Data Structures and Algorithms
Lecture notes: Algorithm paradigms: Dynamic programming, Part 1

Lecturer: Michel Toulouse

Hanoi University of Science and Technology
michel.toulouse@soict.hust.edu.vn

15 mai 2021

Outline

Divide-&-conquer algorithms

Recursive algorithms like D&C have two phases :

1. A "divide" phase where a problem instance is decomposed into smaller instances
2. A "conquer" phase where solutions to sub-problems are composed into solutions of larger problems

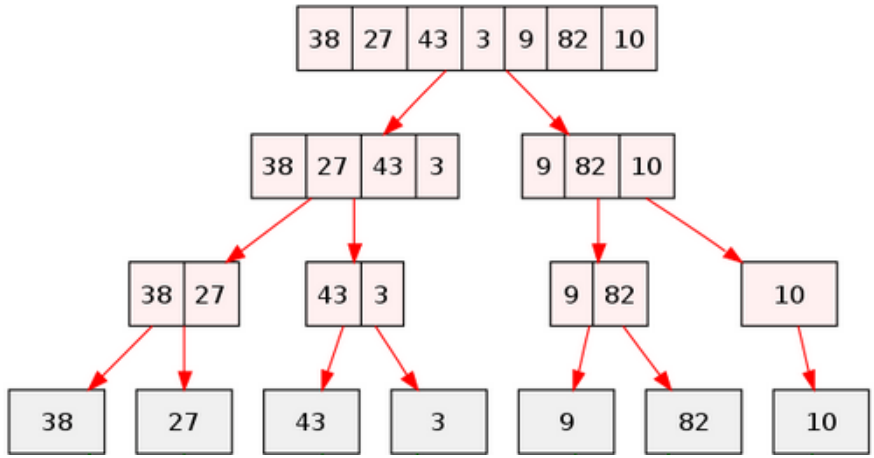
Merge sort : recursive decomposition

During the divide phase of a divide-and-conquer algorithm like merge sort, problem instances are recursively decomposed into subproblems.

For merge sort, a problem instance is array of integers to be sorted, which is then recursively decomposed into subarrays.

Each subarray is a different problem instance, there are 14 problem instances, 14 arrays to be sorted, in the example below.

Mergesort($A[p..r]$)
  if $p < r$
    $q = \lfloor \frac{p+r}{2} \rfloor$
    Mergesort($A[p..q]$)
    Mergesort($A[q+1..r]$)
    Merge(A, p, q, r)



