

# Programação Dinâmica

16/10/2023

# Sumário

- Recap
- Números de Fibonacci
- Programação Dinâmica & Memoization
- $C(n,p)$  – O Triângulo de Pascal
- Números de Delannoy – Exercício adicional
- O Problema da Fileira de Moedas (“The Coin Row Problem”)
- Sugestões de leitura

# Recapitulação

Let's  
Recap

# Procura num Array – Tentaram fazer ?

- Dado um array de **n elementos** inteiros
- Encontrar a **1ª ocorrência** do **maior elemento**
- Divide-And-Conquer
  - Encontrar o **maior valor** do **1º sub-array** :  $((n+1) \text{ div } 2)$  elementos
  - Encontrar o **maior valor** do **2º sub-array** :  $(n \text{ div } 2)$  elementos
  - **Comparar** os dois valores e devolver o maior
- Quantas **comparações** ?

# Procura num Array – Nº de comparações

- $C(1) = 0$
- $C(2) = 1$
- $C(n) = 1 + C(n \text{ div } 2) + c((n + 1) \text{ div } 2)$
- Seja  $n = 2^k$
- $C(n) = 1 + 2 \times C(n / 2) = ?$
- Fazer o desenvolvimento telescópico ou aplicar o “Teorema Mestre”

# The Master Theorem

- Dada uma recorrência, para  $n = b^k$ ,  $k \geq 1$

$$T(1) = c \quad \text{e} \quad T(n) = a T(n / b) + f(n)$$

com  $a \geq 1$ ,  $b \geq 2$ ,  $c \geq 0$

- Se  $f(n)$  em  $\Theta(n^d)$ , em que  $d \geq 0$ , então

$$T(n) \text{ em } \Theta(n^d), \text{ se } a < b^d$$

$$T(n) \text{ em } \Theta(n^d \log n), \text{ se } a = b^d$$

$$T(n) \text{ em } \Theta(n^{\log_b a}), \text{ se } a > b^d$$

# Procura num Array – Nº de comparações

$$C(n) = 2 \times C(n / 2) + 1$$

$$f(n) = 1, f(n) \text{ em } \Theta(n^0), d = 0$$

$$a = 2, b = 2, a > b^d$$

$$C(n) \text{ em } \Theta(n)$$

# Generalização – The Smoothness Rule

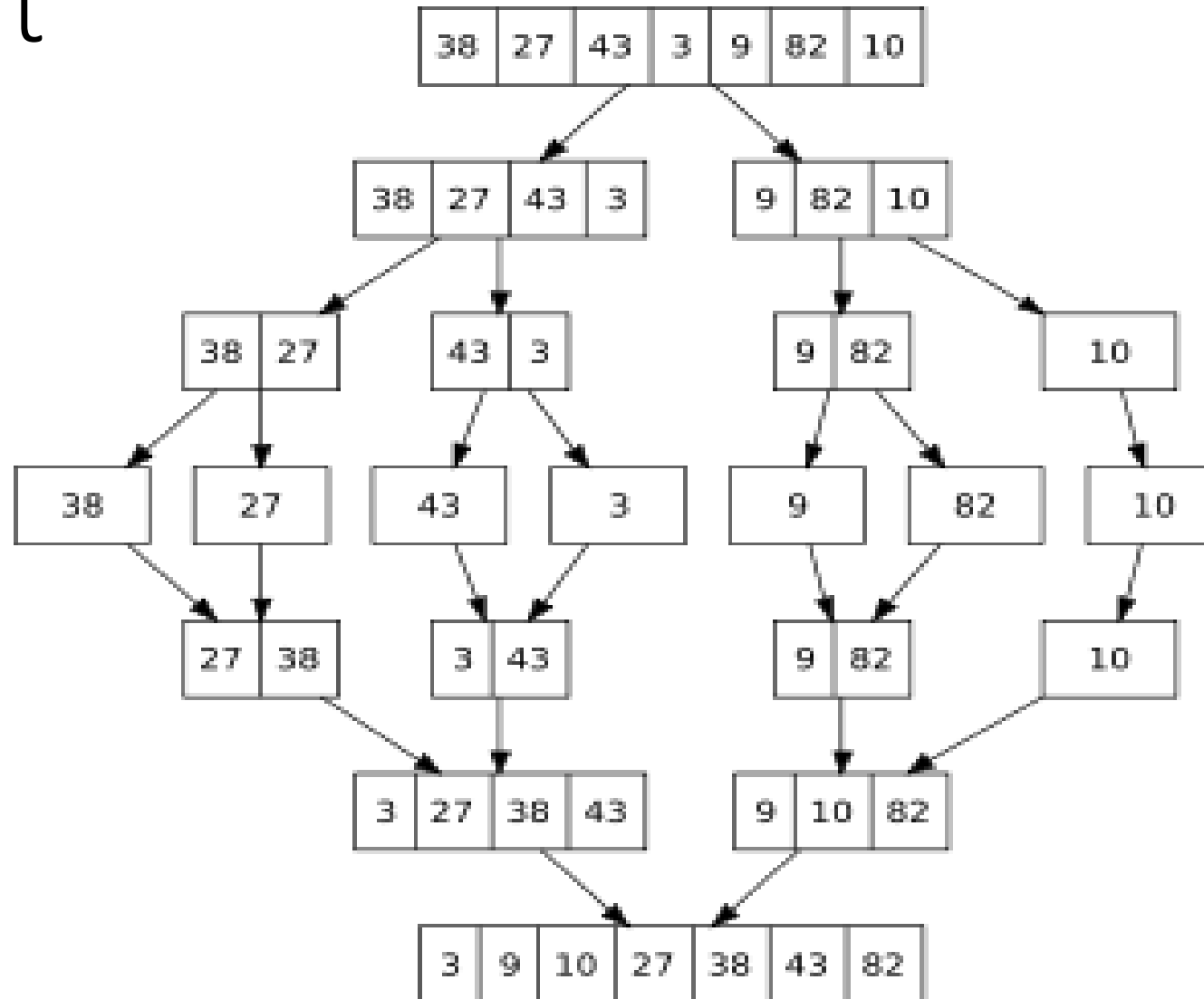
- $T(n)$  em  $\Theta(n)$  para valores de  $n$  que são potências de 2
- Então  $T(n)$  em  $\Theta(n)$ , para qualquer  $n$



