# 1ª Aula Prática – Programação Dinâmica – Resolução parcial

## 1. Factorial *(Factorial.h)*

a. Implementação recursiva.

```
int factorialRecurs(int n)
{
	if(n <= 1)
		return 1;
	else
		return n * factorialRecurs(n - 1);
}
```

b. Implementação iterativa.

```
int factorialDinam(int n)
{
	int result = 1;
	for (int i = 2; i <= n; i++)
		result *= i;
	return result;
}
```

c. Complexidade temporal e espacial.

| | Recursiva | Iterativa |
|---|---|---|
| T(n) | O(n) | O(n) |
| S(n) | O(n) | O(1) |

Nota: neste caso a implementação recursiva não origina trabalho repetido, pelo que a passagem a uma implementação iterativa serve essencialmente para melhorar a eficiência espacial.

## 2. Troco *(Change.h)*

a. Formalização do problema.

Dados
- $m$ – valor do troco
- $v_1, \ldots, v_n$ – valores unitários das moedas

Escolher valores das variáveis de decisão
- $x_1, \ldots, x_n$ – número de moedas de cada valor

Com o objetivo de
- minimizar $\sum_{i=1}^{n} x_i$

Sujeito à restrição
- $\sum_{i=1}^{n} x_i v_i = m$

b. Fórmulas recursivas.

$$minCoins(i,k) = \begin{cases} 0, \ \ se\ k = 0 \\ \varepsilon, \qquad se\ k > 0\ \wedge i = 0 \\ a+1, \ \ se\ \ v_i \le k \wedge a \ne \varepsilon \wedge (b = \varepsilon \vee a+1 < b) \\ b, \ \ noutros\ casos \end{cases}$$

com $\varepsilon = impossível, \ \ a = minCoins(i, k - v_i), \ \ \ b = minCoins(i-1, k)$

$$lastCoin(i,k) = \begin{cases} (nenhum), & se\ k = 0 \\ (nenhum), & se\ k > 0\ \wedge i = 0 \\ v_i, & se\ \ v_i \leq k \wedge a \neq \varepsilon \wedge (b = \varepsilon \vee a + 1 < b) \\ lastCoin(i-1,k), & noutros\ casos \end{cases}$$

Se usarmos um nº de moedas mais elevado que qualquer valor válido para representar "impossível" (por exemplo, $\varepsilon = m + 1$), podem-se remover as partes assinalados a amarelo. Em *lastCoin*, pode-se usar 0 para representar "(nenhum)".

c. Tabelas de *minCoins(i, k)* e *lastCoin(i,k)* para um exemplo com m=9.

| minCoins(i,k) | k=0 | k=1 | k=2 | k=3 | k=4 | k=5 | k=6 | k=7 | k=8 | k=9 |
|---|---|---|---|---|---|---|---|---|---|---|
| i=0 | 0 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 |
| i=1 (1 cent.) | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| i=2 (2 cent.) | 0 | 1 | 1 | 2 | 2 | 3 | 3 | 4 | 4 | 5 |
| i=3 (5 cent.) | 0 | 1 | 1 | 2 | 2 | 1 | 2 | 2 | 3 | **3** |

| lastCoin(i,k) | k=0 | k=1 | k=2 | k=3 | k=4 | k=5 | k=6 | k=7 | k=8 | k=9 |
|---|---|---|---|---|---|---|---|---|---|---|
| i=0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| i=1 (1 cent.) | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| i=2 (2 cent.) | 0 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| i=3 (5 cent.) | 0 | 1 | **2** | 2 | **2** | 5 | 5 | 5 | 5 | **5** |

d. Implementação iterativa com programação dinâmica.

```
string calcChange(int m, int numCoins, int *coinValues)
{
    int INF = m + 1;
    int minCoins[m + 1];
    int lastCoin[m + 1] = {0};

    // initialize for the base case of no coins used (i=0)
    minCoins[0] = 0; // empty solution
    for (int k = 1; k <= m; k++)
          minCoins[k] = INF; // no solution

    // iterate bottom up
    for (int i = 1; i <= numCoins; i++)
          for (int k = coinValues[i - 1]; k <= m; k++)
                if (minCoins[k - coinValues[i - 1]] + 1 < minCoins[k])
                {
                      minCoins[k] = 1 + minCoins[k - coinValues[i - 1]];
                      lastCoin[k] = coinValues[i - 1];
                }

    // produce string with the result
    if (minCoins[m] == INF)
          return "-"; // no solution
    ostringstream oss;
    for (int k = m; k > 0; k -= lastCoin[k])
          oss << lastCoin[k] << ";";
    return oss.str();
}
```

## 3. Soma de Subsequência *(Sum.h)*

a. Algoritmo baseado em programação dinâmica.

```
string calcSum(int* sequence, int size)
{
	// sum[i] and index[i] will have the sum and start index of the
	// subsequence of length i+1 with smallest sum
	int sum [size];
	int index [size];

	// experiment with different start indices
	for (int i = 0; i < size; i++) {
		int val = 0; // sum of subsequence starting at index i
		for (int j = i; j < size; j++) {
			val += sequence[j];
			if(i == 0 || val < sum[j-i]) {
				sum[j-i] = val;
				index[j-i] = i;
			}
		}
	}

	// return result as a string to compare more easily
	ostringstream oss;
	for(int i=0; i < size; i++)
		oss << sum[i] << "," << index[i] << ";";
	return oss.str();
}
```