Merge sort : the conquer phase

The conquer phase is executed by the Merge() function

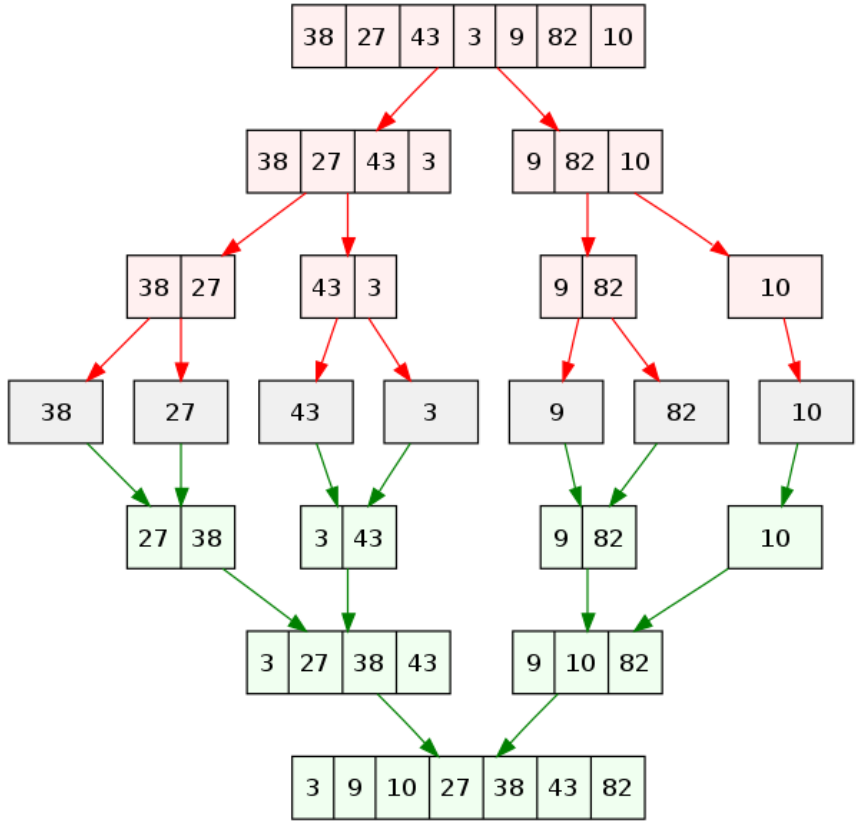
All the 14 problem instances are sorted

First the arrays with a single integer which are inherently sorted

Then the arrays of size 1 are merged together in sorted arrays of size 2

Arrays of size 2 are merged into sorted arrays of size 4, etc

Mergesort($A[p..r]$)
  if $p < r$
    $q = \lfloor \frac{p+r}{2} \rfloor$
    Mergesort($A[p..q]$)
    Mergesort($A[q+1..r]$)
    Merge(A, p, q, r)



Problems with divide-and-conquer

During the divide part of divide-and-conquer some subproblems could could be generated more than one time.

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

Example : D&C algo for computing the Fibonacci numbers

It is a sequence of integers. After the first two numbers in the sequence, each other number is the sum of the two previous numbers in the sequence :

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89$$

The following divide-and-conquer algorithm computes the first $n + 1$ values of the Fibonacci sequence :

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

Computing Fibonacci numbers

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

First the following divide steps take place :

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

Computing Fibonacci numbers

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

Next solutions to sub-problems are combined to obtain the solution of the original problem instance :

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

Time complexity of DC Fibonacci

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

One possible recurrence relation for this algorithm that counts the number of time the function *Fib* is called is the following :

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

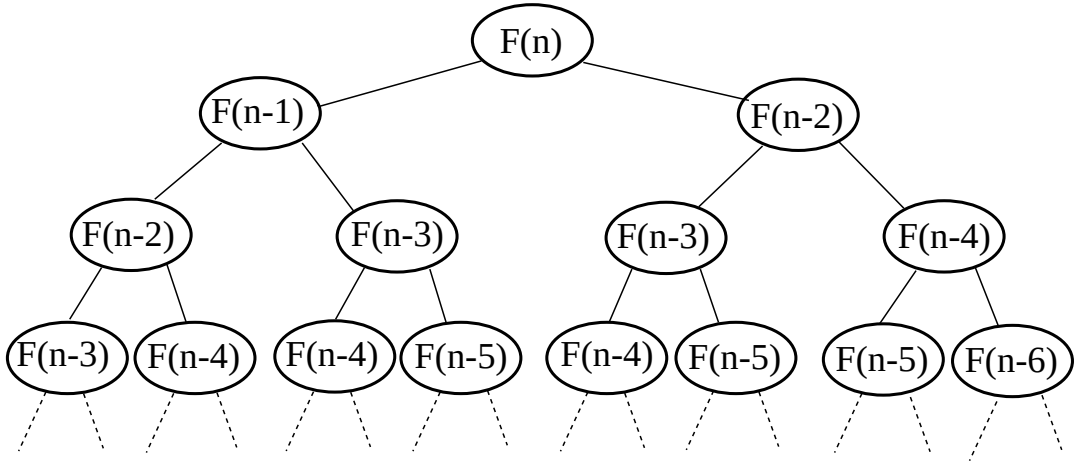
The solution of this recurrence is $O((\frac{1+\sqrt{5}}{2})^n)$, the time complexity is exponential.

Call tree of D&C Fibonacci

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

The poor time complexity of *Fib* derived from re-solving the same sub-problems several times.

D&C generates the following call tree :



Dynamic Programming

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.

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

Save the solution to a subproblem in a table (an array), and refer back to the table whenever we revisit the subproblem

Dynamic programming main ideas

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

Designing dynamic programming algorithms

There are two dynamic programming algorithm design options :

- Top down : start to compute with the whole problem
- Bottom up : start to compute with the D&C base case

Top-down dynamic programming for Fibonacci

A top-down design is a recursive algorithm where the original problem is decomposed into smaller problems and once the base cases are solved, solutions to subproblems are computed out of results in the table :

```
function DyFib(n)
  if (n == 0) return 0;
  if (n == 1) return 1;
  if (table[n] != 0) return table[n];
  else
    table[n] = DyFib(n − 1) + DyFib(n − 2)
    return table[n];
```

input	0	1	2	3	4	5	6	7	8	9	10	11	12	13
solution	0	1												

Bottom up dynamic programming for Fibonacci

Bottom up design is an iterative algorithm which first compute the base cases and then uses the solutions to the base cases to start computing the solutions to the other larger subproblems :

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

input	0	1	2	3	4	5	6	7	8	9	10	11	12	13
solution	0	1												

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

When do we need DP

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

Writing a DP often starts by writing a D&C recursive algorithm and use the conquer phase of the D&C recursive code to write an iterative bottom up DP algorithm :

- Create a table that will store the solution of the subproblems
- Use the "base case" of recursive 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.

