Algorithm Design Strategies IV

Joaquim Madeira

Version 0.2 – October 2019

Overview

- Dynamic Programming
- Fibonacci's Sequence
- Memoization
- Computing Binomial Coefficients
- Computing Delannoy Numbers
- The Coin Row Problem
- The 0-1 Knapsack Problem
- Other Problems

Dynamic Programming

- General algorithm design technique

- Apply to
  - Computing recurrences
  - Solving optimization problems

- How to store "previous" results ?
  - 2D array
  - Vector
  - A few variables

Recurrences – Top-Down

- Exploit the relationship between
  - A solution to a given problem instance
  - Solutions to smaller/simpler instances of the same problem
- Set up a recurrence !
- Decompose into smaller / simpler sub-problems
  - Parameters ?
- Identify the smallest / simplest / trivial problems
  - Base cases

Dynamic Programming – Bottom-up

- Use a recurrence: BUT go bottom-up !
- Start from the smallest / simplest / trivial problems
- Get intermediate solutions from smaller / simpler sub-problems
- Which values / results are computed in each step ?
  - How to store ?

Dynamic Programming – Advantage

- Do sub-problems overlap ?
- NOW, there is no need to repeatedly solve the same sub-problems !!
- Proceed bottom-up and store results for later use
- Compare with Divide-and-Conquer !!

Fibonacci's Sequence

- F(0) = 0 ; F(1) = 1
- F(i) = F(i - 1) + F(i - 2) ; i = 2, 3, 4,…
- F(6) = ? → Number of recursive calls ?
- Do sub-problems overlap ?
- Recursion tree vs. recursion DAG !!
- Complexity order ?

Tasks – V1

- Implement the recursive function of the previous slide in Python
- Count the number of additions carried out for computing a Fibonacci number
  - Use a global variable
- Table ?
- Complexity order ?

Fibonacci's Sequence

```
def fibonacci_DC( n ) :

    """ Recursive computation of Fi """

    # Global variable, for counting the number of additions

    global num_adds

    if ( n == 0 ) or ( n == 1 ) :

        return n

    num_adds += 1

    return fibonacci_DC( n - 1 ) + fibonacci_DC( n - 2 )
```

Number of additions ?

- $A(0) = 0$; $A(1) = 0$
- $A(i) = 1 + A(i - 1) + A(i - 2)$; $i = 2, 3, 4, \ldots$
- Closed formula ?
- You can get it, if you remember Discrete Mathematics…
- BUT, we can get the complexity order from the table…

Fibonacci’s Sequence

- $F(0) = 0$; $F(1) = 1$
- $F(i) = F(i - 1) + F(i - 2)$; $i = 2, 3, 4, \ldots$

- Use Dynamic Programming !!

- Computing F(n) using an array
  - Complexity order ?

- Can we use less memory space ?

Tasks – V2 + V3

- Implement two iterative functions for computing F(i)
  - V2 : using an array
  - V3 : using just 3 variables

- Count the number of additions carried out

- Table ?
- Complexity order ?

Fibonacci's Sequence

i	f(i)	#ADDs-Rec	#ADDs_DP_1	#ADDs_DP_2
0	0	0	0	0
1	1	0	0	0
2	1	1	1	1
3	2	2	2	2
4	3	4	3	3
5	5	7	4	4
6	8	12	5	5
7	13	20	6	6
8	21	33	7	7
9	34	54	8	8
10	55	88	9	9
11	89	143	10	10
12	144	232	11	11
13	233	376	12	12
14	377	609	13	13
15	610	986	14	14

Additions – Recursive version

- How fast does F(n) grow ?
- How fast does A(n) grow ?
- From the table we get:

  A(n) = F(n+1) – 1

- Exponential growth !!
  - Why ?

$$(1+\sqrt{5})/2 = 1{,}618034$$

n	F(n)	Ratio	A(n)	Ratio
0	0		0	
1	1		0	
2	1	1	1	
3	2	2	2	2
4	3	1,5	4	2
5	5	1,666667	7	1,75
6	8	1,6	12	1,714286
7	13	1,625	20	1,666667
8	21	1,615385	33	1,65
9	34	1,619048	54	1,636364
10	55	1,617647	88	1,62963
11	89	1,618182	143	1,625
12	144	1,617978	232	1,622378
13	233	1,618056	376	1,62069
14	377	1,618026	609	1,619681
15	610	1,618037	986	1,619048
16	987	1,618033	1596	1,618661
17	1597	1,618034	2583	1,618421
18	2584	1,618034	4180	1,618273
19	4181	1,618034	6764	1,618182
20	6765	1,618034	10945	1,618125

