1ª Aula Prática – Programação Dinâmica

Instruções

- Faça download do ficheiro **cal_fp01.zip** da página da disciplina e descomprima-o (contém os ficheiros **Test.cpp, Defs.h, Change.h, Factorial.h, Partitioning.h, Sum.h**, um para cada exercício)
- Abra o eclipse e crie um novo projeto C++ do tipo Cute Project (File/New/C++ Project/Cute Project) com o nome CalFp01 e usando o MinGW GCC (CUTE version: CUTE headers 2.1.0)
- Inclua a biblioteca Boost
- Importe para a pasta src do projeto, os ficheiros extraídos: (Import/General/File System)
    - Aparecendo mensagem a perguntar se quer fazer Overwrite a Test.cpp diga sim.
    - Compile o projeto.
    - Execute o projeto como CUTE Test (Run As/CUTE Test). Se surgir a pergunta de qual compilador usar, escolha MinGW gdb.
- ***Deverá realizar esta ficha respeitando a ordem das alíneas.*** Poderá executar o projeto como CUTE Test quando quiser saber se a implementação que fez é suficiente para passar no teste correspondente.
- Considere implementar as várias soluções baseando-se nas indicações dos ficheiros **.h** respetivos.

Enunciado

1. Factorial *(Factorial.h)*

Suponha que se pretende calcular o factorial de um dado inteiro n (≥ 0).

a. Implemente e teste uma função que faça o cálculo de forma recursiva.
b. Implemente e teste uma função que faça o cálculo de forma iterativa com programação dinâmica.
c. Compare as duas abordagens das alíneas a e b em termos de complexidade temporal e espacial.

2. Troco *(Change.h)*

Suponha que se pretende calcular o troco no valor de m cêntimos, utilizando um número mínimo de moedas. Os valores das moedas disponíveis (por exemplo, 1, 2 e 5 cêntimos) são passados como parâmetro. Assume-se que existe um stock ilimitado de moedas de cada valor. Por exemplo, para dar troco de 9 cêntimos, usam-se duas moedas de 2 cêntimos e uma de 5 cêntimos.

a. Formalize o problema como problema de programação linear. (<u>Sugestão</u>: ver problema da mochila nos slides das aulas teóricas.)
b. Escreva em notação matemática funções recursivas $minCoins(i, k)$ e $lastCoin(i,k)$ que indicam o número mínimo de moedas e o valor da última moeda a usar para perfazer exatamente o montante k ($0 \leq k \leq m$), usando apenas as primeiras i moedas ($0 \leq i \leq n$, em que n é o número de moedas diferentes disponíveis). Usar um símbolo ou valor especial para o caso de função não definida.

c. Calcular a tabela com os valores de *minCoins(i, k)* e *lastCoin(i,k)* para o exemplo do troco de 9 cêntimos com moedas de 1, 2 e 5 cêntimos.
d. Conceba um algoritmo utilizando programação dinâmica que determine a solução ótima para um dado *m*, retornando as moedas utilizadas. Deve calcular os valores de *minCoins* e *lastCoin* (como *arrays*), para valores de *i* e *k* crescentes, memorizando apenas valores para o último valor de *i*.

3. Soma de Subsequência *(Sum.h)*

Dada uma sequência de *n* números inteiros ($n > 0$), para cada valor de *m* de 1 a *n*, pretende-se determinar o índice *i* (a começar em 0) da subsequência de *m* elementos contíguos de soma (*s*) mínima.

Por exemplo, a sequência (4,7,2,8,1), com n=5, tem as seguintes subsequências de soma mínima:

Subs. de tamanho m=1 (1): s=1, i=4
Subs. de tamanho m=2 (7,2): s=9, i=1 (se existe mais do que uma possibilidade, retorna-se a primeira)
Subs. de tamanho m=3 (2,8,1): s=11, i=2
Subs. de tamanho m=4 (7,2,8,1): s=18, i=1
Subs. de tamanho m=5 (4,7,2,8,1): s=22, i=0

a. Conceba um algoritmo utilizando programação dinâmica que determine a solução ótima para cada *m*, retornando *i* e *s* para cada caso. Testar para o exemplo acima.
b. Produzir um gráfico com o tempo médio de execução do algoritmo para valores de *n* crescentes 10, 20, …, 500. Para cada valor de *n*, gerar 1000 sequências aleatórias de nºs inteiros entre 1 e $10 \times n$ (admitindo repetições) e medir o tempo total. <u>Sugestão</u>: gerar um ficheiro em formato CSV e construir um gráfico a partir de Excel ou outra ferramenta similar; basear-se no código abaixo para medir o tempo decorrido em milissegundos (μs).

```
#include <chrono>
...
auto start = std::chrono::high_resolution_clock::now();
...
auto finish = std::chrono::high_resolution_clock::now();
auto mili = chrono::duration_cast<chrono::milliseconds>(finish – start).count();
```

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

O número de maneiras de partir um conjunto de *n* elementos $(n > 0)$ em *k* subconjuntos disjuntos não vazios $(1 < k < n)$ é dado pelo número de Stirling de segunda ordem, *S(n,k)*, o qual pode ser calculado pela fórmula de recorrência:

$$S(n,k) = S(n-1,k-1) + k\,S(n-1,k), \text{ se } 1 < k < n$$
$$S(n,k) = 1, \text{ se } k=1 \text{ ou } k=n$$

Por sua vez, o número de maneiras de partir um conjunto de *n* elementos $(n > 0)$ em subconjuntos disjuntos não vazios é dado pelo n-ésimo número de Bell, denotado *B(n)*, o qual pode ser calculado pela fórmula:

$$B(n) = \sum_{k=1}^{n} S(n,k)$$

Por exemplo, o conjunto {a, b, c} pode ser partido de 5 maneiras diferentes:

{a, b, c}
{a, b}, {c}
{a, c}, {b}
{b, c}, {a}
{a}, {b}, {c}

Neste caso tem-se B(3) = S(3,1) + S(3,2) + S(3,3) = 1 + (S(2,1)+2S(2,2)) + 1 = 1 + 3 +1 = 5.

B(n) cresce rapidamente. Por exemplo, B(15) =1382958545.

a. Implemente as funções S(n,k) e B(n) utilizando uma abordagem recursiva, com base nas definições.
b. Implemente as funções S(n,k) e B(n) usando programação dinâmica, com base no exemplo do cálculo de ${}^{n}C_{k}$ apresentado nas aulas teóricas. Qual é a eficiência temporal e espacial dessa rotina?
c. Compare o desempenho das versões recursiva e com programação dinâmica usando o *gnu profiler* (ver instruções nos slides da aula teórica).