Here is a recursive D&C algorithm for returning 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) ;

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

Making change : a recursive solution

```
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);
```

How did we get this D&C algo ? The making change problem has 2 input parameters :

1. n the number of different denominations
2. N the amount of change to return

Sub-problems are obtained by reducing the value of one of the two inputs :

1. Try to solve a sub-problem using $n - 1$ denominations, i.e. the sub-problem make_change$(i - 1, j)$
2. Use denomination n to reduce the amount of money to return, thus we are left with the sub-problem make_change$(i, j - d_i)$ to solve.

The base case is when the amount to return is 0, in which case no coin is used, the solution to the base case is 0.

C code implementing recursive make_change

```
#include <stdio.h>
#define min(a,b)((a<b)? a:b)
int make_change(int d[], int n, int N)
{
  if(N == 0) return 0;
  else if (N < 0 || (N > 0 && n <= 0)) return 1000;
  else{
    return min(make_change(d,n-1,N), make_change(d,n,N-d[n-1]) + 1);
  }
}
int main()
{
    int d[] = {1, 5, 10, 25};
    int N = 13;
    int n = sizeof(d)/sizeof(d[0]);
    int ans = make_change(d, n, N);
    printf("Minimal # of coins = %d\n",ans);
    return 0;
}
```

Designing DP based on the D&C algo

The D&C algo has two parameters because a make change problem has two dimensions : number of different types of coins available and the amount to return.

An instance of make change problem can be uniquely identified by these two values.

DP uses a table to store the values of instances that have already been computed, it makes sense here to use a 2-dimensional table where one dimension refers to the coin types and the other to the amounts to return

Coins/Amounts	0	1	2	3	4	5	6	7	8
$d_1 = 1$									
$d_2 = 4$									
$d_3 = 6$									

The entries of the above table can store the values of all the sub-problems of a make change problem instance with 3 types of coins (1,4,6) and an amount to return equal to 8.

For example, entry $(d_1 = 1, 2)$ refers to a sub-problem where the amount to return is 2 and where only coins of type $d_1 = 1$ are available to return this amount.

Designing DP based on the D&C algo

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.

Coins/Amounts	0	1	2	3	4	5	6	7	8
$d_1 = 1$									
$d_2 = 4$									
$d_3 = 6$									

Entry $t[i, j]$ will store the solution to sub-problem instance i, j, i.e. the minimum number of coins needed to refund an amount of j units using only coins from denominations 1 to i.

Designing DP based on the D&C algo

Using the base case of the D&C algo :

```
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);
```

one can immediately fill the 3 entries of the table where $j = 0$, i.e. for $i = 1, 2, 3$

Coins/Amounts	0	1	2	3	4	5	6	7	8
$d_1 = 1$	0								
$d_2 = 4$	0								
$d_3 = 6$	0								

Designing DP based on the D&C algo

The solution to make change for instances where $i = 0$ is not defined. Thus the DP for computing the entries of instances where $i = 1$ can only be based on the second recursive call of the D&C algo :

```
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);
```

Thus the solution to instance $t[1,1] = \text{make\_change}(i, j - d_i) + 1 = \text{make\_change}(1, 1-1) + 1 = \text{make\_change}(1,0) + 1 = t[1,0] + 1 = 1$

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

Coins/Amounts	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								

Example : the value stored in entry $t[1,4]$ is interpreted as the minimum number of coins to return 4 units using only denomination 1, which is the minimum number of coins to return 3 units, i.e. $t[1,3] + 1 = 4$ coins.

Designing DP based on the D&C algo

For the solutions of instances where $i > 1$, one needs to consider both recursive calls of the D&C algo. and take the minimum value return from these two.

```
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);
```

Thus the solution to instance $t[i, j]$ is the minimum of the values returned by make_change$(i - 1, j)$ and make_change$(i, j - d_i) + 1$, i.e. $t[i, j] = \min(t[i - 1, j], t[i, j - d_i])$.

