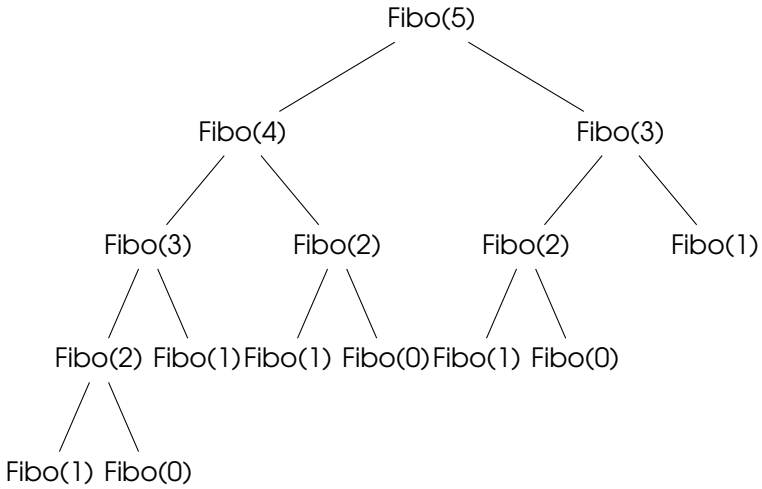
Contents

1. Lecture 3

Now we consider problems that may seem difficult until we approach them efficiently. This week we use **dynamic programming**.

DP can be applied when the solution of a problem can be found from solutions to sub-problems.
Solutions to subproblems are stored in memory (the DP table). To solve a subproblem we make use of the stored information on other subproblems.
Example 1:

```c
#include <stdio.h>
#include <stdlib.h> // EXIT_SUCCESS
int Fibonacci(int n){
  if (n == 0 || n == 1) return n;
  else
    return (Fibonacci(n-1) + Fibonacci(n-2))
    ;
}
int main(void){
    int n=5;
    printf( "Fibonacci(%d)=%d\n" , n ,
    Fibonacci(n) );
    return EXIT_SUCCESS;
}
```



Notice that there is repeated work that can be avoid.

```c
#include <stdio.h>
#include <stdlib.h> // EXIT_SUCCESS

int Fibonacci(int n){
    if(n == 0 || n == 1)    return n;
    int tab_fib[n];
    tab_fib[0]=0;
    tab_fib[1]=1;
    for(int i=2;i<n;i++){
        tab_fib[i] = tab_fib[i-1] +tab_fib[i-
    2];
    }
    n = tab_fib[n-1];
    return n;
}
int main(void){
    int n=5;
    printf( "Fibonacci(%d)=%d\n" , n ,
    Fibonacci(n) );
    return EXIT_SUCCESS;
}
```

0	1	2	3	4	5
0	1	1	2	3	5

We can imagine that we fill the table from the left to the right. Filling one value is constant time and there is n value to fill therefore O(n).

The difference between algorithm 1 and algorithm 2 to compute Fibonacci numbers illustrates the difference between recursion and dynamic programming:
Dynamic programming builds new states of computation from previous states without repeating the same computations.

Each Dynamic Programming assignment requires the following steps in the answer:

1. The subproblems to solve.
2. The optimal value, expressed in terms of the subproblems
3. Initial values.
4. The recurrence
5. Analysis of the used space.
6. Analysis of the running time.

Let's see this through an example.
A contiguous subsequence of a list S is a subsequence made up of consecutive elements of S.For instance, if $S = [5, 15, -30, 10, -5, 40, 10]$ then $[15, -30, 10]$ is a contiguous subsequence but $[5, 15, 40]$ is not.
Given a list of numbers $[a_1, a_2, \ldots, a_n]$, find the maximum sum of a contiguous subsequence (a subsequence of length zero has sum zero).

1. Subproblem: use a state s(j) which is the value of the maximum contiguous subsequence ending in the j th number of the list.
2. optimal value: $\max(0, max\ s(j))$
3. initial value $s(1) = a_1$
4. the recurrence: $s(j+1) = \max\left(a_{j+1}, s(j) + a_{j+1}\right)$ for $j = 1, \ldots, n-1$.
5. Complexity in space: we only need to know s_j and a_{j+1} to compute$s(j+1)$ and keep track of the maximum $s(j)$ seen so far. Therefore $O(1)$.
6. Running time: one subproblem take 0(1) to be solve but there is n subproblems. Therefore $O(n)$.

