Dynamic Programming

Divide-and-conquer algorithms decompose problems into subproblems, and then combine solutions to subproblems to obtain solutions for the larger problems.

Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems.

However, during the divide part of divide-and-conquer some subproblems could appear more than once.

Dynamic programming

However, during the divide part of divide-and-conquer some subproblems could appear more than once.

If subproblems are duplicated, the computation of the solution of the subproblems is also duplicated, solving the same subproblems several times obviously yield very poor running times.

Dynamic programming algorithms avoid recomputing the solution of same subproblems by storing the solution of subproblems the first time they are computed, and referring to the stored solution when needed.

Example : Fibonacci numbers

A divide-and-conquer algorithm :

$$\text{Fib}(n) = \begin{cases} 0 & \text{if } n{=}0 \\ 1 & \text{if } n{=}1 \\ \text{Fib}(n-1) + \text{Fib}(n-2) & \text{if } n > 1 \end{cases}$$

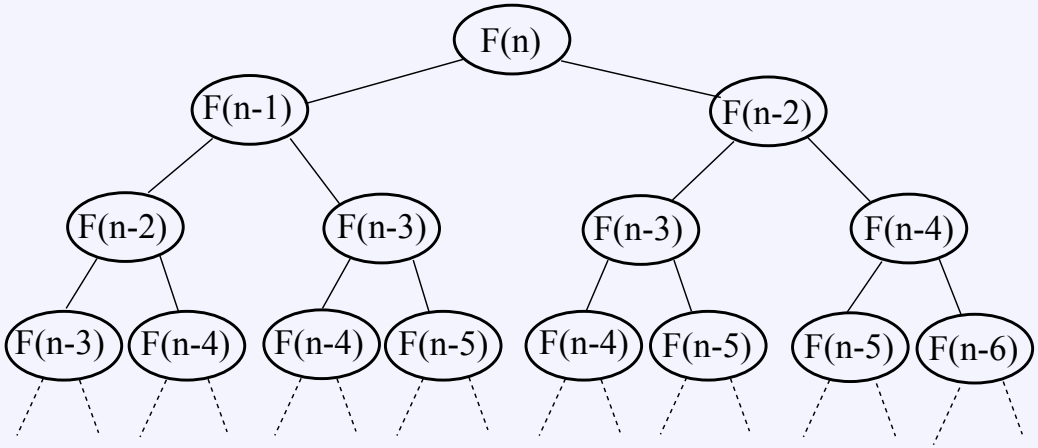
To compute Fibonacci of 7 (Fib(7)), the following decompositions take place :

$$\begin{array}{lcllcllcl} Fib(0) & = & 0 & & & & & & \\ Fib(1) & = & 1 & & & & & & \\ Fib(2) & = & Fib(0) + Fib(1) & = & 0+1 & = & 1 & & \\ Fib(3) & = & Fib(1) + Fib(2) & = & 1+1 & = & 2 & & \\ Fib(4) & = & Fib(2) + Fib(3) & = & 1+2 & = & 3 & & \\ Fib(5) & = & Fib(3) + Fib(4) & = & 2+3 & = & 5 & & \\ Fib(6) & = & Fib(4) + Fib(5) & = & 3+5 & = & 8 & & \\ Fib(7) & = & Fib(5) + Fib(6) & = & 5+8 & = & 13 & & \end{array}$$

Another view of Fibonacci

function *Fib*(n)
  if ($n \leq 1$) **then return** n;
  else
    return(Fib($n-1$) + Fib($n-2$));

The divide-and-conquer algorithm generates the following call tree :



The running time of Fib is $O((\frac{1+\sqrt{5}}{2})^n)$, it grows exponentially.