# Mergesort

– Ordenação por Fusão

# Mergesort



**SUBDIVISÃO**

**FUSÃO**

**Tarefa:** associar a cada seta um rótulo que identifica a sequência pela qual as chamadas são executadas

[Wikipedia]


# Mergesort

```
// mergeSort(A, tmpA, 0, n - 1);

void mergeSort(int* A, int* tmpA, int left, int right) {
    // Mais do que 1 elemento ?

    if (left < right) {
        int center = (left + right) / 2;

        mergeSort(A, tmpA, left, center);
        mergeSort(A, tmpA, center + 1, right);
        merge(A, tmpA, left, center + 1, right);
    }
}
```




# Mergesort

```
void merge(int* A, int* tmpA, int lPos, int rPos, int rEnd) {
    int lEnd = rPos - 1;
    int tmpPos = lPos;
    int nElements = rEnd - lPos + 1;

    // COMPARAR O 1o ELEMENTO DE CADA METADE
    // E COPIAR ORDENADAMENTE PARA O ARRAY TEMPORÁRIO

    while (lPos <= lEnd && rPos <= rEnd) {
        if (A[lPos] <= A[rPos])
            tmpA[tmpPos++] = A[lPos++];
        else
            tmpA[tmpPos++] = A[rPos++];
    }
}
```



# Mergesort

```
// SOBRA, PELO MENOS, 1 ELEMENTO NUMA DAS METADES

while (lPos <= lEnd) { ...
}

while (rPos < rEnd) { ...
}

// COPIAR DE VOLTA PARA O ARRAY ORIGINAL

for (int i = 0; i < nElements; i++, rEnd--) {
    A[rEnd] = tmpA[rEnd];
}
}
```

# Eficiência


- Todas as **comparações** são feitas pela função de fusão
- $C_{\text{merge}}(n)$  : nº de **comparações** efetuadas para fundir 2 sub-arrays ordenados, usando um **array auxiliar**
- Caso particular :  $n = 2^k$

$$C(1) = 0$$


$$C(n) = 2 \times C(n / 2) + C_{\text{merge}}(n)$$

- $C_{\text{merge}}(n) = ?$

# Eficiência – $C_{\text{merge}}(n)$ – Melhor Caso


- Todos elementos de um dos sub-arrays são copiados primeiro
- Apenas  $n / 2$  comparações para fazer isso !!
- $C(n) = 2 \times C(n / 2) + n / 2$
- Teorema Mestre :  $\Theta(n \log n)$
- Construir um exemplo ! 

# Eficiência – $C_{\text{merge}}(n)$ – **Pior Caso**

- Os elementos de um dos sub-arrays são copiados de modo intercalado, um a um !
- Necessárias  **$(n - 1)$  comparações !**
- $C(n) = 2 \times C(n / 2) + (n - 1)$
- Teorema Mestre :  **$\Theta(n \log n)$**
- Construir um exemplo ! 