Example of application with $[2, -4, 5, 3, 8]$:

$s(1)$	$s(2)$	$s(3)$	$s(4)$	$s(5)$
2	$\max(-4, -4+2) = -2$	$\max(5, 5-2) = 5$	$\max(5, 5+3) = 8$	$\max(8, 8+8) = 16$

We don't need to keep all these numbers, we only need to keep max ($s(j)$, $s(j+1)$) and the location of the maximum:

$x = 0$	$\rightarrow$	$x = 1$	$\rightarrow$	$x = 2$	$\rightarrow$	$x = 3$	$\rightarrow$	$x = 5$
2		-2		5		8		16

The location x allows us to restitute the subsequence by subtracting from the max until we reach 0.

Given two strings x and y of size m and n, find the size longest common subsequence:
Proceed from the end of the strings:

- If $x_m = y_n$ add 1 to the number of symbols in $LCS(x_{m-1}, y_{n-1})$
- If $x_m \neq y_n$: skip last symbol from x or last symbol from y and decide which symbol to skip by comparing$LCS(x_m, y_{n-1})$ and$LCS(x_{m-1}, y_n)$

In other words,

$$LCS(x_m, x_n) = \begin{cases} 1 + LCS(x_{m-1}, x_{n-1}) & \text{if } x_m = x_n \\ \max(LCS(x_m, x_{n-1}), LCS(x_m, x_{n-1})) & \text{otherwise} \end{cases}$$

If x_m or $x_n = \emptyset$ then $LCS(x_n, y_n) = 0$
Let x_m = lit and y_n = ile.

$$\begin{aligned} LCS(x_m, y_n) &= 1 + \text{LCS(lit,ile)} \\ &= 1 + \text{max(LCS(li,ile) , LCS(lit,il))} \\ &= \text{max(max(LCS(l,ile) , LCS(li,il)) ,max(LCS(li,il) , LCS(li,i)))} \\ &= \text{max(max(max(0, LCS(l,il)) ,max(LCS(l,il) , LCS(li,i))) ,max(max(LCS(l,il) , LCS(li,i)), 1))} \\ &\vdots \\ &= 2 \end{aligned}$$

We must therefore recursively calculate those expression.

```c
#include <stdio.h>
#include <stdlib.h> // EXIT_SUCCESS

int LCS(int i, int j) {
    if (i==0 || j==0) return 0;
    else if (x(i) == y(j))  return lcsRec(i-1,j-1) + 1;
    else    return max(lcsRec(i-1,j),lcsRec(i,j-1));
}
int main(void){
    int n=5;
    printf( "LCS(%d)=%d\n" , n , LCS(n) );
    return EXIT_SUCCESS;
}
```

Running time:
$T(2n) = k$ if $n = 0$
$T(2n) = T(2n-2) + k$ if $x[n] = y[n]$
$T(2n) = 2T(2n-1) + k$ if $x[n] \neq [n]$

$$T(2n) = 2T(2n-1) = 2(2T(2n-2)) = 4T(2n-2) = 8T(2n-3) = 4^n$$

Can we find a better solution ? What we can do instead (...)

Let's look matrix multiplication. Let $A : p \times q$ and $B : q \times r$ requires $p \times q \times r$ scalar multiplications.

Sizes: 50x10, 10x40, 40x30, 30x5
We can compute ABCD in several ways:
(((A B) C) D) : 87500 multiplications
((A (B C)) D) : 34500 multiplications
((A B) (C D)) : 36000 multiplications
(A ((B C) D)) : 16000 multiplications
(A (B (C D))) : 10500 multiplications

Optimization problem: what's the best way to put the brackets?

Let's state this in more generality: rather than multiplying ABCD, we want to multiply $A_1, \ldots, A_n$

Let $m(L,R)$ be the optimal number of multiplications needed to compute $A_L, \ldots, A_R$ where $1 \leq L$ and $R \leq n$. Therefore our objective is to compute $m(1,L)$.
It's clear that $m(L,R) = 0$ if $L = R$ but $m(L,R) =?$ if $L < R$.

We need to introduce more notation about the size: A_i is a $c_i \times c_{i+1}$ matrix.