Special case : if the amount of change to return is smaller than domination d_i, i.e. $j < d_i$, then the change needs to be returned can only be based on denominations smaller than d_i, i.e $t[i, j] = t[i - 1, j]$

Coins/Amounts	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			

A bottom-up DP algorithm for making change

For the general, the following table look-ups deduced from the D&C algo is used to compute entries in the DP table :

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

DP_making_change(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

Coins/Amounts	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 returned, we have

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

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

Using the DP table to return coins

Entry $t[n, N]$ display the minimum number of coins that can used to return change for N.

Coins/Amounts	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

From other entries in the DP table we can also find what is the denomination of the coins to return :

- 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 $t[i, j - d_i]$.

Exercises 1 and 2

1. Construct the table and solve the making change problem where $n = 3$ with denominations $d_1 = 1, d_2 = 2$ and $d_3 = 3$ where the amount of change to be returned is $N = 7$

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

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

2. Construct the table and solve the making change problem where $n = 4$ with denominations $d_1 = 1, d_2 = 3, d_3 = 4$ and $d_4 = 5$ where the amount of change to be returned is $N = 12$

Solutions table

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 in the above table is a solution to a subproblem.

For example, entry $t[1, 3]$ is the solution to the subproblem where an amount of 3 needs to be returned using only coins of denomination 1.

Amount	0	1	2	3
$d_1 = 1$	0	1	2	3

Entry $t[2, 6]$ is the number of coins that have to be returned when an amount of 6 needs to be returned using only coins of denomination 1 or 4.

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

Optimization problems

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 problem consists to minimize the function $f(x)$, i.e. to minimize the number of coins returned

There is only one constraint : the sum of the value of the coins is equal to the amount to be returned

Solutions of an optimization problems

Optimization problems often have many solutions

The problem below has, in the last column, "3 feasible solutions", 6, 3 and 1.

The 3 solutions are feasible because they satisfy the constraint that the sum of the coin values is equal the amount to return.

However, only the solution with the smallest value is a solution to the optimization problem which is to minimize the number of coins

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

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 :

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

Coins/Amounts	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

The optimal solution for $t[i, j]$ is obtained by comparing $t[i-1, j]$ and $t[i, j-d_i]+1$, 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 the solution for n.

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

DP for optimization problems

The basic steps are :

1. Characterize the structure of an optimal solution, i.e. the problem meet the optimal substructure property
2. Give a recursive definition for computing the optimal solution based on optimal solutions of smaller problems.
3. 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 $W' = W - w_{x_k}$.

Recursive solution : problem decomposition

Seeking for a recursive algorithm to a problem one must think about the ways the problem can be reduced to subproblems.

Let $K[n, W]$ denote the 0-1 knapsack problem instance to be solved where n is the initial number of objects to be considered and W is the initial capacity of the sack.

There are two ways this problem can be decomposed into smaller problems. Let $K[i, j]$ be the recursive function :

- one can add the ith object in the knapsack, thus reducing the initial problem to one with $i - 1$ objects yet to consider and capacity $j - w_i$: $K[i - 1, j - w_i]$
- one can choose to disregard object i (don't put in the sack), thus generating a new subproblem with $i - 1$ objects and capacity j unchanged : $K[i - 1, j]$

Recursive solution : writing the algorithm

The base case will be when one object is left to consider. The solution is

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

Once the value of the base case is computed, the solution to the other subproblems is obtained as followed :

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

This recursive function is initially called with $K[n,W]$.