Memoization

- Turning the results of a function into something to be remembered
- I.e., avoid repeating the calculation of results for previously processed inputs
- Use a table / array to store previously computed results
  - Initialization !
- Time vs. space trade-off

Memoization

- Initialize all table entries to “null”
  - Not yet computed

- Whenever a result is to be computed for a given input
  - Check the corresponding table entry
  - If not “null”, retrieve the result
  - Otherwise, compute by a recursive call(s)
  - And store the result

Fibonacci's Sequence

- Initialization

```
for(i=1, i<n, i++)   f[i] = -1;
```

- Recursive function

```
int fib( int n ) {
        int r;
        if( f[n] != -1 )   return f[n];
        if( n == 1 )   r = 1;
        else if( n == 2 )   r = 1;
        else {
                r = fib( n – 2 );
                r = r + fib( n – 1 );
        }
        f[n] = r;
        return r;
}
```

The Python way

```
# M. Hetland, Python Algorithms, Apress, 2010 - Chapter 8

from functools import wraps

def memo( func ) :

    cache = {}                              # Stored subproblem solutions

    @wraps(func)                            # Make wrap look like func

    def wrap( *args ) :                     # The memoized wrapper

        if args not in cache :              # Not already computed?

            cache[args] = func( *args )     # Compute & cache the solution

        return cache[args]                  # Return the cached solution

    return wrap                             # Return the wrapper
```

The Python way

i	f(i)	#ADDs_Memo
0	0	0
1	1	0
2	1	1
3	2	1
4	3	1
5	5	1
6	8	1
7	13	1
8	21	1
9	34	1
10	55	1

```
# Testing the memoized version

fibonacci_DC = memo( fibonacci_DC )
```

85	259695496911122585	1
86	420196140727489673	1
87	679891637638612258	1
88	1100087778366101931	1
89	1779979416004714189	1
90	2880067194370816120	1

Another example

- Linear robot
- Can move forward by 1 meter, or 2 meters, or 3 meters

- In how many ways can it move a distance of n meters ?

- Establish the recurrence !!
  - Base cases ?

Tasks – V1 + V2 + V3

- Implement three functions for computing R(i)
  - V1 : using recursion
  - V2 : using an array
  - V3 : using a few variables – how many ?
- Count the number of additions carried out
  - Formulas ?
- Tables ?
- Complexity order ?

Example – Results table

i	r(i)	#ADDs-Rec	#ADDs_DP_1	#ADDs_DP_2
1	1	0	0	0
2	2	0	0	0
3	4	0	0	0
4	7	2	2	2
5	13	4	4	4
6	24	8	6	6
7	44	16	8	8
8	81	30	10	10
9	149	56	12	12
10	274	104	14	14
11	504	192	16	16
12	927	354	18	18
13	1705	652	20	20
14	3136	1200	22	22
15	5768	2208	24	24

Computing Binomial Coefficients

- C(n,0) = 1 ; C(n,n) = 1
- C(n,j) = C(n -1,j) + C(n -1,j -1) ; j = 1, 2,…, n -1

- Two arguments !!
- C(4,3) = ? ➔ Number of recursive calls ?

- Do sub-problems overlap ?
- Recursion tree vs. recursion DAG !!

- Complexity order ?

Computing Binomial Coefficients

- V1 : Compute C(n,j) recursively
- V2 : Compute C(n,j) using a 2D array
  - How to proceed ?
  - Have you seen this “triangle” before ?
- Can we use less memory space ?
- And other, more efficient recurrences ?

Tasks – V1 + V2 + V3

- Implement three functions for computing C(n,j)
  - V1 : using recursion
  - V2 : using a 2D array
  - V3 : using a 1D array
- Count the number of additions carried out
- Tables ?
- Complexity order ?

Pascal's Triangle

```
Pascal's Triangle - Recursive Function

1
1       1
1       2       1
1       3       3       1
1       4       6       4       1
1       5       10      10      5       1
1       6       15      20      15      6       1
1       7       21      35      35      21      7       1
1       8       28      56      70      56      28      8       1
1       9       36      84      126     126     84      36      9       1
1       10      45      120     210     252     210     120     45      10      1
```

V1 – Number of additions

```
Number of Additions - Recursive Function

0
0       0
0       1       0
0       2       2       0
0       3       5       3       0
0       4       9       9       4       0
0       5       14      19      14      5       0
0       6       20      34      34      20      6       0
0       7       27      55      69      55      27      7       0
0       8       35      83      125     125     83      35      8       0
0       9       44      119     209     251     209     119     44      9       0
```