# Eficiência – $C_{\text{merge}}(n)$ – Caso Médio

- Podemos assumir que ocorre o nº médio de comparações
- $\sim 3 \times n / 4$  comparações
- $C(n) = 2 \times C(n / 2) + 3 \times n / 4$
- Teorema Mestre :  $\Theta(n \log n)$
- Construir um exemplo ! 

# Números de Fibonacci

# Números de Fibonacci – Recorrência

$$F(0) = 0 ; F(1) = 1$$

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

- $F(5) = ?$        $\rightarrow$  Número de chamadas recursivas ?
- Árvore das chamadas recursivas ?
- Há chamadas recursivas **repetidas** ?
- Ordem de complexidade ?       $\rightarrow$  Contar as adições

# Números de Fibonacci

```
unsigned int
fibonacciRec( unsigned int n )
{
    if( n == 0 )
    {
        return 0;
    }

    if( n == 1 )
    {
        return 1;
    }

    N_SOMAS++;

    return fibonacciRec( n - 1 ) + fibonacciRec( n - 2 );
}
```

# Tarefa – Função recursiva

- Implemente a **função recursiva** para calcular  $F(n)$
- Qual é o **maior número de Fibonacci** que consegue obter --- **em tempo útil** --- no seu computador ?
  - $F(20) = ?$
  - $F(30) = ?$
  - $F(35) = ?$
  - $F(40) = ?$
  - ...

# Número de adições ?

- $A(0) = 0 ; A(1) = 0$
- $A(i) = 1 + A(i - 1) + A(i - 2) ; i = 2, 3, 4, \dots$
- **Expressão ?**
- Matemática Discreta : solução da **equação de recorrência**
- MAS, podemos obter a ordem de complexidade analisando a **tabela...**

# Número de adições

- Como cresce  $F(n)$  ?
- Como cresce  $A(n)$  ?
- Da tabela obtemos

$$A(n) = F(n+1) - 1$$

- Crescimento **Exponential** !!
  - Porquê ?

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

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

# Programação Dinâmica



# Programação Dinâmica

- Estratégia algorítmica genérica
- Aplicável a
  - Cálculo de **recorrências**
  - Resolução de **problemas de otimização combinatória**
- Ideia : **armazenar e reutilizar resultados “anteriores”** usando um **array local**
  - Array 1D / 2D / ...

# Recursividade – Top-Down

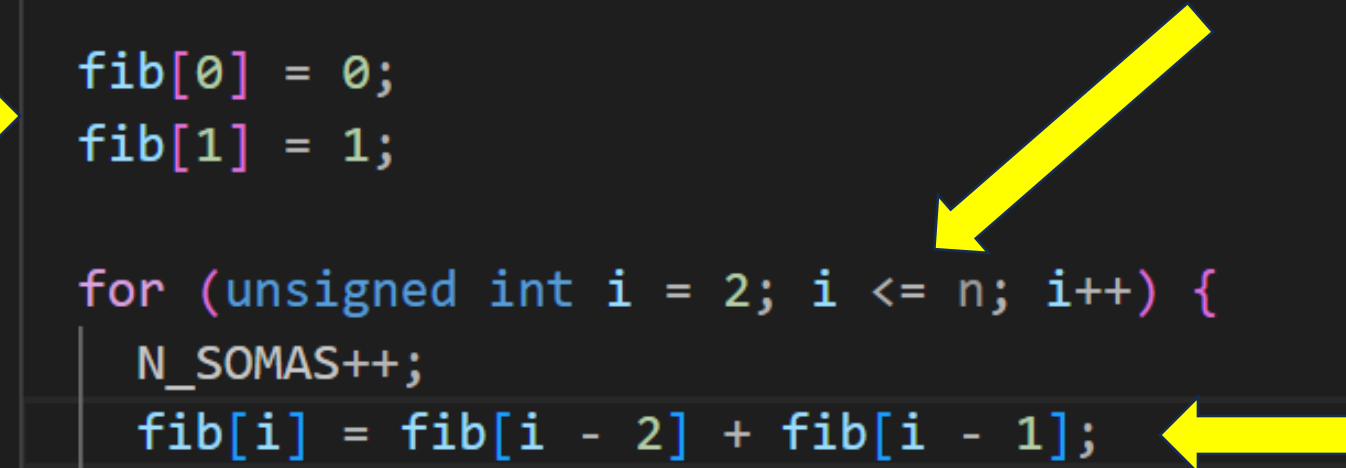
- Explorar a relação entre
  - A solução de uma dada instância de um problema
  - As soluções de **instâncias mais pequenas / mais simples** do mesmo problema
- Estabelecer uma **recorrência** !
- Decomposição em **sub-problemas mais pequenos / mais simples**
  - Argumentos ?
- Identificar as **instâncias mais pequenas / mais simples / triviais**
  - Casos de base