A convenient way to think about this is to think is thinking about the last multiplication in term of i:

$$(A_L, \ldots, A_i)(A_{i+1}, \ldots, A_R) \text{ where } L \leq i < R$$

Therefore we need to calculate the first part, the second part and finally calculate both parts.

$$m(L,R) = m(L,i) + m(i+1,R) + c_{L-1} \cdot c_i \cdot c_R$$

The thing is we don't know where is the best place to put the brackets so we have to try all positions for i:

$$m(L,R) = \min_{L \leq i < R} \{m(L,R) = m(L,i) + m(i+1,R) + c_{L-1} \cdot c_i \cdot c_R\}$$

Let's think in shape of a table, what are the easy value to fill in:

- the property $m(L,R) = 0$ if $L = R$
- when $L > R$

L \ R	1	2	3	4	5	6
1	0				■	
2	-	0				
3	-	-	0			
4	-	-	-	0		
5	-	-	-	-	0	
6	-	-	-	-	-	0

Let's say we want to compute the value in the cell with a red square. One of the way to put bracket is $(A_1)(A_2A_3A_4A_5)$. To do that, you need to watch the value from the cells (1,1) and (2,5). For $(A_1A_2)(A_3A_4A_5)$, we look (1,2) and (3,5) etc... How many entries to fill in the matrix? $0.5n^2$
Effort to compute one entry? Since we're looking for the minimum we have to check at least every entry so $O(n)$. Therefore the running time is $O(n^3)$.

Exercise 1 You are going on a long trip. You start on the road at mile post 0. Along the way there are n hotels, at mile posts $a1 < a2 < \cdots < an$, where each a_i is measured from the starting point. The only places you are allowed to stop are at these hotels, but you can choose which of the hotels you stop at. You must stop at the final hotel (at distance an), which is your destination. You'd ideally like to travel 200 miles a day, but this may not be possible (depending on the spacing of the hotels). If you travel x miles during a day, the penalty for that day is $(200-x)^2$.
You want to plan your trip so as to minimize the total penalty that is, the sum, over all travel days, of the daily penalties.
Give an efficient algorithm that determines the optimal sequence of hotels at which to stop.

Exercise Yuckdonald's is considering opening a series of restaurants along Quaint Valley Highway (QVH). The n possible locations are along a straight line, and the distances of these locations from the start of QVH are, in miles and in increasing order: $m_1, m_2, \ldots, m_n$. The constraints are as follows:

- At each location, Yuckdonald's may open at most one restaurant. The expected profit from opening a restaurant at location i is p_i , where $p_i > 0$ and $i = 1, 2, \ldots, n$.
- Any two restaurants should be at least k miles apart, where k is a positive integer.

Give an efficient algorithm to compute the maximum expected total profit subject to the given constraints.

Exercise You are given a string of n characters $s[1 \ldots n]$, which you believe to be a corrupted text document in which all punctuation has vanished (so that it looks something like "itwasthebestoftimes..."). You wish to reconstruct the document using a dictionary, which is available in the form of a Boolean function dict(·): for any string w,

$$dict(w) = \begin{cases} true, & \text{if w is a valid word} \\ false, & \text{otherwise} \end{cases} \tag{1.1}$$

Give a dynamic programming algorithm that determines whether the string $s[]$ can be reconstituted as a sequence of valid words. The running time should be at most $O(n^2)$, assuming calls to dict take unit time.

Exercise Pebbling a checkerboard. We are given a checkerboard which has 4 rows and n columns, and has an integer written in each square. We are also given a set of 2n pebbles, and we want to place some or all of these on the checkerboard (each pebble can be placed on exactly one square) so as to maximize the sum of the integers in the squares that are covered by pebbles. There is one constraint: for a placement of pebbles to be legal, no two of them can be on horizontally or vertically adjacent squares (diagonal adjacency is fine).

(a) Determine the number of legal patterns that can occur in any column (in isolation, ignoring the pebbles in adjacent columns) and describe these patterns.

Call two patterns compatible if they can be placed on adjacent columns to form a legal placement. Let us consider subproblems consisting of the first k columns $1 \leq k \leq n$. Each subproblem can be assigned a type, which is the pattern occurring in the last column.

(b) Using the notions of compatibility and type, give an $O(n)$-time dynamic programming algorithm for computing an optimal placement.