V2 – Number of additions

```
Number of Additions - Dynamic Programming - V. 1

0
0       0
1       1       1
3       3       3       3
6       6       6       6       6
10      10      10      10      10      10
15      15      15      15      15      15      15
21      21      21      21      21      21      21      21
28      28      28      28      28      28      28      28      28
36      36      36      36      36      36      36      36      36      36
45      45      45      45      45      45      45      45      45      45      45
```

V3 – Number of additions

```
Number of Additions - Dynamic Programming - V. 2

0
0       0
0       1       0
0       3       3       0
0       6       6       6       0
0       10      10      10      10      0
0       15      15      15      15      15      0
0       21      21      21      21      21      21      0
0       28      28      28      28      28      28      28      0
0       36      36      36      36      36      36      36      36      0
0       45      45      45      45      45      45      45      45      45      0
```

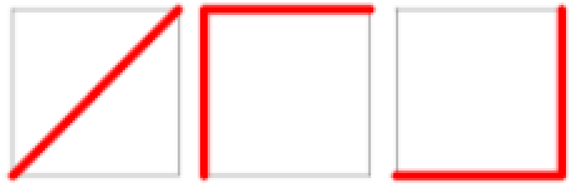
Delannoy Numbers – D(i,j)

- Rectangular grid of size (m,n)
- Start at SW corner : (0,0)
- Steps allowed in N, E or NE directions

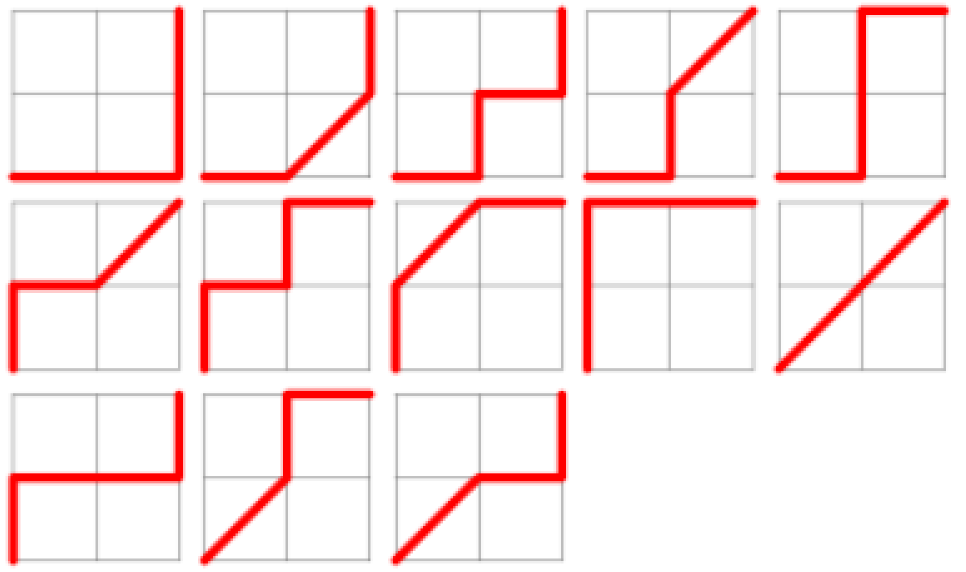
- D(i,j) = number of different paths from (0,0) to (i,j)
  - Recursive definition ?
  - Trivial cases ?

D(n,n) – Central Delannoy Numbers

- D(1,1)



