it is understood that $\kappa(m)=0$ if $m \le 0$.

for $m \leftarrow 1$ **to** x **do**
    $\kappa[m] \leftarrow \infty$
    for $i \leftarrow 1$ **to** n **do**
        if $m>c_i$ **then** $\kappa(m) = \min(\kappa(m), 1+\kappa(m-c_i))$
PRINTANSWER(κ, x)

PRINTANSWER(κ, m)
    if $m>0$ **then**
    $i \leftarrow 1$
    repeat
        if $m>c_i \wedge \kappa[m] = 1+\kappa[m-c_i]$
        then {Print c_i
            return PRINTANSWER$(\kappa, m-c_i)$ }
        else $i \leftarrow i+1$

****

9 ▸ An *independent set* of vertices of a graph $G=(V,E)$ is a set of vertices such that there does not exist an edge between any pair of vertices of the set. As stated in **Problem *34-1*** on page 1018 of our text, finding a maximum independent set of a graph is very difficult. However, many problems which are difficult in graphs become easier if we restrict the graph to be a tree. Describe an algorithm, with time complexity in $O(m+n)$, to find a maximum independent set in a tree. So MAXINDEPENDENTSET(r) would return $\{a,b,d,e,f,h,i\}$ on



SOLUTION: For each node v in the tree, we compute $\iota(v)$, the maximum number of independent nodes in the tree rooted at v. We compute $\iota(v)$ from the bottom up in the tree, so that when computing $\iota(v)$ we have already computed $\iota(w)$ for all children and grandchildren w of v. We note that if v is in a maximum independent set of the tree rooted at *v*, then none of its children is in the maximum independent set. The dynamic programming recurrence is

$$\iota(v) = \max\left\{ \sum_{\text{children } w \text{ of } v} \iota(w),\ 1 + \sum_{\text{grandchildren } w \text{ of } v} \iota(w) \right\}$$