Dynamic Programming for Fibonacci

1. Divide-and-conquer :

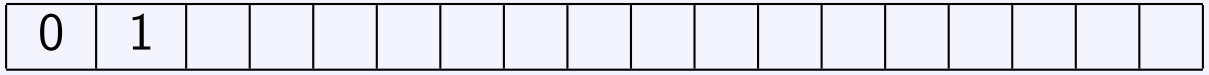
```
function Fib_rec(n)
        if (n ≤ 1) then return n ;
        else
          return(Fib_rec(n − 1) + Fib_rec(n − 2)) ;
```

2. Dynamic Programming :

```
function  fib_dyn(n)
        int *f, i ;
        f = malloc((n + 1) ∗ sizeof(int)) ;
        for (i = 0; i ≤ n; i + +)
            if (i ≤ 1)
              f[i] = i ;
            else
              f[i] = f[i − 1] + f[i − 2] ;
        return f[n] ;
```

Dynamic Programming for Fibonacci

```
function fib_dyn(n)
        int *f, i;
        f = malloc((n + 1) * sizeof(int));
        for (i = 0; i ≤ n; i++)
            if (i ≤ 1)
                f[i] = i;
            else
                f[i] = f[i − 1] + f[i − 2];
        return f[n];
```



$fib\_dyn \in \Theta(n)$ as opposed to the exponential complexity $O((\frac{1+\sqrt{5}}{2})^n)$ for *fib_rec*.

Summary

Instead of solving the same subproblem repeatedly, arrange to solve each subproblem only one time

Save the solution to a subproblem in a table, and refer back to the table whenever we revisit the subproblem

"Store, don't recompute"

0	1	2	3	4	5	6	7	8	9	10
0	1	2	3							

- here computing Fib(4) and Fib(5) both require Fib(3), but Fib(3) is computed only once

Can turn an exponential-time solution into a polynomial-time solution

When do we need DP

Dynamic programming is useful because it solves each subproblem only once.

Before writing a dynamic programming algorithm, first do the following :

- Write a divide-and-conquer algorithm to solve the problem
- Next, analyze its running time, if it is exponential then :
  - it is likely that the divide-and-conquer generates a large number of identical subproblems
  - therefore solving many times the same subproblems

If D&C has poor running times, we can consider DP.

But successful application of DP requires that the problem satisfies some conditions, which will be introduced later...

Writing a DP algorithm : the bottom-up approach

Create a table that will store the solution of the subproblems

Use the "base case" of D&C to initialize the table

Devise look-up template using the recursive calls of the D&C algorithm

Devise for-loops that fill the table using look-up template

The function containing the for loop returns the last entry that has been filled in the table.

An example : making change

Devise an algorithm for paying back a customer a certain amount using the smallest possible number of coins.

For example, what is the smallest amount of coins needed to pay back \$2.89 (289 cents) using as denominations "one dollars", "quaters", "dimes" and "pennies".

The solution is 10 coins, i.e. 2 one dollars, 3 quaters, 1 dime and 4 pennies.

Making change : a recursive solution

Assume we have an infinite supply of n different denominations of coins.

A coin of denomination i worth d_i units, $1 \leq i \leq n$.

We need to return change for N units.

function Make_Change(i,j)
  if $(j == 0)$ **then return** 0 ;
  else
    return min(make_change$(i-1, j)$, make_change$(i, j-d_i)+1)$;

The function is called initially as Make_Change(n,N).

Making change : DP approach

Assume $n = 3$, $d_1 = 1$, $d_2 = 4$ and $d_3 = 6$. Let $N = 8$.

To solve this problem by dynamic programming we set up a table $t[1..n, 0..N]$, one row for each denomination and one column for each amount from 0 unit to N units.

Amount	0	1	2	3	4	5	6	7	8
$d_1 = 1$									
$d_2 = 4$									
$d_3 = 6$									

The entry $t[i, j]$ indicates the minimum number of coins needed to refund an amount of j units using only coins from denominations 1 to i.

Making change : DP approach

The initialization of the table is obtained from the D&C base case :

if $(j == 0)$ **then return** 0

i.e. $t[i, 0] = 0$, for $i = 1, 2, 3$

Amount	0	1	2	3	4	5	6	7	8
$d_1 = 1$	0								
$d_2 = 4$	0								
$d_3 = 6$	0								

Making change : DP approach

If $i = 1$, then only denomination $d_1 = 1$ can be used to return change.

Therefore $t[1, j]$ for $j = 1..8$ is $t[i, j] = t[i, j - d_i] + 1$

Amount	0	1	2	3	4	5	6	7	8
$d_1 = 1$	0	1	2	3	4	5	6	7	8
$d_2 = 4$	0								
$d_3 = 6$	0								

For example, the content of entry $t[1, 4] = t[1, 3] + 1$ means that the minimum number of coins to return 4 units using only denomination 1 is the minimum number of coins to return 3 units + 1 = 4 coins.