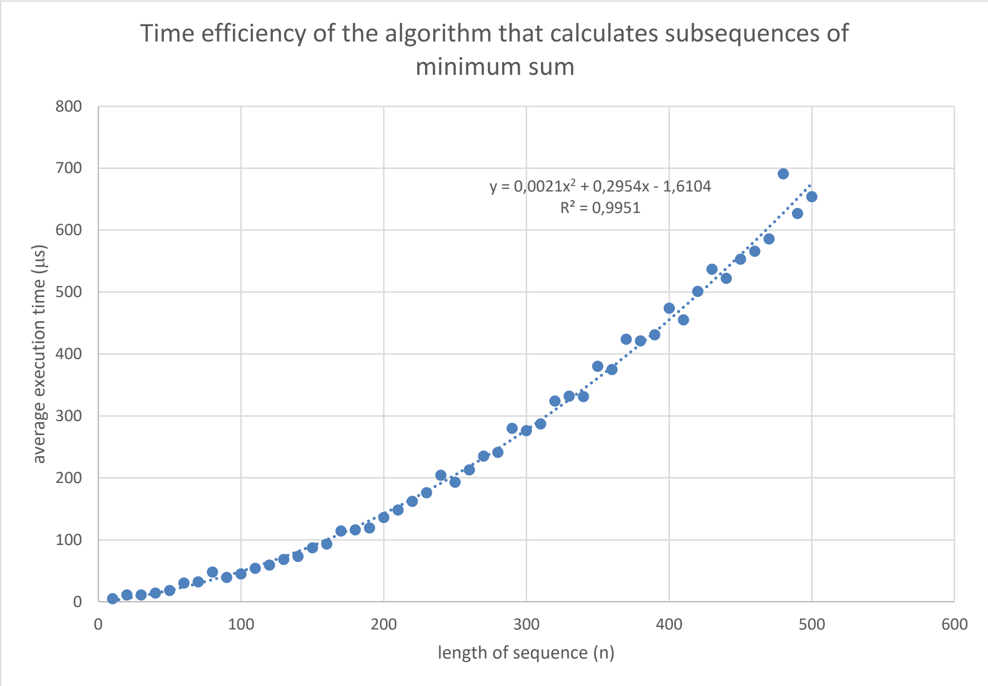
<u>Nota</u>: Qualquer que seja o algoritmo, de alguma forma tem de: (1) calcular as somas $S_{ij}$ de elementos entre índices $i$ e $j$, para todos os índices $0 \leq i \leq j < n$; (2) para cada comprimento possível ($j$-$i$+$1$), tem de escolher a soma mínima. A otimização a efetuar consiste em, de alguma forma, evitar repetir trabalho no cálculo dos $S_{ij}$ (ou seja, calcular cada $S_{ij}$ em tempo O(1)).

b. Medicação do tempo médio de execução do algoritmo para valores de $n$ crescentes.

```
void testPerformanceCalcSum()
{
	srand(time(NULL)); //generates random seed

	int seq[1+1000];

	cout << "size; time" << endl;
	for (int size = 10; size <= 500; size += 10) {
		auto start = std::chrono::high_resolution_clock::now();
		for (int k = 0; k < 1000; k++) {
			for (int i = 0; i <size; i++)
				seq[i]= rand() % (10 * size) + 1;
			string res = calcSum(seq, size);
		}
		auto finish = std::chrono::high_resolution_clock::now();
		auto elapsed = chrono::duration_cast<chrono::milliseconds>(finish -
start).count();
		cout << size << ";" << elapsed << endl;
	}
}
```




<u>Nota</u>: Os resultados experimentais são coerentes com a complexidade temporal teórica do algoritmo, que é $O(n^2)$.

## 4. Particionamentos de um conjunto *(Partitioning.h)*

a. Implementação recursiva.

```
int s_recursive(int n,int k)
{
	if(k == 1 || k == n)
		return 1;
	else //let's suppose k and n are within expected values (1 < k < n)
		return s_recursive(n-1,k-1) + k * s_recursive(n-1,k);
}

int b_recursive(int n)
{
	int sum = 0;

	for(int k = 1; k <= n; k++)
		sum += s_recursive(n,k);

	return sum;
}
```

b. Implementação iterativa.

```
int s_dynamic(int n, int k)
{
	int len = n-k+1;
	int values[len]; // a column of the S(n,k) triangle

	// initialize column for k=1 (all 1's)
	for (int i = 0; i < len; i++)
		values[i] = 1;
```

```
	// compute the next columns up to the given k (similar to C(n,k))
	for (int i = 2; i <= k; i++)
		for (int j = 1; j < len; j++)
			values[j] += i * values[j-1];

	return values[len-1];
}

int b_dynamic(int n)
{
	int values[n + 1]; // a line of the S(n,k) triangle

	// compute the lines up to the given n
	for (int i = 1; i <= n; i++) {
		values[i] = 1;
		for (int k = i - 1; k > 1; k--)
			values[k] = values[k-1] + k * values[k];
	}

	int sum = 0;
	for (int k = 1; k <= n; k++)
		sum += values[k];

	return sum;
}
```

c. Profiling.

```
VS2012 ARM Cross Tools Command Prompt

C:\Users\Jo├o\workspace_cpp\CalFp01v1>gprof -pPartitioning.cpp "Debug Gcov"/CalF
p01v1.exe
Flat profile:

Each sample counts as 0.01 seconds.
 no time accumulated

  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
  0.00      0.00     0.00       20     0.00     0.00  s_recursive(int, int)
  0.00      0.00     0.00        3     0.00     0.00  b_dynamic(int)
  0.00      0.00     0.00        3     0.00     0.00  s_dynamic(int, int)
  0.00      0.00     0.00        2     0.00     0.00  b_recursive(int)
```

<u>Nota</u>: Como apenas se correram os casos de teste, que executam muito rapidamente (em menos de 10 ms), apenas se consegue ver o nº de chamadas de cada função.