

17

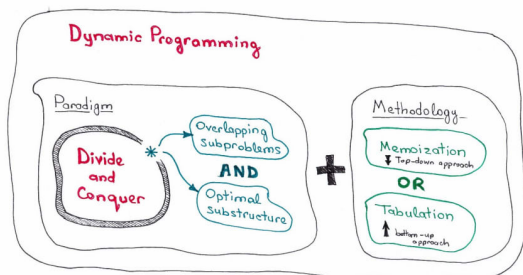
**Submission deadline:** 14 Nov, 23:59

- Write your answers(images) and C++ code inside the *answers* folder in order to generate a single PDF file. Replace the images and cpp files that are already included in the project. Do not change the file name.
- Read the questions carefully and write your answers clearly. Answers that are not legible and that doesn't follow the format will not have any score.

Outcomes:

- Apply appropriate mathematical and related knowledge to computer science.
- Analyze problems and identify the appropriate computational requirements for its solution.

## A. Warm up



In the previous lecture we presented 2 approaches to store the results of subproblems that are progressively used to construct the final solution to a particular problem:

- Memoization ( top-down approach )
- Tabulation ( bottom-up approach )

### Problem 1 (Outcome b) - 4 points

Remember the idea behind the top-down approach and for each of the following items implement 2 algorithms in C++, one with the regular recursive implementation and the other using the memoization approach:

- Fibonacci function

```
#include <iostream>
#include <vector>
using namespace std;
```

```
int recursiveFibonacci(int n){
    if(n==1 or n==0)
        return n;
    else
        return recursiveFibonacci(n-1)+recursiveFibonacci(n-2);
}
```



```
int worker(int n, int* dm,int size){
    if(dm[n] != -1){
        return dm[n];
    }
    else{
        if(n==0 or n ==1){
            dm[n]=n;
            return n;
        }
        int result= worker(n-1,dm,size)+worker(n-2,dm,size);
        dm[n]=result;
        return result;
    }
}
```



```
int dynamicFibonacci(int n){
    int dm[n];
    for(int i = 0; i <= n; i++){
        dm[i] = -1;
    }
    return worker(n,dm,n);
}
```



```
int main() {
    cout<<recursiveFibonacci(10)<<endl;
    cout<<dynamicFibonacci(1000)<<endl;
}
```

- Factorial function

```
#include <iostream>
#include <vector>
using namespace std;
```

```
//Se usa long long int para probar con valores más grandes, la tasa
↪ de crecimiento del factorial es sumamente grande
```

```
long long int recursiveFactorial(int n){
    if(n==1)
        return n;
    else
        return n * recursiveFactorial(n-1);
}
```

```
long long int worker(int n, vector<long long int>& dm){
    int size=dm.size();
    if(size-1 >= n){
        return dm[n];
    }
    else{
        long long int result;
        if(n==1){
            result=n;
        }
        else{
            result= worker(n-1,dm) * n;
        }
        dm.push_back(result);
        return result;
    }
}
```

```
long long int dynamicFactorial(int n){
    vector<long long int> dm;
    return worker(n,dm);
}
```

```
int main() {
    cout<<recursiveFactorial(20)<<endl;
    cout<<dynamicFactorial(20)<<endl;
}
```

✓  
should be  
"global"

## Problem 2 (Outcome b) - 2 points

Answer the following questions:

- What are the differences between the memoization and tabulation approaches?  
Which one is faster?

- What are the main conditions that a problem must present in order to be solvable using dynamic programming? Explain those conditions with the C++ algorithms proposed above.

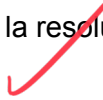

- La principal diferencia entre memoization y tabulation es que dado un problema, en memoization se usa recursión hacia los subproblemas mientras estos se resuelven de forma recursiva, almacenando los valores obtenidos en una estructura. Este es un approach Top-Down ya que va desde el problema original hacia los subproblemas más pequeños que el original. Tabulation es un approach iterativo que busca justamente evitar la recursión invirtiendo la forma en cómo se resuelven los problemas, este usa un Bottom-up approach, es decir, parte de él/los casos base y llega a la resolución del problema original de forma iterativa. La recursión tiene un overhead que es la que busca evitar tabulation, ya que esto la hace más lenta, por ende, **tabulation es un método más rápido que memoization, a pesar de que ambos tengan un time complexity de  $O(n)$** . Como ejemplo se muestra el pseudocódigo de fibonacci para apreciar la diferencia entre ambos métodos de resolución:

- Memoization

```
fib(n, dp):  
    if dp[n] != null:  
        return dp[n]  
    if n==0 or n==1:  
        dp[n]=n  
        return n  
    else:  
        result = fib(n-1,dp) + fib(n-2,dp)  
        dp[n]=result  
        return result
```

- Tabulation

```
fib_tab(n):  
    if n==0 or n==1:  
        return n  
    dp= array of size n +1  
    dp[0]=0  
    dp[1]=1  
    for i=2 to n:  
        dp[i]=dp[i-1] + dp[i-2]  
    return dp[n]
```

- Las dos condiciones para poder usar programación dinámica en la resolución de un problema es que el problema cumpla lo siguiente: 
  - Subproblemas superpuestos (overlapping): 
  - Subestructura óptima: Entiéndase como que el problema se resuelve de forma óptima si y sólo si, los subproblemas que contiene también son resueltos de forma óptima por el mismo algoritmo.

## B. Dynamic Programming Problems

### B.1 0-1 Knapsack Problem

Given a backpack that can carry  $W$  kg and a number of  $n$  items where each item has an associate weight( $w_i$ ) and value( $v_i$ ) we want to know what is the maximum value that we can carry in the backpack.

Consider that if we select any of the items and put it into the backpack we are increasing the total value(\$) but we are decreasing the capacity(kg) of the backpack.

#### Problem 3 (Outcome b) - 2 points

Consider the items below to calculate the maximum value that we can put into the backpack.

Item	1	2	3	4	5
Weight (kg)	1	2	4	1	12
Value (\$)	1	2	10	2	4

- Write down your solution using the tabulation method. ( C++ code shouldn't be attached )
- Explain the procedure to get the items that maximize the value in the backpack and write down those items.

**Tabulation**

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	0	1	2	3	3	3	3	3	3	3	3	3	3	3	3	3
3	0	1	2	3	10	11	12	13	13	13	13	13	13	13	13	13
4	0	2	3	4	10	12	13	14	15	15	15	15	15	15	15	15
5	0	2	3	4	10	12	13	14	15	15	15	15	15	15	15	15



**Explicación**

Se empieza de la esquina inferior derecha (Celda resaltada de amarillo), en caso esta tenga un valor diferente a la celda que se encuentra inmediatamente arriba, quiere decir que dicho item sí se ha tomado en cuenta para la resolución del problema por lo que nos desplazamos hacia la izquierda tantos espacios como peso tenga la celda, caso contrario, no se ha tomado en cuenta dicho item, nos desplazamos un espacio hacia arriba y realizamos el mismo proceso, así hasta no tener a donde subir, o hasta no tener hacia donde moverse a la izquierda.

→ ooo missing subarray construction X

## B.2 Maximum Contiguous Subarray Sum Problem

Consider the maximum-subarray problem presented in the Chapter 4.1 of the Introduction to Algorithms book.



**Problem 4 (Outcome b) - 3 points**

Use the rate of change of the daily stock prices presented in the book (Fig. 4.1) to calculate the maximum sum of a contiguous subarray using the tabulation method. ( No C++ code required in this problem )

- Write down the table and the procedure used to solve the problem using dynamic programming.
- Once the table is constructed, explain how can we obtain the contiguous subarray given the maximum sum?

Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price \$	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Rate of Change		13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7



Se desea maximizar un subarray del array rate of change, ya que es quien define las ganancias.

## Tabulation

Sea cada celda la ganancia máxima hasta el día correspondiente a dicha celda. En cada día la comparación que maximiza la ganancia está dada por:

**Max(Rate of Change del nuevo día, Ganancia acumulada de n días previos + Rate of Change del nuevo día)**

Se parte de que el primer día no hay rate of change por lo que se empieza con máximo subarray de 13 (113-100), luego se va iterando con cada valor,  $-3 < 10$  (13-3) y así se sigue realizando las comparaciones con la lógica del recuadro.

Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Rate of Change	13	10	-15	20	17	1	-22	18	38	31	43	38	16	31	27	34

En otras palabras, empezar un nuevo arreglo o agregar mi nueva ganancia al existente. En base a esto, para obtener el máximo subarray se trabajan con i y j como iteradores, ambos inician en la primera posición. 'i' representará al máximo y 'j' a un temporal que validará como cambia el subarray conforme se itera la lista.