Making change : DP approach

If the amount of change to return is smaller than domination d_i, then the change needs to be return using denominations smaller than d_i

For those cases, i.e. if $(j < d_i)$ then $t[i, j] = t[i - 1, j]$

Amount	0	1	2	3	4	5	6	7	8
$d_1 = 1$	0	1	2	3	4	5	6	7	8
$d_2 = 4$	0	1	2	3					
$d_3 = 6$	0	1	2	3	1	2			

For all the other entries of the table we write the code of the DP algorithm using the recursive function

Making change : DP approach

The recursive function Make_Change

function Make_Change(i,j)
  if $(j == 0)$ **then return** 0 ;
  else
    return min(make_change$(i-1, j)$, make_change$(i, j-d_i)+1$) ;

tells us that to fill entry $t[i, j], j > 0$, we have two choices :

1. Don't use a coin from d_i, then $t[i, j] = t[i-1, j]$
2. Use at least one coin from d_i, then $t[i, j] = t[i, j-d_i] + 1$.

The recursive function also tells us that we take the min of these two values :

$$t[i, j] = \min(t[i-1, j], t[i, j-d_i] + 1)$$

The DP algorithm

coins(n, N)
  int $d[1..n] = d[1, 4, 6]$;
  int $t[1..n, 0..N]$;
  for $(i = 1; i \leq n; i++)$ $t[i, 0] = 0$; */base case */
  for $(i = 1; i \leq n; i++)$
    for $(j = 1; j \leq N; j++)$
      if $(i = 1)$ **then** $t[i, j] = t[i, j - d_i] + 1$
      else if $(j < d[i])$ **then** $t[i, j] = t[i - 1, j]$
      else $t[i, j] = \min(t[i - 1, j], t[i, j - d[i]] + 1)$
  return $t[n, N]$;

The algorithm runs in $\Theta(nN)$.

Making change : DP approach

Amount	0	1	2	3	4	5	6	7	8
$d_1 = 1$	0	1	2	3	4	5	6	7	8
$d_2 = 4$	0	1	2	3	1	2	3	4	2
$d_3 = 6$	0	1	2	3	1	2	1	2	2

To fill entry $t[i, j], j > 0$, we have two choices :

1. Don't use a coin from d_i, then $t[i, j] = t[i - 1, j]$
2. Use at least one coin from d_i, then $t[i, j] = t[i, j - d_i] + 1$.

Since we seek to minimize the number of coins return, we have

$$t[i, j] = \min(t[i - 1, j], t[i, j - d_i] + 1)$$

The solution is in entry $t[n, N]$

Making change : getting the coins

Amount	0	1	2	3	4	5	6	7	8
$d_1 = 1$	0	1	2	3	4	5	6	7	8
$d_2 = 4$	0	1	2	3	1	2	3	4	2
$d_3 = 6$	0	1	2	3	1	2	1	2	2

We can use the information in the table to get the list of coins that should be returned :

- Start at entry $t[n, N]$;
- If $t[i, j] = t[i-1, j]$ then no coin of denomination i has been used to calculate $t[i, j]$, then move to entry $t[i-1, j]$;
- If $t[i, j] = t[i, j-d_i] + 1$, then add one coin of denomination i and move to entry $t[i, j-d_i]$.

Optimal Substructure

DP is often used to solve optimization problems that have the following form

$$\begin{array}{ll} \min & f(x) \textbf{ or} \\ \max & f(x) \\ \text{s.t.} & \textit{some constraints} \end{array} \tag{1}$$

Making change is an optimization problem.

The function $f(x)$ to minimize is the number of coins

The constraint is the sum of the value of the coins is equal to the amount to return

Optimal Substructure

In solving optimization problems with DP, we find the optimal solution of a problem of size n by solving smaller problems of same type

The optimal solution of the original problem is made of optimal solutions from subproblems

Thus the subsolutions within an optimal solution are optimal subsolutions

Solutions to optimization problems that exhibit this property are say to be based on optimal substructures

Optimal Substructure

Make_Change() exhibits the optimal substructure property :

- The optimal solution of problem (i, j) is obtained using optimal solutions (minimum number of coins) of sub-problems $(i-1, j)$ and $(i, j-d_i)$.