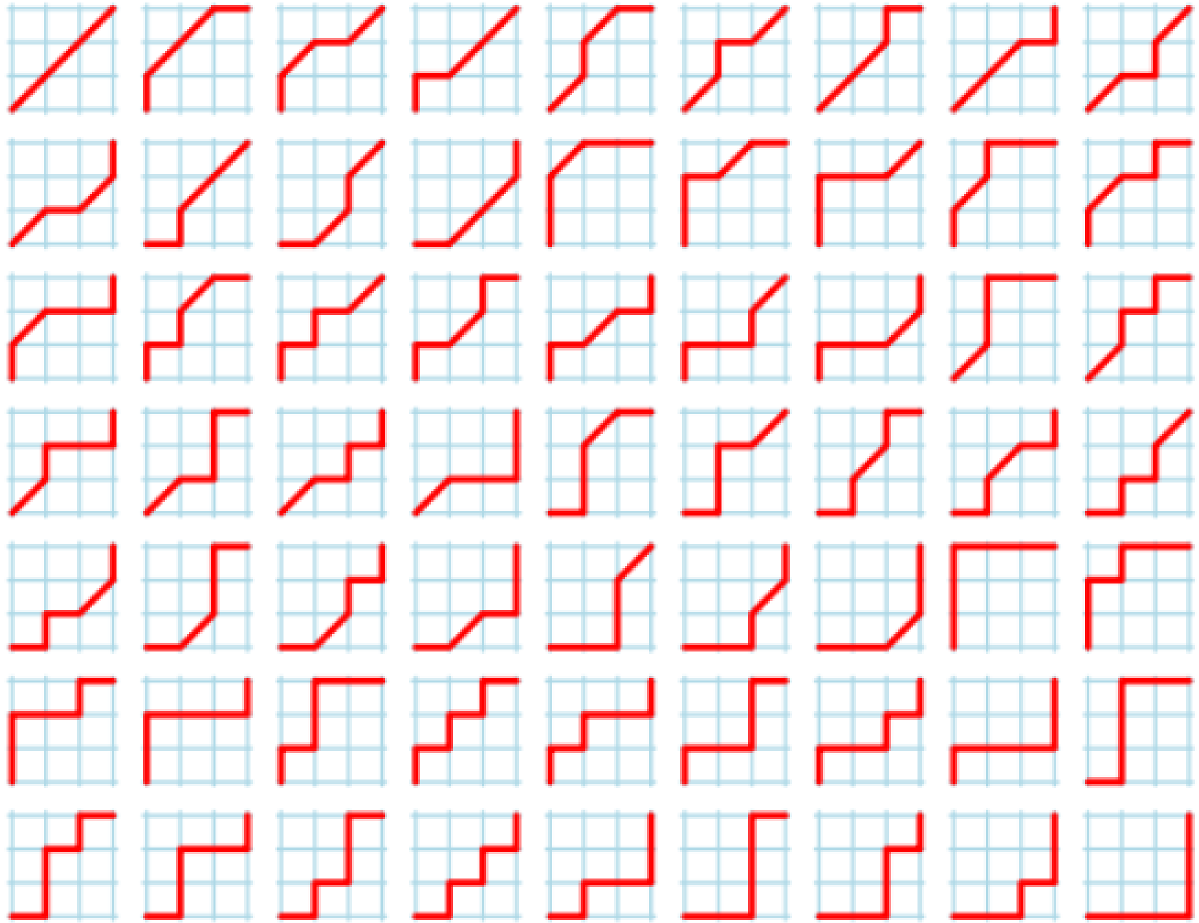
- D(2,2)



[Mathworld]

D(n,n) – Central Delannoy Numbers

- D(3,3)



[Wikipedia]

Delannoy Numbers

$D(m,n) = 1$, if $m = 0$ or $n = 0$

$D(m,n) = D(m-1, n) + D(m-1, n-1) + D(m, n-1)$

- $D(1,1) = ?$
- $D(2,2) = ?$
- $D(2,3) = ?$
- $D(3,2) = ?$

- Arrange the calculations in a triangular representation !
  - Have you seen a similar triangle before ?

Tasks – V1 + V2 + V3

- Implement three functions for computing D(i,j)
  - V1 : using recursion
  - V2 : using a 2D array
  - V3 : using two 1D arrays

- Count the number of additions carried out

- Tables ?
- How fast does D(n,n) grow ?

Delannoy Numbers

Delannoy's Matrix - Recursive Function

1	1	1	1	1	1	1	1	1	1	1
1	3	5	7	9	11	13	15	17	19	21
1	5	13	25	41	61	85	113	145	181	221
1	7	25	63	129	231	377	575	833	1159	1561
1	9	41	129	321	681	1289	2241	3649	5641	8361
1	11	61	231	681	1683	3653	7183	13073	22363	36365
1	13	85	377	1289	3653	8989	19825	40081	75517	134245
1	15	113	575	2241	7183	19825	48639	108545	224143	433905
1	17	145	833	3649	13073	40081	108545	265729	598417	1256465
1	19	181	1159	5641	22363	75517	224143	598417	1462563	3317445
1	21	221	1561	8361	36365	134245	433905	1256465	3317445	8097453

Computing Bernstein Polynomials

$B_{0,0}(t) = 1$

$B_{n,0}(t) = (1 - t)\, B_{n-1,0}(t)$; t in [0,1]

$B_{n,n}(t) = t\, B_{n-1,n-1}(t)$; t in [0,1]

$B_{n,j}(t) = (1 - t)\, B_{n-1,j}(t) + t\, B_{n-1,j-1}(t)$; j = 1, 2,…, n – 1 ; t in [0,1]

- There are (n + 1) polynomials of degree n
- How to obtain the expression of such a polynomial ?
- Arrange the calculations in a triangular representation !
  - Have you seen that triangle before ?