En cada nueva iteración de la lista se validará lo siguiente:

**J // Temp = MAX( newElement, Temp + newElement)**

Esto quiere decir que si el subarray tenía suma negativa, se empieza un nuevo array en vez de aumentar el existente. newElement solo será mayor a (Temp + newElement) con Temp negativo.

**I // MAXIMO = MAX( MAXIMO, Temp)**

Esto quiere decir que se verifica en cada iteración si hay que actualizar la suma nueva del subarray máximo, de no ser el caso, se mantiene el anterior.

- Para sacar los índices de inicio y fin se sigue un procedimiento sumamente similar, tomando en cuenta dos iteradores i (inicio) y j (fin) nuevos, tenemos que:

```
i = j = array[0]
```

```
//La lógica para temp y maximo y ya fue descrita
```

- para i:

Tomará el índice (pos) del elemento iterado si  $\text{Array}[\text{pos}] > \text{Array}[\text{pos}] + \text{maximo}$ , de lo contrario, i mantiene su posición actual

- para j:

Tomará el índice (pos) del elemento iterado si  $\text{Array}[\text{pos}] > \text{Array}[\text{pos}] + \text{temp}$ , de lo contrario, j mantiene su posición actual

### Problem 5 (Outcomes a, b) - 6 points

Implement 2 algorithms in C++ to solve the Maximum Contiguous Subarray Sum Problem given an array of positive and negative numbers. One algorithm should implement the solution using the Divide and Conquer approach from Chapter 4 and the other algorithm should implement a Dynamic Programming bottom-up approach.

Perform all the trials needed to see the asymptotic behavior of both algorithms considering that the Divide and Conquer approach should behave like a  $n\log(n)$  function and the Dynamic Programming algorithm should behave like a linear function.

### Recommendations

1. Implement both algorithms in C++ based on a given array of numbers ( positives and negatives ).
2. Write a function that given a size it generates a random array of numbers.
3. Write a function that measures the execution time of an algorithm
4. Record the execution time of both algorithms given an array of size  $k$  and repeat until  $k$  is sufficiently large.
5. Tune your trials and parameters in order to see an smooth execution time similar to the  $n\log(n)$  and linear functions.
6. Attach the source code and generate one picture with both results and attach that image in this problem.

### Divide and Conquer

```
#include <iostream>
#include <vector>
#include <chrono>
#include <random>
#include <fstream>
using namespace std;
using namespace chrono;
pair<pair<int,int>,long>findMaxCross(vector<int> &values, int min, int mid
↵ , int max){
    long sum=0;
    int posLeft=mid,posRight=mid;
    long rightSum=INT64_MIN;
    long leftSum=INT64_MIN;
    for(auto i=mid; i>=min;--i){
        sum+= values[i];
        if(sum>leftSum){
```

```

        leftSum=sum;
        posLeft=i;
    }
}
sum=0;

for(auto i=mid; i<=max;++i){
    sum+= values[i];
    if(sum>rightSum){
        rightSum=sum;
        posRight=i;
    }
}
return make_pair(make_pair(posLeft,posRight),leftSum+rightSum);
}

pair<pair<int,int>,long> findMaxSubArray(vector<int> &values, int min ,
↪ int max){
    if(min==max)
        return make_pair(make_pair(min,max), values[min]);
    else{
        int mid=(min+max)/2;
        auto left = findMaxSubArray(values, min, mid);
        auto right = findMaxSubArray(values, mid + 1, max);
        auto cross = findMaxCross(values, min, mid, max);

        if(left.second>=right.second and left.second>=cross.second) return
↪ left;
        else if(right.second>=left.second and right.second>=cross.second)
↪ return right;
        else return cross;
    }
}

pair<int,int> problem5(std::vector<int> & values){
    return findMaxSubArray(values,0,(int)values.size()-1).first;
}

int main() {
    vector<double> times;
    for(auto i=1;i<10000;++i){
        vector<int> values;
        for(auto j=0;j<i;++j)
            values.push_back(rand() % 1000 -1000);
    }
}

```

```

        high_resolution_clock::time_point start =
            ↪ high_resolution_clock::now();
        problem5(values);
        high_resolution_clock::time_point stop =
            ↪ high_resolution_clock::now();
        double duration = duration_cast<microseconds>(stop -
            ↪ start).count();
        times.push_back(duration);
    }

    ofstream file("dc_times.csv", std::ios::out);
    for(double time : times){
        file <<time << "\n" << std::flush;
    }
    file.close();
    return 0;
}