Amount	0	1	2	3	4	5	6	7	8
$d_1 = 1$	0	1	2	3	4	5	6	7	8
$d_2 = 4$	0	1	2	3	1	2	3	4	2
$d_3 = 6$	0	1	2	3	1	2	1	2	2

Each entry $t[i, j]$ in the table is the optimal solution (minimum number of coins) that can be used to return an amount of j units using only denominations d_1 to d_i.

The optimal solution for $t[i, j]$ is obtained by comparing $t[i-1, j]$ and $t[i-1, j]$, taking the smallest of the two.

Optimal Substructure

To compute the optimal solution, we can compute all optimal subsolutions

Often we start with all optimal subsolutions of size 1, then compute all optimal subsolutions of size 2 combining some subsolutions of size 1. We continue in this fashion until we have out solution for n.

Note, optimal substructure does not apply to all optimization problems. When it fails to apply, we cannot use DP.

DP for optimization problems

The basic steps are :

- Characterize the structure of an optimal solution.
- Give a recursive definition for computing the optimal solution based on optimal solutions of smaller problems.
- Compute the optimal solutions and/or the value of the optimal solution in a bottom-up fashion.

Integer 0-1 Knapsack problem

Given n objects with integer weights w_i and values v_i, you are asked to pack a knapsack with no more than W weight (W is integer) such that the load is as valuable as possible (maximize). You cannot take part of an object, you must either take an object or leave it out.

Example : Suppose we are given 4 objects with the following weights and values :

Object	1	2	3	4
Weight	1	1	2	2
Value	3	4	5	1

Suppose $W = 5$ units of weight in our knapsack.

Seek a load that maximize the value

Problem formulation

Given

- n integer weights $w_1, \ldots, w_n$,
- n values $v_1, \ldots, v_n$, and
- an integer capacity W,

assign either 0 or 1 to each of $x_1, \ldots, x_n$ so that the sum

$$f(x) = \sum_{i=1}^{n} x_i v_i$$

is maximized, s.t.

$$\sum_{i=1}^{n} x_i w_i \leq W.$$

Explanation

$x_i = 1$ represents putting Object i into the knapsack and $x_i = 0$

represents leaving Object i out of the knapsack.

The value of the chosen load is $\sum_{i=1}^{n} x_i v_i$. We want the most valuable load, so we want to maximize this sum.

The weight of the chosen load is $\sum_{i=1}^{n} x_i w_i$. We can't carry more than W units of weight, so this sum must be $\leq W$.

Solving the 0-1 Knapsack

0-1 knapsack is an optimization problem.

Should we apply dynamic programming to solve it ? To answer this question we need to investigate two things :

1. Whether subproblems are solved repeatedly when using a recursive algorithm.
2. An optimal solution contains optimal sub-solutions, the problem exhibits optimal substructure

Optimal Substructure

Does integer 0-1 knapsack exhibits the optimal substructure property ?

Let $\{x_1, x_2, \ldots, x_k\}$ be the objects in an optimal solution x.

The optimal value is $V = v_{x_1} + v_{x_2} + \cdots + v_{x_k}$.

We must also have that $w_{x_1} + w_{x_2} + \cdots + w_{x_k} \leq W$ since x is a feasible solution.

Claim :

If $\{x_1, x_2, \ldots, x_k\}$ is an optimal solution to the knapsack problem with weight W, then $\{x_1, x_2, \ldots, x_{k-1}\}$ is an optimal solution to the knapsack problem with $W' = W - w_{x_k}$.

Optimal Substructure

Proof : Assume $\{x_1, x_2, \ldots, x_{k-1}\}$ is not an optimal solution to the subproblem. Then there are objects $\{y_1, y_2, \ldots, y_l\}$ such that

$$w_{y_1} + w_{y_2} + \cdots + w_{y_l} \leq W',$$

and

$$v_{y_1} + v_{y_2} + \cdots + v_{y_l} > v_{x_1} + v_{x_2} + \cdots + v_{x_{k-1}}.$$

Then

$$v_{y_1} + v_{y_2} + \cdots + v_{y_l} + v_{x_k} > v_{x_1} + v_{x_2} + \cdots + v_{x_{k-1}} + v_{x_k}.$$

However, this implies that the set $\{x_1, x_2, \ldots, x_k\}$ is not an optimal solution to the knapsack problem with weight W.