# Programação Dinâmica – Bottom-up

- Usar a recorrência : MAS proceder **bottom-up** !
- **Começar** com as instâncias mais pequenas / mais simples / triviais
- Determinar **resultados intermédios** a partir das instâncias anteriores
- Atingir o **resultado final**
- Vantagem : **não se repetem** quaisquer **cálculos intermédios**

# Números de Fibonacci – Prog. Dinâmica

```
long unsigned int fibonacci(unsigned int n) {  
    long unsigned int fib[n + 1];  
  
    fib[0] = 0;  
    fib[1] = 1;  
  
    for (unsigned int i = 2; i <= n; i++) {  
        N_SOMAS++;  
        fib[i] = fib[i - 2] + fib[i - 1];  
    }  
  
    return fib[n];  
}
```



# Recursão vs Prog. Din.

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

# Tarefa – Versão de Programação Dinâmica

- Implemente e teste o **algoritmo de Programação Dinâmica** para calcular  $F(n)$
- Como é preenchido o array ?
- Qual é a **ordem de complexidade** do algoritmo ?

# Memoization

# Memoization

- Resultados de uma função são **memorizados** para uso futuro
- I.e., **evita-se** o cálculo de resultados (intermédios ou finais) obtidos no processamento de inputs anteriores
- Usar **um array global** / **uma cache** para armazenar os resultados calculados
  - Como inicializar ?
- **Recursividade + Programação Dinâmica**



# Memoization

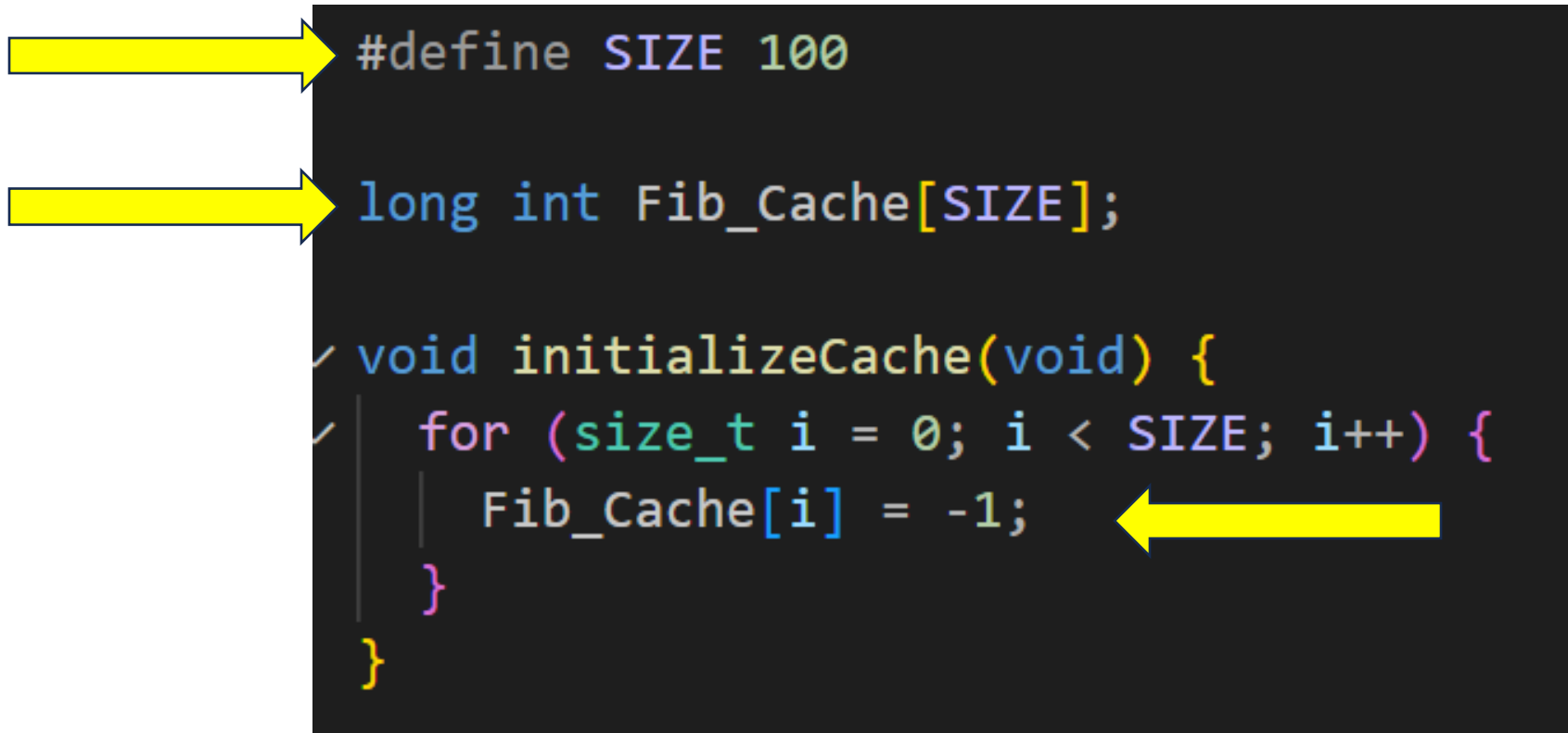
- **Inicializar** os elementos da cache com um **valor por omissão**
  - Não ocorre durante o processo de cálculo
  - Sinaliza um resultado ainda não calculado
- Sempre que se pretende um resultado para um dado input
  - **Verificar** o correspondente **elemento da cache**
  - **Se não** for o **valor por omissão**, **consultar** esse resultado
  - Caso contrário, **calculá-lo** efetuando chamada(s) recursive(s)
  - **Armazenar** o **resultado** obtido