Divide & Conquer 0-1 Knapsack

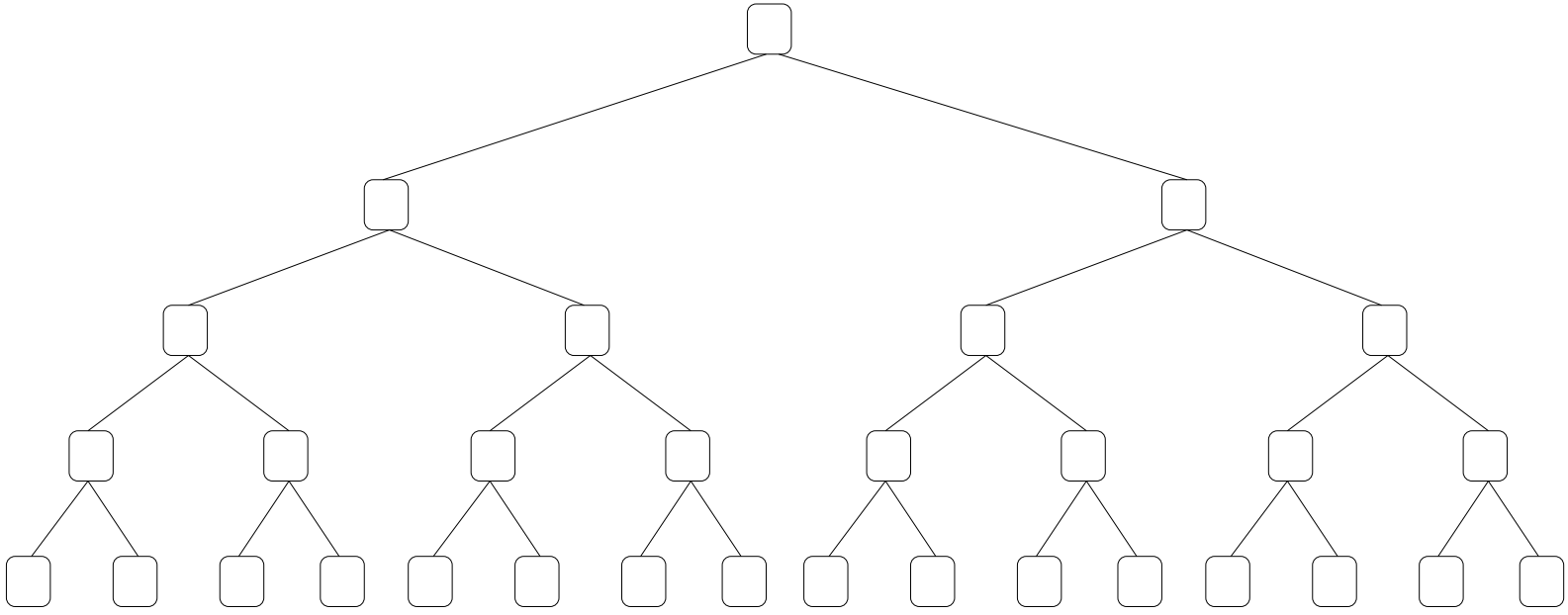
Below is the pseudo-code of a D&C algorithm that computes the optimal solution of a knapsack problem instance.

This algorithm has 3 inputs : the capacity W, an array v of n values and an array w of n weights

```
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]);
```

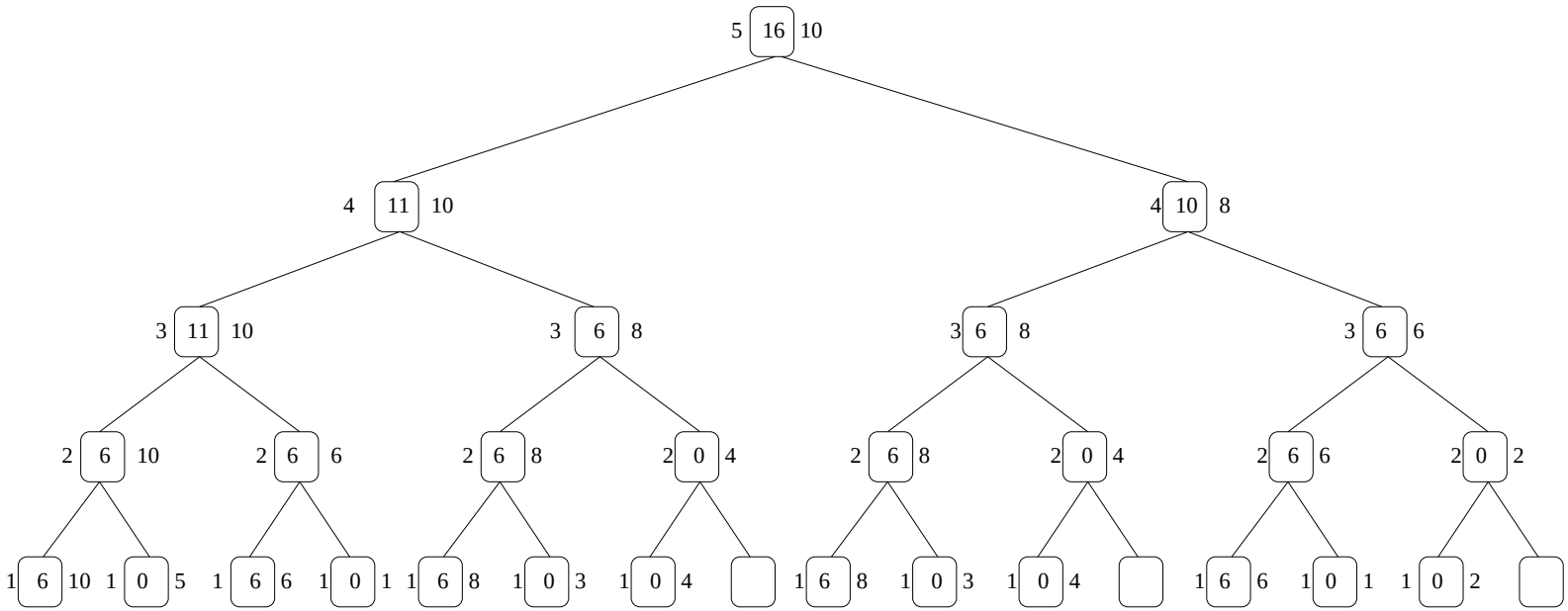
Solve for the following problem instance where $W = 10$:

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



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]);
```



C code implementing recursive 0-1 knapsack

The initial call to $K(n-1, W)$ because array indexes in C start at 0, so values of object 1 are in val[0] and wt[0], etc.

```
#include <stdio.h>
int max(int a, int b) { return (a > b) ? a : b; }
int K(int W, int wt[], int val[], int n) {
   // Base Case
   if (n ==0) return (W < wt[0])? 0 : val[0];
   //Knapsack does not have residual capacity for object n
   if (wt[n] > W) return K(W, wt, val, n - 1);
   else
     return max(
         val[n] + K(W - wt[n], wt, val, n - 1),
         K(W, wt, val, n - 1));
}
int main() {
   int val[] = { 6, 3, 5, 4, 6};  int wt[] = { 6, 5, 4, 2, 2 };
   int W = 10;
   int n = sizeof(val) / sizeof(val[0]);
   printf("The solution is %d\n", K(W, wt, val, n-1));
   return 0;
}
```

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 of work plus two recursive calls involving $n-1$ objects.

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

The solution to this recurrence is $T(n) \in \Theta(2^n)$

Overlapping Subproblems

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

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

While the number of subproblems is only 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 : Bottom up 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 : Bottom up DP algorithm

Initialization of the table :

The value of the knapsack is 0 when the capacity is 0. Therefore, $K[i, 0] = 0$, $i = 1..6$.

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

0-1 Knapsack : Bottom up 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 : Bottom up DP algorithm

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

i	1	2	3	4	5	6
w_i	3	2	6	1	7	4
v_i	7	10	2	3	2	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]);
```

$i \backslash j$	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 : Bottom up DP algorithm

The bottom-up 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];
```

Caution on the running time of the DP algo for knapsack

The previous algorithm runs in $O(nW)$, this seems polynomial in the input size, but this is not the case because W is not polynomial

Two of the inputs of the previous algo are a vector of n weights and a vector of n values. Assume the largest number in these two vectors is 29, thus we need max 5 bits to represent any number in the two vectors. Thus the total number of input bits to represent these two vectors $2 \times 5 \times n$. The outer for loop **for** $(i = 2; i \leq n; i++)$ runs n times which is linear in the size of the two input vectors

The other input is W. Assume $W = 16$, thus we need only 4 bits to represent W. However, the inner loop **for** $(j = 1; j \leq W; j++)$ runs 16 times, i.e. 2^4 times its input size of 4 bits !

For this reason, the running time of DP is said to be "pseudo-polynomial". Knapsack is a NP-hard problem (actually weakly NP-hard), which means it is unlikely to have a polynomial time solution

Nonetheless, the knapsack problem is an intuitive example to introduce dynamic programming, considering the notions of pseudo-polynomial time and NP-hardness will not be on the final...

0-1 Knapsack example solved with DP

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 ≤ 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];
```

$i \backslash j$	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

0-1 Knapsack example solved with D&C

i	1	2	3	4	5	6
w_i	3	2	6	1	7	4
v_i	7	10	2	3	2	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]);
```

$i \backslash j$	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

max(K(i-1,W), K(i-1,W-w[i]) + v[i])

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

Exercise 3 : 0-1 Knapsack

Solve the following 0-1 knapsack instance with $W = 10$:

```
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	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 ? Which objects are part of the optimal solution ?

Exercise 4 : 0-1 Knapsack

Solve the following 0-1 knapsack problem : $W = 10$

i	1	2	3	4	5	6
w_i	4	2	3	1	6	4
v_i	6	4	5	3	9	7