This contradicts our assumption. Thus $\{x_1, x_2, \ldots, x_{k-1}\}$ is an optimal solution to the knapsack problem with $W' = W - w_{x_k}$.

Behavior of recursive solutions

Define $K[i, j]$ to be the maximal value for the 0-1-knapsack involving the first i objects for a knapsack of capacity j.

Then we have

$$K[1, j] = \begin{cases} v_1 & \text{if } w_1 \leq j \\ 0 & \text{if } w_1 > j. \end{cases}$$

To compute $K[i, j]$, notice that

- if we add the ith element to the knapsack, the sack had weight $j - w_i$ before it was added, and
- if we don't add the ith element, then $K[i, j] = K[i - 1, j]$.

Behavior of recursive solutions

To compute $K[i,j]$, notice that

- if we add the ith element to the knapsack, the sack had weight $j - w_i$ before it was added, and
- if we don't add the ith element, then $K[i,j] = K[i-1,j]$.

Thus,

$$K[i,j] =$$

$$\begin{cases} K[i-1,j] & \text{if } w_i > j \\ max\{K[i-1,j], K[i-1,j-w_i] + v_i\} & \text{if } w_i \leq j. \end{cases}$$

The maximum value is $K[n,W]$.

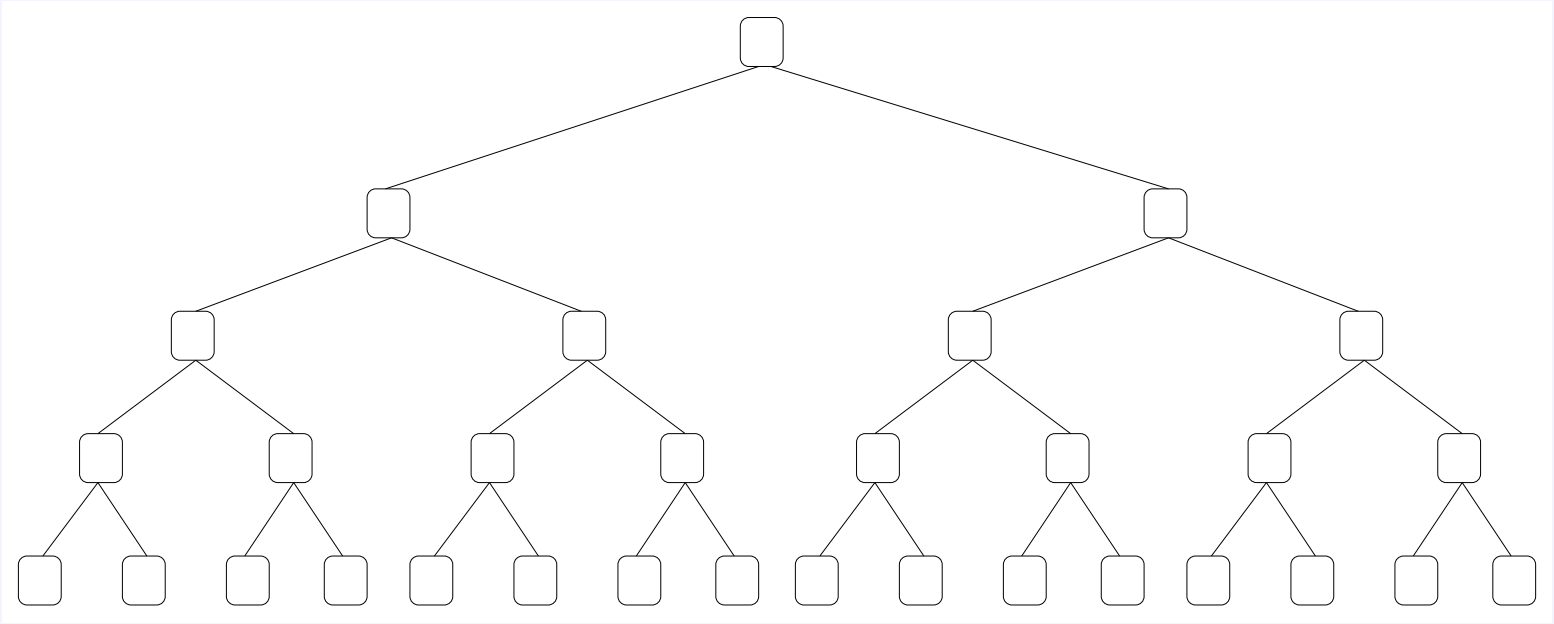
Divide & Conquer 0-1 Knapsack

int K(i, W)
  if ($i = 1$) **return** ($W < w[1]$) ? 0 : $v[1]$;
  if ($W < w[i]$) **return** K($i - 1, W$) ;
  return max(K($i - 1, W$), K($i - 1, W - w[i]$) + $v[i]$) ;

Solve for the following problem instance where $W = 10$:

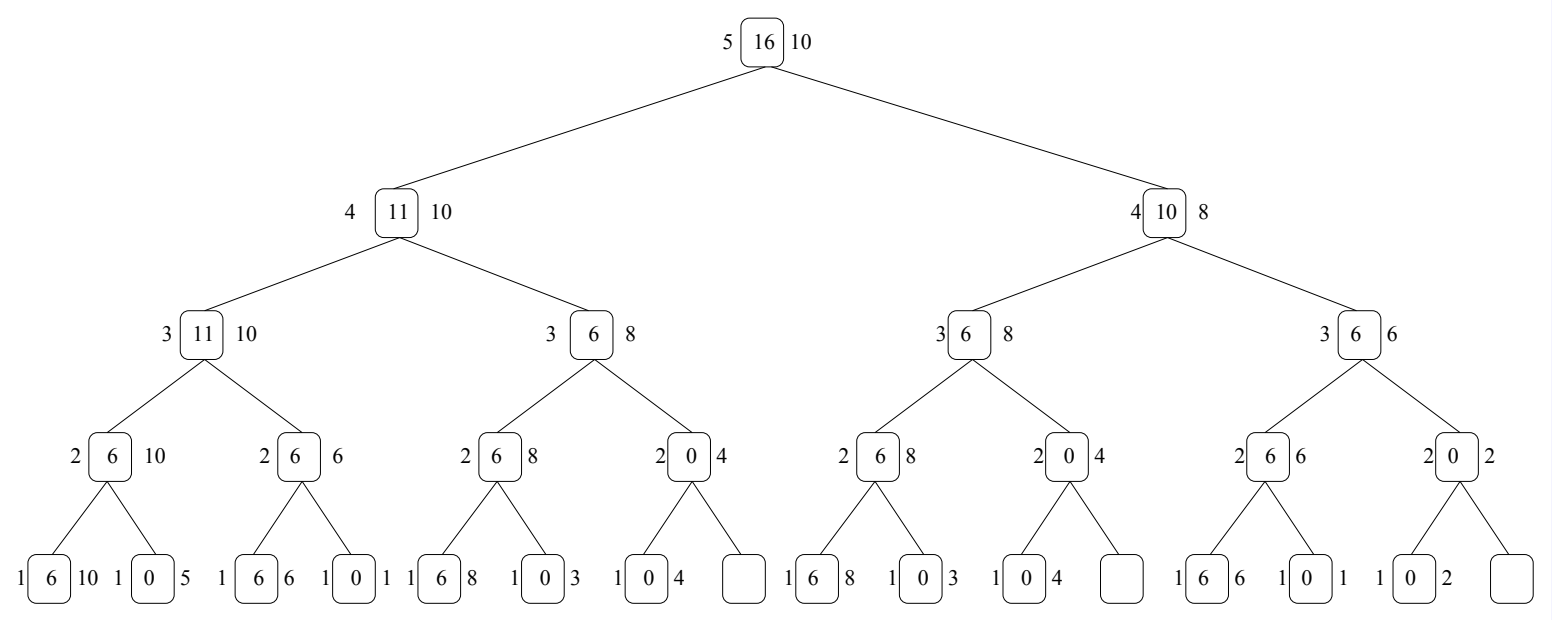
i	1	2	3	4	5
w_i	6	5	4	2	2
v_i	6	3	5	4	6

Optimal substructure



i	1	2	3	4	5
w_i	6	5	4	2	2
v_i	6	3	5	4	6

int K(i, W)
  if $(i = 1)$ **return** $(W < w[1])$? 0 : $v[1]$;
  if $(W < w[i])$ **return** K$(i - 1, W)$;
  return max(K$(i - 1, W)$, K$(i - 1, W - w[i]) + v[i]$);



Analysis of the Recursive Solution

Let $T(n)$ be the worst-case running time on an input with n objects.