Computing Bernstein Polynomials

- How to compute the value of a polynomial for a given t* ?
- V1 : Compute $B_{n,j}(t^*)$ recursively
- $B_{3,2}(1/2) = ?$
- Number of recursive calls ?
- Are there overlapping sub-problems ?

Computing Bernstein Polynomials

- V2 : Compute $B_{n,j}(t^*)$ using a 2D array
- $B_{3,2}(1/2) = ?$
- How to ?
- Have you seen a similar procedure before ?
- Can we use less memory space ?

Tasks – V1 + V2 + V3

- Implement three functions for computing $B_{n,j}(t)$
  - V1 : using recursion
  - V2 : using a 2D array
  - V3 : using a 1D array
- Count the number of multiplications carried out
- Tables ?
- Complexity order ?

Bernstein Polynomials for t = 0.5

```
Polynomials' Triangle - Recursive Function - t =  0.5

1.000
0.500   0.500
0.250   0.500   0.250
0.125   0.375   0.375   0.125
0.062   0.250   0.375   0.250   0.062
0.031   0.156   0.312   0.312   0.156   0.031
0.016   0.094   0.234   0.312   0.234   0.094   0.016
0.008   0.055   0.164   0.273   0.273   0.164   0.055   0.008
0.004   0.031   0.109   0.219   0.273   0.219   0.109   0.031   0.004
```

Multiplications count – Recursive

Number of Multiplications - Recursive Function

0								
1	1							
2	4	2						
3	8	8	3					
4	13	18	13	4				
5	19	33	33	19	5			
6	26	54	68	54	26	6		
7	34	82	124	124	82	34	7	
8	43	118	208	250	208	118	43	8

Multiplications count – Dynamic Prog.

```
Number of Multiplications - Dynamic Programming - V. 2

0
2       2
6       6       6
12      12      12      12
20      20      20      20      20
30      30      30      30      30      30
42      42      42      42      42      42      42
56      56      56      56      56      56      56      56
72      72      72      72      72      72      72      72      72
```

Multiplications count – Memoization

Number of Multiplications - Memoized Function

0								
1	1							
1	2	1						
1	2	2	1					
1	2	2	2	1				
1	2	2	2	2	1			
1	2	2	2	2	2	1		
1	2	2	2	2	2	2	1	
1	2	2	2	2	2	2	2	1

Optimization Problems

- Goal
  - Minimize or maximize an objective function
  - Store the solution's components

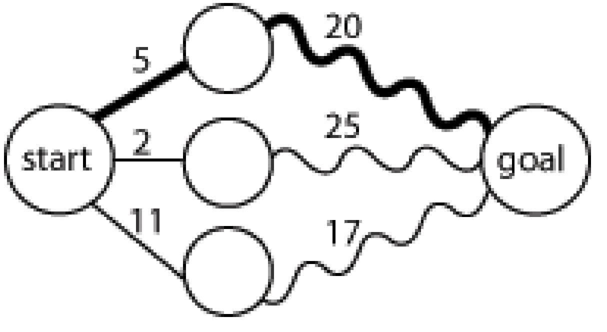
- When can we use dynamic programming ?
  - Overlapping sub-problems
  - Optimal substructure
    - The principle of optimality

The Principle of Optimality

- Does an optimization problem satisfy the principle of optimality ?

- An optimal solution to any of its instances must be made up of optimal solutions to its sub-instances.

- Example
  - Shortest path



[Wikipedia]

The Coin Row Problem

- Row of n coins
- Integer values $c_1, c_2, \ldots, c_n$
  - Not necessarily distinct
- Goal: Pick up the maximum amount of money
- Restriction: No two adjacent coins can be picked up

The Coin Row Problem

- Can we solve it by Exhaustive Search ?
- Or using heuristics ?
- How ?
- Efficiency ?

The Coin Row Problem

- How to derive a recurrence ?
- F(n) = ?
  - Maximum amount that can be picked up from the row of n coins
- n^{th} coin was picked up / not picked up ?
- Trivial cases ?

