## binary search pseudocode

binary\_search(A, target):

lo = 1, hi = size(A)

while lo <= hi:

mid = lo + (hi-lo)/2

if A[mid] == target:

return mid

else if A[mid] < target:

lo = mid+1

else:

hi = mid-1

// target was not found

## coin change dp

Set Min[i] equal to Infinity for all of i

Min[0]=0

For i = 1 to S

For j = 0 to N - 1

If (Vj<=i AND Min[i-Vj]+1<Min[i])

Then Min[i]=Min[i-Vj]+1

Output Min[S]

## Non-decreasing sequence dp

Given a sequence of N numbers - **A[1]** , **A[2]** , ..., **A[N]** . Find the length of the longest non-decreasing sequence.   
  
As described above we must first find how to define a "state" which represents a sub-problem and thus we have to find a solution for it. Note that in most cases the states rely on lower states and are independent from greater states.   
  
Let's define a state **i** as being the longest non-decreasing sequence which has its last number **A[i]** . This state carries only data about the length of this sequence. Note that for **i<j** the state **i** is independent from **j**, i.e. doesn't change when we calculate state **j**. Let's see now how these states are connected to each other. Having found the solutions for all states lower than **i**, we may now look for state **i**. At first we initialize it with a solution of 1, which consists only of the **i-th** number itself. Now for each **j<i** let's see if it's possible to pass from it to state i. This is possible only when **A[j]≤A[i]** , thus keeping (assuring) the sequence non-decreasing. So if **S[j]** (the solution found for state **j**) + **1** (number **A[i]** added to this sequence which ends with number **A[j]** ) is better than a solution found for **i** (ie. **S[j]+1>S[i]** ), we make **S[i]=S[j]+1**. This way we consecutively find the best solutions for each **i**, until last state N.   
Let's see what happens for a randomly generated sequence: 5, 3, 4, 8, 6, 7:

|  |  |  |
| --- | --- | --- |
| I | The length of the longest non-decreasing sequence of first i numbers | The last sequence i from which we "arrived" to this one |
| 1 | 1 | 1 (first number itself) |
| 2 | 1 | 2 (second number itself) |
| 3 | 2 | 2 |
| 4 | 3 | 3 |
| 5 | 3 | 3 |
| 6 | 4 | 5 |

## Knapsack DP

// Returns the maximum value that can be put in a knapsack of capacity W

int knapSack(int W, int wt[], int val[], int n) {

int i, w;

int K[n+1][W+1];

// Build table K[][] in bottom up manner

for (i = 0; i <= n; i++)

{

for (w = 0; w <= W; w++)

{

if (i==0 || w==0)

K[i][w] = 0;

else if (wt[i-1] <= w)

K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]], K[i-1][w]);

else

K[i][w] = K[i-1][w];

}

}

return K[n][W];

}

int main() {

int val[] = {60, 100, 120};

int wt[] = {10, 20, 30};

int W = 50;

int n = sizeof(val)/sizeof(val[0]);

printf("%d", knapSack(W, wt, val, n));

return 0;

}

**// WIKIPEDIA’s**

// Values (stored in array v)

// Weights (stored in array w)

// Number of distinct items (n)

// Knapsack capacity (W)

for w from 0 to W do

m[0, w] := 0

end for

for i from 1 to n do

for j from 0 to W do

if j >= w[i] then

m[i, j] := max(m[i-1, j], m[i-1, j-w[i]] + v[i])

else

m[i, j] := m[i-1, j]

end if

end for

end for