# Números de Fibonacci – Memoization

```
#define SIZE 100

long int Fib_Cache[SIZE];

void initializeCache(void) {
    for (size_t i = 0; i < SIZE; i++) {
        Fib_Cache[i] = -1;
    }
}
```



# Números de Fibonacci – Memoization

```
long int fibonacci(unsigned int n) {  
    if (Fib_Cache[n] != -1) return Fib_Cache[n];  
  
    long int r;  
    if (n == 0)  
        r = 0;  
    else if (n == 1)  
        r = 1;  
    else {  
        N_SOMAS++;  
        r = fibonacci(n - 2) + fibonacci(n - 1);  
    }  
    Fib_Cache[n] = r;  
  
    return r;  
}
```

# Memoization

- Cálculo do valor de sucessivos números de Fibonacci
- Sucessivas chamadas à função
- Apenas mais **uma adição** para calcular o número de Fibonacci seguinte

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

# Tarefa – Versão com Memoization

- Implemente e teste o **algoritmo recursivo com “memoization”** para calcular  $F(n)$
- Como é preenchido o array global ?
- Qual é a **ordem de complexidade** do algoritmo ?

# $C(n,p)$ – O Triângulo de Pascal

# Expressão recorrente para $C(n,p)$

- $C(n,0) = 1$
- $C(n,n) = 1$
- $C(n,j) = C(n-1,j) + C(n-1,j-1)$  ;  $j = 1, 2, \dots, n - 1$
- Dois argumentos !
- $C(4,3) = ?$        $\rightarrow$  Número de chamadas recursivas ?
- Chamadas recursivas **repetidas** ?

# $C(n,p)$ – Função recursiva

```
long unsigned int combination_rec(unsigned int n, unsigned int p) {  
    if (p == 0) {  
        return 1;  
    }  
    if (n == p) {  
        return 1;  
    }  
    return combination_rec(n - 1, p - 1) + combination_rec(n - 1, p);  
}
```