The Coin Row Problem

- $F(0) = 0$
- $F(1) = c_1$
- $F(n) = \max\{\, c_n + F(n-2),\ F(n-1) \,\}, \quad \text{for } n > 1$
- Example: 5, 1, 2, 10, 6, 2
- $F(6) = ?$

The Coin Row Problem

- The DP algorithm solves the problem for the first i coins in the row, $1 \leq i \leq n$
  - We get the optimal solution for every sub-problem
- How to find the coins of an optimal solution ?
  - Backtrace the computations
  - OR use an additional array to record which term was larger at every step

Tasks – V1 + V2 + V3

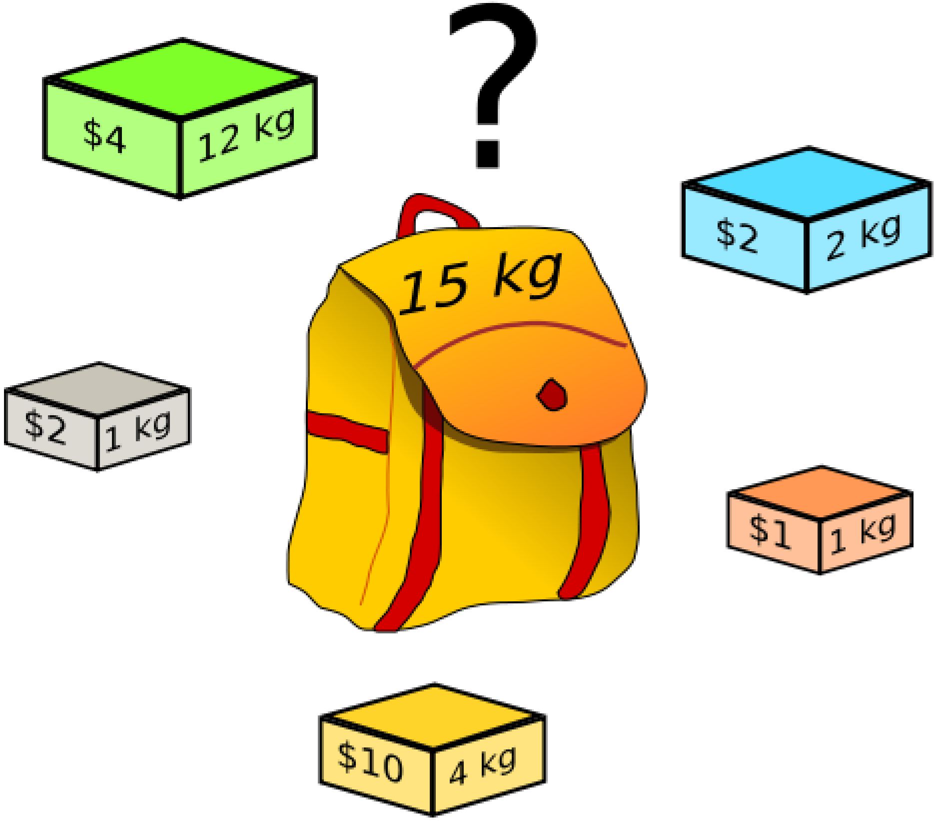
- Implement two functions for computing F(n)
  - V1 : using recursion
  - V2 : using a 1D array
  - V3 : using an extra array to identify the optimal set of coins

- Count the number of comparisons carried out

- Tables ?
- Complexity order ?

The 0-1 Knapsack Problem

- Find the most valuable subset of items, that fit into the knapsack



[Wikipedia]

The 0-1 Knapsack Problem

- Given n items
  - Known weight $w_1, w_2, \ldots, w_n$
  - Known value $v_1, v_2, \ldots, v_n$
- A knapsack of capacity W
- Which one is the / a most valuable subset of items that fit into the knapsack ?
  - More than one solution ?

The 0-1 Knapsack Problem

- How to formulate ?

max $\sum x_i v_i$

subject to $\sum x_i w_i \leq W$

with x_i in $\{0, 1\}$

The 0-1 Knapsack Problem

- An alternative to exhaustive search is to use a simple heuristics
  - Rule to construct a feasible solution step-by-step
  - Sometimes, only an approximate solution is found
- Very simple idea:
  - Successively choose the most valuable item that still fits into the knapsack
- Apply it to the example
  - Do you get the optimal solution ?

The Principle of Optimality

- The 0-1 Knapsack Problem satisifies the Principle of Optmality !!
- We have solved it by exhaustive search…
- Now, we can solve it using Dynamic Programming !!
- Recurrence ?

[Wikiedia]

The 0-1 Knapsack Problem

- Particular instance (i, j)
  - The first i items ($1 \leq i \leq n$)
    - Weights $w_1, w_2, \dots , w_i$
    - Values $v_1, v_2, \dots , v_i$
  - Knapsack capacity j ($1 \leq j \leq W$)
- Value of an optimal solution to instance (i, j) ?
  - $V[i, j] = ?$

The 0-1 Knapsack Problem

- Goal : V[n, W] = ?
- Recurrence ?
- Trivial cases
  - V[0, j] = 0 , for all j ≥ 0
  - V[i, 0] = 0 , for all i ≥ 0

The 0-1 Knapsack Problem

- General cases:
- The ith item does not fit into the knapsack
  - $V[i, j] = V[i-1, j]$, if $j - w_i < 0$
- The ith item fits into the knapsack
  - $V[i, j] = \max\{V[i-1, j], v_i + V[i-1, j - w_i]\}$, if $j - w_i \geq 0$

The 0-1 Knapsack Problem

- To determine V[i, j], if $(j - w_i) \geq 0$ inspect
  - Element in the same column and previous row
  - Element in column $(j - w_i)$ and previous row
- How to proceed ?
  - Fill the table row by row or column by column
- Implement an iterative function !!

