

3ª Aula Prática

Pesquisa Sequencial, Pesquisa Binária e Variantes

Instruções

- Faça download do ficheiro *aed2223_p03.zip* da página da disciplina e descomprima-o (contém a pasta *lib*, a pasta *Tests* com os ficheiros *funWithSearch.cpp*, *funWithSearch.h*, *Timer.cpp*, *Timer.h* e *tests.cpp*, as pastas *search*, *lowerBound*, *countRange*, *partitions*, *closestSums* e *closestSums*, e os ficheiros *CMakeLists* e *main.cpp*)
- No CLion, abra um projeto, seleccionando a pasta que contém os ficheiros do ponto anterior
- Se não conseguir compilar, efetuar “Load CMake Project” sobre o ficheiro *CMakeLists.txt*
- Faça a sua implementação no ficheiro *funWithSearch.cpp*
- Note que apenas os primeiros testes de *search* (o 1º exercício) estão descomentados; deverá ir descomentando os outros testes à medida que vai fazendo os exercícios.

1. **Pesquisa Elementar.** (o objectivo deste exercício é testar uma implementação base de pesquisa sequencial e de pesquisa binária e verificar os seus tempos de execução)



Função a implementar:

```
int FunWithSearch::search(const vector<int> & v, int key)
```

Complexidade temporal esperada: $\Theta(\log n)$ onde n é o tamanho do vector.

Esta função deve devolver a posição (índice) de *key* no vector *v* ou -1 caso *key* não exista no vector. Pode assumir que o array está **ordenado de forma crescente** e que **não existem números repetidos**.

Exemplo de chamada e output esperado:

```
cout << FunWithSearch::search({2,3,5,7,8}, 2) << endl;
cout << FunWithSearch::search({2,3,5,7,8}, 8) << endl;
cout << FunWithSearch::search({2,3,5,7,8}, 7) << endl;
cout << FunWithSearch::search({2,3,5,7,8}, 1) << endl;
cout << FunWithSearch::search({2,3,5,7,8}, 10) << endl;
cout << FunWithSearch::search({2,3,5,7,8}, 6) << endl;
```

```
0
4
3
-1
-1
-1
```

Explicação: 2 aparece no início, na posição 0; 8 aparece no final, na posição 4; 7 aparece na posição 3; os números 1, 10 e 6 não aparecem.

a) Uma primeira solução com “pesquisa sequencial” em tempo $\Theta(n)$

Comece por implementar uma simples **pesquisa sequencial** (*slide 6 do capítulo 3 – Pesquisa*) que passa por todos os números do vector tentando encontrar a *key* em tempo linear. Copie o código seguinte para a função, compile e execute para ver o tempo que demora em cada teste.

[note em tests.cpp como os testes são feitos via ficheiros e como em cada um são feitas várias chamadas à função respetiva]

```
for (unsigned i=0; i<v.size(); i++)
    if (v[i] == key)
        return i; // found key
return -1; // not found
```

b) Melhorando a solução para $\Theta(\log n)$

Intuitivamente, deve ser “aproveitável” o facto do array estar ordenado. E para isso vamos usar... **pesquisa binária** (*slides 9 a 12 slides do capítulo 3 – Pesquisa*).

```
int low = 0, high = (int)v.size() - 1;
while (low <= high) {
    int middle = low + (high - low) / 2;
    if (key < v[middle])    high = middle - 1;
    else if (key > v[middle]) low = middle + 1;
    else return middle; // found key
}
return -1; // not found
```

Implemente esta nova solução e veja o que acontece em todos os testes. Quanto tempo demorou agora? Submeta a solução no [Mooshak](#) e veja como assim obtém **Accepted**.

2. Limite Inferior.

Neste exercício vamos implementar uma primitiva muito útil que pode ser feita com pesquisa binária.

Função a implementar:

```
int FunWithSearch::lowerBound(const vector<int> &v, int key)
```

Complexidade temporal esperada: $\Theta(\log n)$ onde n é o tamanho do vector.