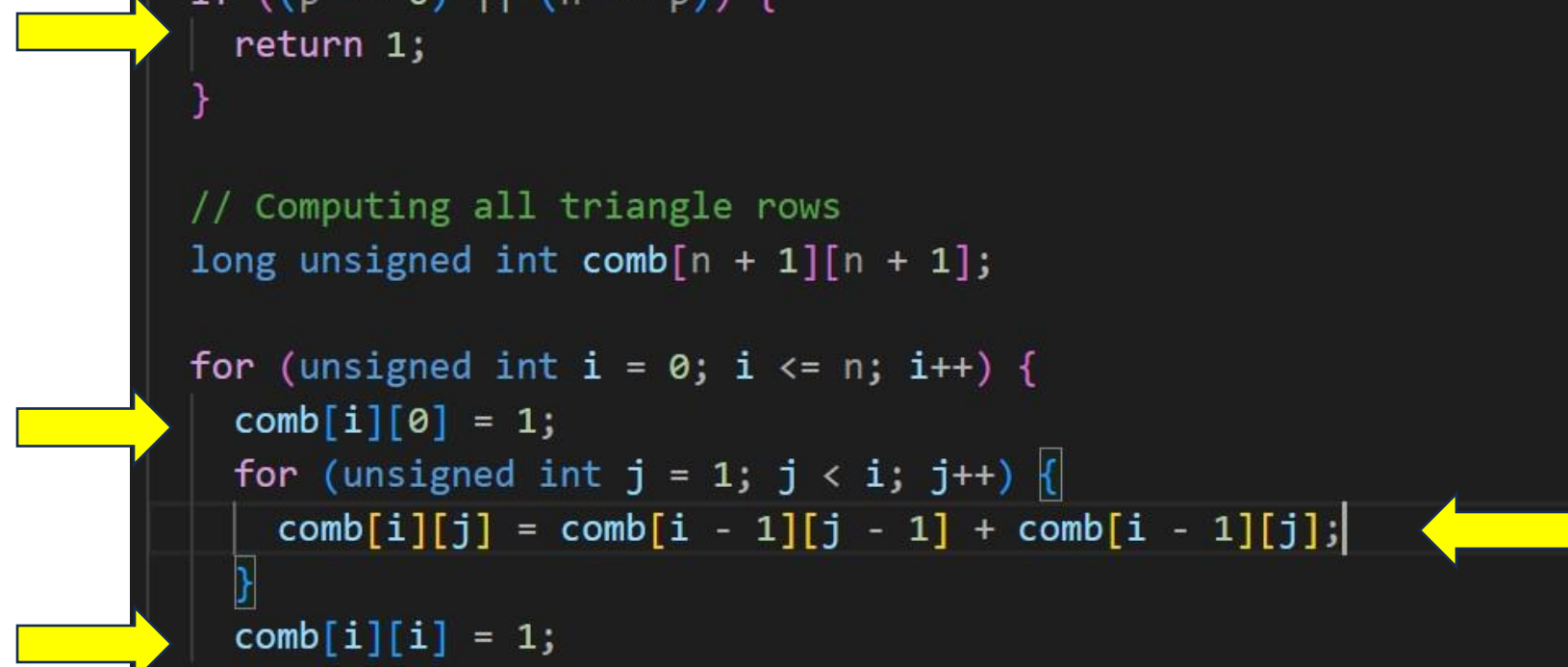
# Triângulo de Pascal

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
```

# C(n,p) – Programação Dinâmica

```
long unsigned int combination_dp(unsigned int n, unsigned int p) {  
    if ((p == 0) || (n == p)) {  
        return 1;  
    }  
  
    // Computing all triangle rows  
    long unsigned int comb[n + 1][n + 1];  
  
    for (unsigned int i = 0; i <= n; i++) {  
        comb[i][0] = 1;  
        for (unsigned int j = 1; j < i; j++) {  
            comb[i][j] = comb[i - 1][j - 1] + comb[i - 1][j];  
        }  
        comb[i][i] = 1;  
    }  
  
    return comb[n][p];  
}
```

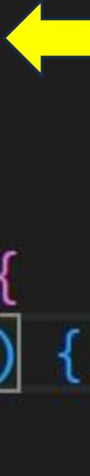


# C(n,p) – Memoization

```
#define SIZE 20

long unsigned int Comb_Cache[SIZE][SIZE];

void initialize_cache(void) {
    for (unsigned int i = 0; i < SIZE; i++) {
        for (unsigned int j = 0; j < SIZE; j++) {
            Comb_Cache[i][j] = 0;
        }
    }
}
```



# C(n,p) – Memoization

```
long unsigned int combination_memo(unsigned int n, unsigned int p) {  
    if (Comb_Cache[n][p] != 0) return Comb_Cache[n][p];  
  
    long unsigned int r;  
    if ((p == 0) || (n == p)) {  
        r = 1;  
    } else {  
        r = combination_memo(n - 1, p - 1) + combination_memo(n - 1, p);  
    }  
  