The 0-1 Knapsack Problem

- Example
  - Capacity W = 10
  - 4 items
    - Item 1 : w = 7 ; v = $42
    - Item 2 : w = 3 ; v = $12
    - Item 3 : w = 4 ; v = $40
    - Item 4 : w = 5 ; v = $25

- Optimal solution
  - Value ?
  - Which items ?
    - Trace back the computations !!

The 0-1 Knapsack Problem

- Complexity ?
  - O(n W)
  - Pseudo-Polynomial !!

- It depends on the magnitude of W !!
  - Not just on the number of items
  - It will take much time for very large values of W !!

- What happens, if W increases and we need an additional bit to represent its value ?

The 0-1 Knapsack Problem

- BUT, it is a NP-Complete problem !!
  - Exhaustive search is exponential
  - Is there a contradiction ?

- Number of bits needed to represent W ?
  - O(log W)

- Complexity in terms of that number of bits ?
  - $O(2^{\log W})$
  - Exponential !!

The 0-1 Knapsack Problem

- Could it be different ?
  - What would that entail ?

- Weakly NP-Complete versus Strongly NP-Complete

- The dynamic programming algorithm serves our purposes !!
  - Except for "exponentialy large" values of W

Tasks – V1 + V2

- Implement two functions for computing the solution to an instance of the Knapsack problem
- V1 : a recursive function using the recurrence defined for the DP approach
- V2 : an iterative function implementing the DP algorithm
    - How to identify items belonging to the solution ?

Tasks – V1 + V2

- How to analyze ?
- Register execution times for some test instances
- What happens if we consider
  - 1 more item / 2 more items / …
  - twice the number of items ?
- Extrapolate the execution time for much larger problem instances

Solution – Dynamic Programming

```
0-1-Knapsack - Dynamic Programming Solution

Item Values: [None, 42, 12, 40, 25]
Item Weights: [None, 7, 3, 4, 5]

Capacity: W = 0        Optimal value: V = 0    Items = []
Capacity: W = 1        Optimal value: V = 0    Items = []
Capacity: W = 2        Optimal value: V = 0    Items = []
Capacity: W = 3        Optimal value: V = 12   Items = [2]
Capacity: W = 4        Optimal value: V = 40   Items = [3]
Capacity: W = 5        Optimal value: V = 40   Items = [3]
Capacity: W = 6        Optimal value: V = 40   Items = [3]
Capacity: W = 7        Optimal value: V = 52   Items = [2, 3]
Capacity: W = 8        Optimal value: V = 52   Items = [2, 3]
Capacity: W = 9        Optimal value: V = 65   Items = [3, 4]
Capacity: W = 10       Optimal value: V = 65   Items = [3, 4]
Capacity: W = 11       Optimal value: V = 82   Items = [1, 3]
Capacity: W = 12       Optimal value: V = 82   Items = [1, 3]
```

The Coin-Changing Problem

- Make change for an amount A
- Available coin denominations
  - Denom[1] > Denom[2] > … > Denom[n] = 1
- Use the fewest number of coins !!
- Assumption
  - Enough coins of each denomination !!

The Coin-Changing Problem

- How to formulate ?

$$\min \sum x_i$$

subject to $\sum x_i \, d[\, i \,] = A$

with $x_i = 0, 1, 2, \ldots$

- Compare with the 0-1 Knapsack formulation

The Coin-Changing Problem

- Particular instance (i, j)
  - Amount j
  - Use the smallest (n-i+1) coin denominations ($1 \leq i \leq n$)
- Value of an optimal solution to instance (i, j) ?
  - Minimum number of coins to make change for amount j
  - C[i, j] = ?
- Recurrence ?
- Optimal solution ? : C[1, A] = ?

The Coin-Changing Problem

- Trivial cases
  - C[n, j] = j , for all j ≥ 0
  - C[i, 0] = 0 , for all i ≥ 0
- How to establish the recurrence ?
- Try to do it !!
- Note
  - Minimization problem
  - Compute row by row
  - How to start ?