If there is only one object, we do a constant amount of work.

$$T(1) = 1.$$

If there is more than one object, this algorithm does a constant amount work plus two recursive calls involving $n - 1$ objects.

$$T(n) = \begin{cases} 1 & \texttt{if } n = 1 \\ 2T(n-1) + c = 1 & \texttt{if } n > 1 \end{cases}$$

Solving the recurrence

$$T(n) = \begin{cases} 1 & \texttt{if } n = 1 \\ 2T(n-1) + c = 1 & \texttt{if } n > 1 \end{cases}$$

we first observe that we cannot apply the Master Theorem because $b \leq 1$. So we use the substitution method here.

Solving the recurrence

Educated guess, $T(n) \in \Theta(2^n)$:

$$
\begin{aligned}
T(n) &= 2T(n-1)+1 \\
&= 2[2T(n-2)+1]+1 \\
&= 2^2T(n-2)+2+1 \\
&= 2^2[2T(n-3)+1]+2+1 \\
&= 2^3T(n-3)+2^2+2+1 \\
&= \cdots \\
&= 2^{n-1}T(n-(n-1))+2^{n-1}-1 \\
&= 2^{n-1}T(1)+2^{n-1}-1 \\
&= 2^{n-1}+2^{n-1}-1 \\
&= 2^n-1
\end{aligned}
$$

Solving the recurrence

Basic Step (n = 1) :

$$\begin{aligned} T(1) &= 1 \text{ by definition} \\ &= 2^1 - 1 = 1 \end{aligned}$$

Solving the recurrence

Inductive Step :
Assume the closed formula holds for n-1, that is, $T(n-1) = 2^{n-1} - 1$, for all $n \geq 2$. Show the formula also holds for n, that is, $T(n) = 2^n - 1$.

$$\begin{aligned} T(n) &= 2T(n-1) + 1 \text{ by recursive definition} \\ &= 2(2^{n-1} - 1) + 1 \text{ by inductive hypothesis} \\ &= 2(2^{n-1}) - 2 + 1 \\ &= 2(2^{n-1}) - 1 \\ &= 2^n - 1 \end{aligned}$$

Therefore $T(n) \in \Theta(2^n)$

Overlapping Subproblems

We have seen that the maximal value is $K[n, W]$.

But computing $K[n, W]$ recursively cost $2^n - 1$.

But the number of subproblems is nW.

Thus, if $nW < 2^n$, then the 0-1 knapsack problem will certainly have overlapping subproblems, therefore using dynamic programming is most likely to provide a more efficient algorithm.

0-1 knapsack satisfies the two pre-conditions (optimal substructure and repeated solutions of identical subproblems) justifying the design of an DP algorithm for this problem.

0-1 Knapsack : DP algorithm

Declare a table K of size $n \times W + 1$ that stores the optimal solutions of all the possible subproblems. Let $n = 6$, $W = 10$ and

i	1	2	3	4	5	6
w_i	3	2	6	1	7	4
v_i	7	10	2	3	2	6

$i \backslash j$	0	1	2	3	4	5	6	7	8	9	10
1											
2											
3											
4											
5											
6											

0-1 Knapsack : DP algorithm

Initialization of the table :

The value of the knapsack is 0 when the capacity is 0. Therefore, $K[i, 0] = 0$, $i = 1..10$.