Esta função deve devolver a posição (índice) de primeiro elemento do vetor v que é maior ou igual a key ou -1 caso todos os elementos do vector sejam inferiores a key . Pode assumir que o array está **ordenado de forma crescente**. Note que **podem existir números repetidos**.

Exemplo de chamada e output esperado:

```
cout << FunWithSearch::lowerBound({2,2,2,3,5,5,8,8,8,8}, 2) << endl;
cout << FunWithSearch::lowerBound({2,2,2,3,5,5,8,8,8,8}, 8) << endl;
cout << FunWithSearch::lowerBound({2,2,2,3,5,5,8,8,8,8}, 5) << endl;
cout << FunWithSearch::lowerBound({2,2,2,3,5,5,8,8,8,8}, 1) << endl;
cout << FunWithSearch::lowerBound({2,2,2,3,5,5,8,8,8,8}, 4) << endl;
cout << FunWithSearch::lowerBound({2,2,2,3,5,5,8,8,8,8}, 9) << endl;
```

```
0
6
4
0
4
-1
```

Explicação: o primeiro número ≥ 2 é $v[0]$ (2); o primeiro número ≥ 8 é $v[6]$ (8);

o primeiro número ≥ 5 é $v[4]$ (5); o primeiro número ≥ 1 é $v[0]$ (2);

o primeiro número ≥ 4 é $v[4]$ (5); não existe nenhum número ≥ 9

Para este problema, uma solução "bruta" que faça pesquisa sequencial por todos os elementos não passa não tempo, e precisamos de uma solução melhor... (*não se esqueça de submeter no [Mooshak](#)*)

Dicas (não leia se não quiser spoilers):

- Este exercício pode ser feito com pesquisa binária e foi explicado nas teóricas.

(*slides 14 e 15 slides do capítulo 3 – Pesquisa*).

- Procure fazer implementando manualmente para perceber como funciona esta variação de pesquisa binária. Depois de ter **Accepted** pode experimentar resolver novamente mas agora chamando diretamente a função `lower_bound` disponível em `<algorithm>` (note que a função devolve um iterador, mas este é de *RandomAccess* e pode facilmente descobrir a posição (ver por exemplo o *slide 32*))

3. Intervalos.

Função a implementar:

```
int FunWithSearch::countRange(const vector<int> & v, int a, int b)
```

Complexidade temporal esperada: $\Theta(\log n)$ onde n é o tamanho do vector.

Esta função deve devolver a quantidade de elementos do vector v que estão no intervalo de valores $[a, b]$. Pode assumir que o array está **ordenado de forma crescente** e **podem existir números repetidos**.

Exemplo de chamada e output esperado:

```
cout << FunWithSearch::countRange({2,2,2,3,5,5,8,8,8,8}, 1, 9) << endl;
cout << FunWithSearch::countRange({2,2,2,3,5,5,8,8,8,8}, 3, 5) << endl;
cout << FunWithSearch::countRange({2,2,2,3,5,5,8,8,8,8}, 4, 7) << endl;
cout << FunWithSearch::countRange({2,2,2,3,5,5,8,8,8,8}, 6, 7) << endl;
cout << FunWithSearch::countRange({2,2,2,3,5,5,8,8,8,8}, 1, 1) << endl;
```

```
10
3
2
0
0
```

Explicação: todos os 10 números estão em $[1, 9]$; 3 números estão em $[3, 5]$ ($\{3, 5, 5\}$);

2 números estão em $[4, 7]$ ($\{5, 5\}$); nenhum número está em $[6, 7]$ ou em $[1, 1]$

Neste exercício **poderá ser muito útil chamar a função desenvolvida no exercício anterior**.

Dicas (não leia se não quiser spoilers):

- Se chamar `lowerBound(v, a)` que posição obtém?
- Uma chamada a `lowerBound(v, b)` obtém a primeira ocorrência de b , se existir. Como pode usar esta função para obter o elemento *exactamente à frente* de todas as ocorrências de elementos $\leq b$?
- Tendo a primeira posição que pertence ao intervalo, e a posição imediatamente depois do intervalo, que conta falta fazer para obter o resultado?
- Com duas pesquisas binárias, cada uma demorando $\Theta(\log n)$, qual fica a complexidade da função?

