General Approach to Solving optimization problems using Dynamic Programming

1. characterize the structure of an opt. solution

2. Recursively define the value of an opt. solution

3. Compute the value of an opt. solution in a bottom up fashion

4. Construct an opt. sol. from computed information

Problem Statement

- We have $\underline{1}$ resource
- " " $\underline{n}$ requests labeled $\underline{1}$ to $\underline{n}$
- Each request has start time s_i,
  finish time f_i, and
  weight w_i

Goal: Select a subset $S \subseteq \{1..n\}$
of mutually compatible intervals
so as to Maximize $\sum_{i \in S} w_i$

Case 1 - if it is, value of the opt. sol. = w_i + value of the opt. sol. for the subproblem that consists only of compatible requests with i

Case 2 - if it isn't, value of the opt. sol. = value of the opt. sol. without job i

Sort requests in order of non-decreasing finish time.

$$f_1 \leq f_2 \leq \cdots \leq f_n$$

Define $P(j)$ for an interval j to be the largest index $i < j$ such that interval i & j are disjoint.

Def. Let O_j denote the opt. solutions to the problem consisting of requests $\{1..j\}$
Let $OPT(j)$ denote the value of O_j

Solution:

```
Compute-opt (j)
    if j = 0 then
        return 0
    else
        return
```

Memoization

Store the value of compute-opt. in a globally accessible place the first time we compute it. Then simply use this precomputed value in place of all future recursive calls.

M-compute-opt (j)
  if $j=0$ then
    return 0
  else if $M[j]$ is not empty then

```
Find-Solution
    if j > 0 then
        if w_j + M[p(j)] ≥ M[j-1] then

            output j together w/ the results
              of Find-Solution ( p(j) )
        else
            output the results of
                  Find-solution (j-1)
        endif
    endif
```

Videogame Problem

0-1 knapsack &

subset Sum

Problem Statement

- A single resource
- Requests {1..n} each take time w_i to process
- Can schedule jobs at any time between $\underline{0}$ to $\underline{W}$

Objective: To schedule jobs such that we maximize the machine's utilization

$OPT(i, w)$ = value of the opt. solution using a subset of the items $\{1..i\}$ with max. allowed weight $\underline{w}$.

if $n \notin O$, Then $OPT(n, w) =$

if $n \in O$, Then $OPT(n, w) =$

If $w < w_i$, then $OPT(i, w) = OPT(i-1, w)$

else, $OPT(i, w) = \max(OPT(i-1, w),$
$w_i + OPT(i-1, w - w_i))$

```
Subset-sum (n, w)
  array M[0,w]=0  for each w=0 to W
  for i=1 to n
     for w=0 to W
         use recurrence formula
         to compute M[i,w]
     endfor
  endfor
  Return M[n,W]
```

Pseudo-polynomial time

An algorithm runs in pseudo-polynomial time if its running time is a polynomial in the numeric value of the input

Polynomial time

An algorithm runs in polynomial time if its running time is a polynomial in the length of the input (or output).

Discussion 6

1. You are to compute the total number of ways to make a change for a given amount m. Assume that we have an unlimited supply of coins and all denominations are sorted in ascending order: $1 = d_1 < d_2 < \ldots < d_n$. Formulate the solution to this problem as a dynamic programming problem.

2. Graduate students get a lot of free food at various events. Suppose you have a schedule of the next *n* days marked with those days when you get a free dinner, and those days on which you must acquire dinner on your own. On any given day you can buy dinner at the cafeteria for $3. Alternatively, you can purchase one week's groceries for $10, which will provide dinner for each day that week (that day and the six that follow). However, because you don't have a fridge, the groceries will go bad after seven days (including the day of purchase) and any leftovers must be discarded. Due to your very busy schedule, these are your only two options for dinner each night. Your goal is to eat dinner every night while minimizing the money you spend on food.

3. You are in Downtown of a city and all the streets are one-way streets. You can only go east (right) on the east-west (left-right) streets, and you can only go south (down) on the north-south (up-down) streets. This is called a Manhattan walk.

a) In Figure A below, how many unique ways are there to go from the intersection marked S (coordinate (0,0)) to the intersection marked E (coordinate (n,m))?
Formulate the solution to this problem as a dynamic programming problem. Please make sure that you include all the boundary conditions and clearly define your notations you use.

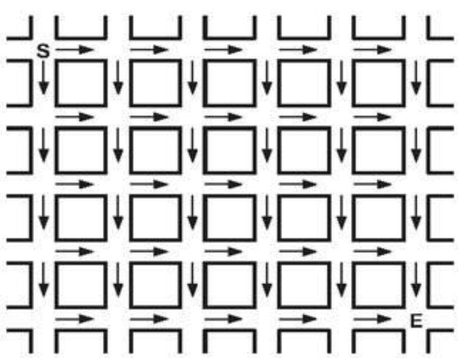

Figure A.

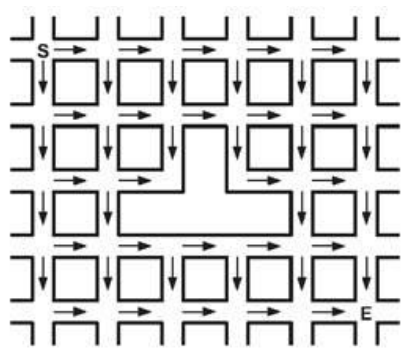

Figure B.

b) Repeat this process with Figure B; be wary of dead ends.

Assume you want to ski down the mountain. You want the total length of your run to be as long as possible, but you can only go down, i.e. you can only ski from a higher position to a lower position. The height of the mountain is represented by an *n* X *n* matrix A. A[i][j] is the height of the mountain at position (i,j). At position (i,j), you can potentially ski to four adjacent positions (i-1,j) (i,j-1), (i,j+1), and (i+1,j) (only if the adjacent position is lower than current position). Movements in any of the four directions will add 1 unit to the length of your run. Provide a dynamic programming solution to find the longest possible downhill ski path starting at any location within the given *n* by *n* grid.

1200	1000	1200	1500	1700	1500	1000	1000
1100	1600	2000	1900	1800	1600	1200	1250
1200	1700	1900	2300	2400	2000	1900	1750
1000	1500	2000	2450	2600	2100	2000	1500
1100	1500	1800	2200	2300	2200	2100	1600
1100	1000	1500	1800	2100	1900	2000	1700
1000	1000	1200	1300	1700	1900	1900	1800
900	800	1000	1200	1500	1900	2000	2100

Imagine starting with the given decimal number n, and repeatedly chopping off a digit from one end or the other (your choice), until only one digit is left. The square-depth SQD(n) of n is defined to be the maximum number of perfect squares you could observe among all such sequences. For example, SQD(32492) = 3 via the sequence

$$32492 \rightarrow \mathbf{3249} \rightarrow \mathbf{324} \rightarrow 24 \rightarrow \mathbf{4}$$

since 3249, 324, and 4 are perfect squares, and no other sequence of chops gives more than 3 perfect squares. Note that such a sequence may not be unique, e.g.

$$32492 \rightarrow \mathbf{3249} \rightarrow 249 \rightarrow \mathbf{49} \rightarrow \mathbf{9}$$

also gives you 3 perfect squares, viz. 3249, 49, and 9.
Describe an efficient algorithm to compute the square-depth SQD(n), of a given number n, written as a d-digit decimal number $a_1 a_2 \ldots a_d$. Analyze your algorithm's running time. Your algorithm should run in time polynomial in d. You may assume the availability of a function IS_SQUARE(x) that runs in constant time and returns 1 if x is a perfect square and 0 otherwise.