    Comb_Cache[n][p] = r;  
    return r;  
}
```

# Tarefas – $C(n,p)$

- Implemente e teste as funções para calcular  $C(n,p)$
- Função **recursiva**
- Algoritmo de **Programação Dinâmica**
- Função com **memoization**

# Números de Delannoy

## – Exercício adicional

# Números de Delannoy – $D(i,j)$

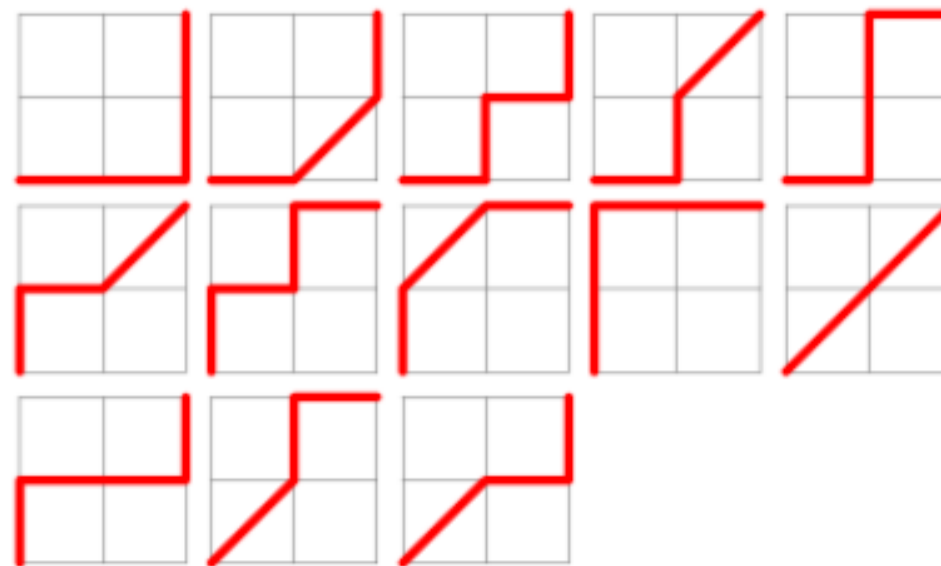
- Grelha retangular de tamanho  $(m,n)$
- Começar no canto SW :  $(0,0)$
- Só é permitido avançar nas direções **N**, **E** ou **NE**
- $D(i,j)$  = número de **caminhos distintos** de  $(0,0)$  para  $(i,j)$ 
  - **Casos triviais ?**
  - **Definição recursiva ?**

# $D(n,n)$ – Números **Centrais** de Delannoy

$D(1,1)$



$D(2,2)$

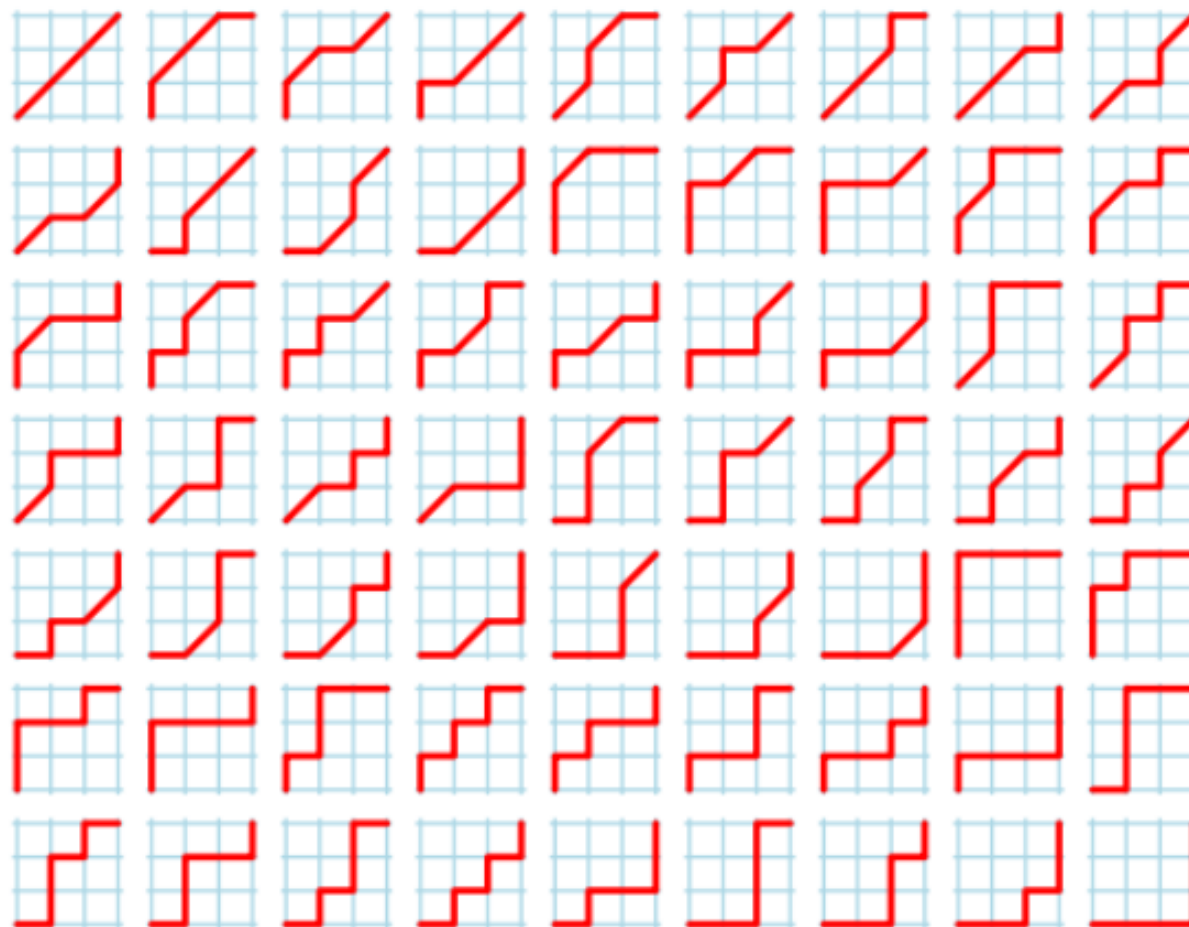


[Mathworld]



# $D(n,n)$ – Números **Centrais** de Delannoy

$D(3,3)$



[Wikipedia]

# Números de Delannoy

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

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

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

# Números de Dalennoy

Delannoy's Matrix - Recursive Function

1	1	1	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			

# Tarefa – Versão recursiva

- Implementar a **função recursiva**
- Executar para vários números centrais  $D(k,k)$
- No seu computador, qual é o **maior valor de  $k$**  que ainda permite obter o número  **$D(k,k)$  em tempo útil** ?

# Tarefa – Programação Dinâmica

- Implementar a **versão iterativa**, usando **programação dinâmica** e um **array 2D**
- Executar para vários números  $D(i,j)$
- Verificar se obtém o resultado correto
- Qual é a **ordem de complexidade** deste algoritmo ?

# Tarefa – Memoization

- Implementar a **versão recursiva com “memoization”**
- Executar para vários números  $D(i,j)$
- Verificar se obtém o resultado correto
- Qual é a **ordem de complexidade** deste algoritmo ?

# The Coin Row Problem

# O Problema da Fileira de Moedas

- Fileira de **n moedas**
- Valores inteiros  **$c_1, c_2, \dots, c_n$** 
  - Pode haver moedas repetidas
- **Objetivo** : Escolher moedas de modo a obter o **maior valor total** possível