The String Alignment Problem

- Strings S and T
  - Length n and m, respectively
- Sometimes an "exact matching" is not possible !!
  - DNA
    - Nature : mutations !!
    - Lab errors
    - Computational errors !!
- "Soft matching" !!
  - The string alignment problem

The String Alignment Problem

- Q1 : How to proceed if there is no exact matching ?
- String alignment !
- Introduce gaps in order to maximize the number of coincident chars
- Example
  - TTATGCATAC–C–TCATGGGTACT
  - TTACGCGTACTCATGGTAC–T—T
  - Number of coincident chars ?

The String Alignment Problem

- Q2 : How to evaluate the score of a given string alignment ?

- How to weigh

  - Matches : $\sigma(X,X) = ?$
  - Mutations : $\sigma(X,Y) = ?$
  - Insertions : $\sigma(–,Y) = ?$
  - Deletions : $\sigma(X, –) = ?$

- How to compute a final score ?

The String Alignment Problem

- A simple scoring matrix

	A	C	G	T	–
A	+2	-1	-1	-1	-2
C	-1	+2	-1	-1	-2
G	-1	-1	+2	-1	-2
T	-1	-1	-1	+2	-2
–	-2	-2	-2	-2	–

The String Alignment Problem

- Q3 : How to compute an optimal (i.e., maximum score) alignment ?
- Ideal situation ?
- Is there just one optimal alignment ?
- How to proceed ?
  - Brute-force ?
  - …

The String Alignment Problem

- Input
  - Strings S and T
    - Length n and m, respectively
- Aim
  - Determine an optimal alignment of S* and T*
  - I.e., with maximal score $\sigma_{opt}(S,T)$
- S* and T* have the same length !!
- And are obtained by introducing gaps
- A gap does not appear simultaneously in the same position of S* and T*

The String Alignment Problem

- Alignment example

ACGAGTTCACT
CTGGCTTGGAT

AC–GA–GTTC–ACT
–CTGGCT–TGGA–T

- Try alternatives !!

The String Alignment Problem

- Brute-force approach ?
- Consider all possible gap insertions in each string !!
- Align and compare all possible string pairs !!
- Exponential approach !!

The String Alignment Problem

- Use Dynamic Programming !!

- Issues
  - Simplest / base cases ?
  - How to establish a recurrence ?

- $\alpha(S[0..i],T[0..j]) = ?$
  - Score of the optimal alignment between S[0..i] and T[0..j]
  - Simplify the notation : $\alpha[i][j]$

Simplest cases

- $\alpha[0][0] = 0$
  - Matching two empty strings !!
- $\alpha[0][j] = \sum \sigma(-,T[k]) = \alpha[0][j-1] + \sigma(-,T[j])$
  - Matching the empty string S[0] to string T[1..j]
- $\alpha[i][0] = \sum \sigma(S[k], -) = \alpha[i-1][0] + \sigma(S[i], -)$
  - Matching string S[1..i] to empty string T[0]

Recurrence

- α[i][j] = max {

  α[i-1][j] + σ(S[i], –),   // S[i] matches a gap

  α[i][j-1] + σ(–,T[j]) ,   // T[j] matches a gap

  α[i-1][j-1] + σ(S[i],T[j])   // S[i] matches T[j]

  }

The String Alignment Problem

- Where is the optimal score ?
  - $\sigma_{opt}(S,T) = \alpha[n][m]$
- Complexity order ?
- How to trace back the computations ?
- How to identify the optimal gap placement ?

Example

- Compute the optimal alignment score for strings AAAC and AGC
- What is the score ?
- Is there just one optimal alignment ?

Other Problems

- The longest common subsequence problem
- Constructing optimal binary search trees
- The chain matrix multiplication problem
- Warshall's algorithm for the transitive closure of a directed graph
- Floyd's algorithm for the all-pairs shortest path problem in a connected graph
- …

Dynamic Programming – Recap

- General algorithm design technique
- Apply to
  - Computing recurrences
  - Solving optimization problems
- Problem solution expressed recursively
- BUT, proceed bottom-up and store results for later use

Dynamic Programming – Recap

- Proceed bottom-up and store results for later use
- Big advantage, if sub-problems overlap !!
- NOW, there is no need to repeatedly solve the same sub-problems !!
- Iterative algorithms with "acceptable" complexity order

References

- A. Levitin, *Introduction to the Design and Analysis of Algorithms,* 3rd Ed., Pearson, 2012
  - Chapter 8
- R. Johnsonbaugh and M. Schaefer, *Algorithms,* Pearson Prentice Hall, 2004
  - Chapter 8
- T. H. Cormen et al., *Introduction to Algorithms*, 3rd Ed., MIT Press, 2009
  - Chapter 15