Exercise Let us define a multiplication operation on three symbols a, b, c according to the following table thus ab = b, ba = c, and so on. Notice that the multiplication operation defined by the table is neither associative nor commutative.
Find an efficient algorithm that examines a string of these symbols, say bbbbac, and decides whether or not it is possible to parenthesize the string in such a way that the value of the resulting expression is a. For example, on input bbbbac your algorithm should return yes because ((b(bb))(ba))c = a.

Solution 1

Subproblem to solve	Define $f(i)$ as the minimum cost for traveling from location 0 to location i.
Optimal value	$f(n)$
Initial value	$f(0) = 0$
Recurrence	For $1 \leq i \leq n$: $\min_{0 \leq j \leq i} (f_j + (200 - (a_i - a_j)^2)$
Space used	The size of the DP table is $O(n)$
Running time	The time used is $O(n^2)$ since it takes $O(n)$ time to compute each if the $O(n)$ values.

Solution

Subproblem to solve	Define $f(i)$ as the maximum profit expected for traveling from location 0 to location m_i.
Optimal value	$\max(f(1), ..., f(i))$
Initial value	$f(1) = p_1$
Recurrence	For $1 \leq i \leq n$: $f(i) = p_i + \max_{1 \leq j < i} (f(i) \mid m_i - m_k \geq k)$
Space used	The size of the DP-table is $O(n)$.
Running time	The time to compute one value in the table is $O(n)$. Hence, the total time is $O(n^2)$.

Solution

Subproblem to solve	Define $$f(i) = \begin{cases} \text{true}, & \text{if } s[1..i] \text{ is a valid substring} \\ \text{false}, & \text{otherwise} \end{cases} \quad (1.2)$$
Optimal value	$f(n) =$true
Initial value	$f(0) =$true
Recurrence	For $i = 0, \ldots, n$: $f(i) =$ true if there is a $j \in \{0, \ldots, i-1\}$ such that $f(j) =$ true and $s[j+1, \ldots, i]$ is a valid word.
Space used	The size of the DP table is $O(n)$
Running time	The time used is $O(n^2)$ since it takes $O(n)$ time to compute each if the $O(n)$ values.

Solution (a) There are 8 different patterns for a single column: 3 patterns with 2 pebbles, 4 with 1 pebbles and 1 pattern with 0 pebbles. (b)

Subproblem to solve	Let $f(k,t)$ be the total profit that can be obtained from the first k columns under the restriction that the k-th column is of type t: $k = 1,...,n$ and $t = 1,...,8$.
Optimal value	$\max(f(n,t)$ where $1 \leq t \leq 8$
Initial value	Let Profit(k, t) be the profit obtained from column k if pattern t is chosen. Then, the initial values are f(1, t) = Profit(1, t) for all t.
Recurrence	For each type t and $2 \leq k \leq n$ the recursion is: $f(k,t) =$ Profit(k, t) $+ \max_s \{f(k-1,s) \mid$ s compatible with t$\}$
Space used	The number of types is 8, so we never have to store more than $2 \cdot 8$ values in memory. As this does not depend on the size of the board, the workspace needed to compute the optimal value is only $O(1)$.
Running time	The number of subproblems is $8n$, so that is $O(n)$. To compute one value in the DP-table only requires taking a maximum over at most 8 elements, so the time that takes is $O(1)$. So the total time is $O(n)$. values.

Solution Let $s = s_1 s_2 \cdots sn$ be the string and denote by $s[i..j]$ the substring $s_i s_{i+1} \cdots s_j$.

Subproblem to solve	Define $f(i,j,v) =$ true iff substring $s[i \ldots j]$ can be turned into v, where $v \in \{a,b,c\}$ and $1 \leq i \leq j \leq n$.
Optimal value	$f(1,n,a) =$ true.
Initial value	$f(i,i,v) =$ true iff $s_i = v$, where $v \in a,b,c$ and $1 \leq i \leq n$.
Recurrence	$f(i,j,v)$ = true iff there is some $k \in \{i,\ldots,j-1\}$ and $v_1, v_2 \in \{a,b,c\}$ such that: • $f(i,k,v1) =$ true and • $f(k+1,j,v2) =$ true and • $v_1 v_2 = v$.
Space used	The size of the table is $O(n^2)$.
Running time	The time to compute one value in the DP-table is O(n). So the total time is $O(n^3)$.