  - **Otimização** combinatória : problema de **maximização**
- **Restrição** : **Não** podem ser escolhidas duas **moedas adjacentes**



# Valor da solução ótima – Recorrência

- Como obter uma **recorrência** para o valor da solução ótima ?
- **$V(n) = ?$**
- **Maior valor total** que se obtém de uma fileira com  **$n$  moedas**
- Casos triviais ?
- **$n$ -ésima moeda é escolhida / não é escolhida ?**

# O Problema da Fileira de Moedas

$$V(0) = 0$$

$$V(1) = c_1$$

$$V(n) = \max \left\{ \begin{array}{l} c_n + V(n - 2), \\ V(n - 1) \end{array} \right\}, \quad \text{para } n > 1$$

- $V(i)$  representa o valor da solução ótima, considerando as primeiras  $i$  moedas
- Exemplo : 5, 1, 2, 10, 6, 2
- $V(6) = ?$

# Tarefa – Versão recursiva

- Implementar a **função recursiva**
- Executar para vários exemplos
- Verificar que se obtém uma solução ótima
- Determine experimentalmente a **ordem de complexidade** da função recursiva
- No seu computador, qual é o **maior valor de n** que ainda permite **obter a solução em tempo útil** ?

# Tarefa – Programação Dinâmica

- Implementar a **versão iterativa**, usando **programação dinâmica**
- Executar para vários exemplos
- Verificar que se obtém uma solução ótima
- Qual é a **ordem de complexidade** desta versão ?

# O Problema da Fileira de Moedas

- Como determinar as **moedas** que constituem uma **solução ótima** ?
- Usar um **array adicional**, para registar se uma moeda é escolhida
  - Array **binário**
- Alternativa : fazer o “**trace back**”

# Outras estratégias

- **Procura exaustiva**
  - Gerar **todas as soluções possíveis** e escolher a(s) melhor(es)
  - Complexidade ?
- Geração de **soluções aleatórias**
  - Gerar um dado número de soluções aleatórias e escolher a(s) melhor(es)
  - **Não há garantia** de se obter a solução ótima, a menos que...
- Construir uma solução usando uma **heurística**
  - **Não há garantia** de se obter sempre a solução ótima

# Sugestões de leitura

# Sugestões de leitura

- A. Levitin, Introduction to the Design and Analysis of Algorithms, 3<sup>rd</sup> Edition, 2012
  - Capítulo 8