$i \backslash j$	0	1	2	3	4	5	6	7	8	9	10
1	0			✓	✓	✓	✓	✓	✓	✓	✓
2	0										
3	0										
4	0										
5	0										
6	0										

0-1 Knapsack : DP algorithm

Initialization of the table using the base case of the recursive function : if (i = 1) return (W < w[1]) ? 0 : v[1]

This said that if the capacity is smaller than the weight of object 1, then the value is 0 (cannot add object 1), otherwise the value is $v[1]$

Since $w[1] = 3$ we have :

$i \backslash j$	0	1	2	3	4	5	6	7	8	9	10
1	0	0	0	7	7	7	7	7	7	7	7
2	0										
3	0										
4	0										
5	0										
6	0										

0-1 Knapsack : DP algorithm

The DP code for computing the other entries of the table is based on the recursive function for 0-1 knapsack :

int K(i, W)
  if ($i = 1$) **return** ($W < w[1]$) ? 0 : $v[1]$;
  if ($W < w[i]$) **return** K($i - 1, W$) ;
  return max(K($i - 1, W$), K($i - 1, W - w[i]$) + $v[i]$) ;

$i \backslash j$	0	1	2	3	4	5	6	7	8	9	10
1	0	0	0	7	7	7	7	7	7	7	7
2	0										
3	0										
4	0										
5	0										
6	0										

0-1 Knapsack : DP algorithm

The dynamic programming algorithm is now (more or less) straightforward.

```
function 0-1-Knapsack(w, v, n, W)
  int K[n, W + 1];
  for(i = 1; i ≤ n; i++)   K[i, 0] = 0;
  for(j = 0; j ≤ W; j++)
     if (w[1] ≤ j) then K[1, j] = v[1];
     else K[1, j] = 0;
  for (i = 2; i ≤ n; i++)
      for (j = 1; j ≤ W; j++)
          if (j ≥ w[i] && K[i − 1, j − w[i]] + v[i] > K[i − 1, j])
             K[i, j] = K[i − 1, j − w[i]] + v[i];
          else
             K[i, j] = K[i − 1, j];
  return K[n, W];
```

0-1 Knapsack Example

i	1	2	3	4	5	6
w_i	3	2	6	1	7	4
v_i	7	10	2	3	2	6

for $(i = 2; i \leq n; i++)$
  for $(j = 1; j \leq W; j++)$
    if $(j \geq w[i]$ && $K[i-1, j-w[i]] + v[i] > K[i-1, j])$
      $K[i, j] = K[i-1, j-w[i]] + v[i]$;
    else
      $K[i, j] = K[i-1, j]$;

$i \backslash j$	0	1	2	3	4	5	6	7	8	9	10
1	0	0	0	7	7	7	7	7	7	7	7
2	0	0	10	10	10	17	17	17	17	17	17
3	0	0	10	10	10	17	17	17	17	17	17
4	0	3	10	13	13	17	20	20	20	20	20
5	0	3	10	13	13	17	20	20	20	20	20
6	0	3	10	13	13	17	20	20	20	23	26

Finding the Knapsack

How do we compute an optimal knapsack ?

With this problem, we don't have to keep track of anything extra. Let $K[n, k]$ be the maximal value.

If $K[n, k] \neq K[n-1, k]$, then $K[n, k] = K[n-1, k-w_n] + v_n$, and the nth item is in the knapsack.

Otherwise, we know $K[n, k] = K[n-1, k]$, and we assume that the nth item is not in the optimal knapsack.

Finding the Knapsack

In either case, we have an optimal solution to a subproblem.

Thus, we continue the process with either $K[n-1, k]$ or $K[n-1, k-w_n]$, depending on whether n was in the knapsack or not.

When we get to the $K[1, k]$ entry, we take item 1 if $K[1, k] \neq 0$ (equivalently, when $k \geq w[1]$)

Finishing the Example

- Recall we had :

i	1	2	3	4	5	6
w_i	3	2	6	1	7	4
v_i	7	10	2	3	2	6

- We work backwards through the table

$i \backslash j$	0	1	2	3	4	5	6	7	8	9	10
1	0	0	0	7	7	7	7	7	7	7	7
2	0	0	10	10	10	17	17	17	17	17	17
3	0	0	10	10	10	17	17	17	17	17	17
4	0	3	10	13	13	17	20	20	20	20	20
5	0	3	10	13	13	17	20	20	20	20	20
6	0	3	10	13	13	17	20	20	20	23	26

- The optimal knapsack contains $\{1, 2, 4, 6\}$

0-1 Knapsack Example

Given the following instance with $W = 10$:

i	1	2	3	4	5
w_i	6	5	4	2	2
v_i	6	3	5	4	6

$i \backslash j$	0	1	2	3	4	5	6	7	8	9	10
1	0										
2	0										
3	0										
4	0										
5	0										

What is the optimal value is ? Which items should we take ?