```

### Dynamic programming

```

#include <iostream>
#include <vector>
#include <chrono>
#include <random>
#include <fstream>
using namespace std;
using namespace chrono;

int problem5(std::vector<int> & values){
    int global = values[0]; //i in problem 4
    int temp = values[0]; //j in problem 4
    for (auto it= values.begin()+1; it!=values.end(); ++it) {
        if(temp<0) temp=0;
        temp += *it;
        if(temp>global)
            global=temp;
    }
    return global;
}

int main() {
    vector<double> times;
    for(auto i=1;i<10000;++i){
        vector<int> values;
        for(auto j=0;j<i;++j)
            values.push_back(rand() % 1000 -1000);
        high_resolution_clock::time_point start =
            ↪ high_resolution_clock::now();
    }
}

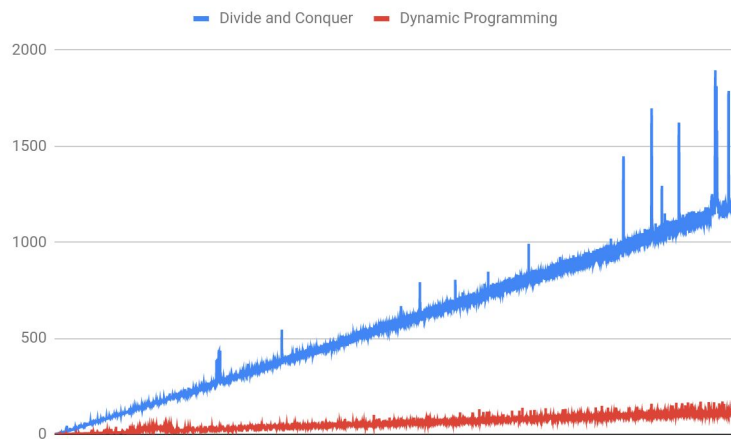
```

```
problem5(values);
high_resolution_clock::time_point stop =
    → high_resolution_clock::now();
double duration = duration_cast<microseconds>(stop -
    → start).count();
times.push_back(duration);
}

fstream file("dp_times.csv", std::ios::out);
for(double time : times){
    file << time << "\n" << std::flush;
}
file.close();
return 0;
}
```



Se puede observar en la gráfica un comportamiento  $O(n \lg n)$  para la versión Divide and Conquer y un comportamiento  $O(n)$  para Dynamic programming



you should consider the average of more trials to get an stable representation of  $n \log n$  and  $n$  functions

### B.3 Coin Change Problem

Given an amount  $A$  and a number of  $n$  coins of different denominations the Coin Change Problem ( or at least one variation ) consists in calculate the minimum number of coins



that we need to obtain the specific amount.

**Problem 6 (Outcome b) - 3 points**

3

Consider the following coins and write the table of the tabulation approach considering that the total amount is 18.

Coins	1	5	2	6	11	15
-------	---	---	---	---	----	----

Se busca minimizar la cantidad de monedas.

Para entregar el cambio, dado un conjunto N de monedas, las combinaciones están dadas por el hecho de elegir si usar una moneda o no usarla. Por ello existen 2 opciones de las cuales se elige la que tenga menos cantidad de monedas:

1. Elegir una nueva moneda junto a las demás monedas elegidas para la cantidad obtenida al restar el valor de la nueva moneda. ( $1 + \text{Coins}(X-c)$ , donde X es el monto y c es el valor de la nueva moneda elegida. )
2. Elegir la cantidad mínima de monedas para el cambio, sin elegir una nueva moneda, solo con las anteriores ya evaluadas.

La tabulación que se obtiene es la siguiente para \$18 de cambio:

0 6 4 6 0 7

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
5	0	1	2	3	4	1	2	3	4	5	2	3	4	5	6	3	4	5	6
2	0	1	2	1	2	1	2	2	3	3	2	3	3	4	4	3	4	4	5
6	0	1	1	2	2	1	1	2	2	3	2	2	2	3	3	3	3	3	3
11	0	1	1	2	2	1	1	2	2	3	2	1	2	2	3	3	2	2	3
15	0	1	1	2	2	1	1	2	2	3	2	1	2	2	3	1	2	2	3

1 + 2  
consider 15