4. Viagem de mochila às costas.

Neste exercício vamos implementar a solução para um problema que foi explicado nas teóricas.

(slides 16 a 21 slides do capítulo 3 – Pesquisa)

O Aniceto e os amigos resolveram fazer uma viagem até aos Alpes, onde vão percorrer um belíssimo trilho de montanha. Como o trilho é muito longo, eles resolveram levar as mochilas às costas e acampar várias noites no caminho. Para o Aniceto, uma das melhores partes de toda a experiência é o planeamento de toda a viagem e ele quer ter a certeza de escolher da melhor forma possível.

Os amigos já sabem quais os sítios onde é possível montar as tendas para passar a noite sabem também qual a distância em kms entre os possíveis locais sucessivos de acampamento. O objectivo do Aniceto dividir o percurso em vários dias, de modo a **minimizar a distância que têm de andar num único dia**. Imagina por exemplo que eles querem dividir em quatro dias as seguintes distâncias consecutivas:

7 9 3 8 2 2 9 4 3 4 7 9 9

Uma hipótese seria dividir percurso nas seguintes quatro partes:

7 9 3 | 8 2 2 | 9 4 3 | 4 7 9 9

Neste caso, eles andariam **19km** no primeiro dia (7+9+3), **12km** no segundo dia (8+2+2), **16km** no terceiro dia (9+4+3) e **29km** no último dia (4+7+9+9). O "custo" deste caminho seria de **29km**, que é a maior distância num único dia.



Uma alternativa melhor seria a seguinte:

7 9 | 3 8 2 2 | 9 4 3 4 | 7 9 9

Agora eles andariam 16km, 15km, 20km e 25km em cada dia, com o custo a ser de 25km (a maior distância).

Esta não é contudo a maneira ótima para quatro dias de viagem... Ainda para mais o Aniceto gostava de saber como dividir o percurso se ao invés de quatro, quisesse três ou cinco dias de viagem. Será que podes ajudá-lo?

4.a - isPossible

Vamos começar por implementar uma função auxiliar que será muito útil.

(não se esqueça de submeter no Mooshak)

Função a implementar:

```
bool FunWithSearch::isPossible(const vector<int> & v, int x, int k)
```

Complexidade temporal esperada: $\Theta(n)$ onde n é o tamanho do vector.

Esta função deve devolver *true* se é possível partir o vector v em k partições (subsequências de números contíguos) onde a soma da maior partição é $\leq x$ ou *false* caso tal não seja possível.

Exemplo de chamada e output esperado:

```
cout << FunWithSearch::isPossible({7,9,3,8,2,2,9,4,3,4,7,9,9}, 21, 4) << endl;
cout << FunWithSearch::isPossible({7,9,3,8,2,2,9,4,3,4,7,9,9}, 20, 4) << endl;
cout << FunWithSearch::isPossible({7,9,3,8,2,2,9,4,3,4,7,9,9}, 22, 4) << endl;
cout << FunWithSearch::isPossible({7,9,3,8,2,2,9,4,3,4,7,9,9}, 27, 3) << endl;
cout << FunWithSearch::isPossible({7,9,3,8,2,2,9,4,3,4,7,9,9}, 26, 3) << endl;
cout << FunWithSearch::isPossible({7,9,3,8,2,2,9,4,3,4,7,9,9}, 18, 5) << endl;
cout << FunWithSearch::isPossible({7,9,3,8,2,2,9,4,3,4,7,9,9}, 17, 5) << endl;
```

```
1
0
1
1
0
1
0
```

Explicação: o vetor corresponde ao dado no enunciado.

≤ 21 pode ser obtido com 4 partições por exemplo do seguinte modo: 7 9 3 | 8 2 2 9 | 4 3 4 7 | 9 9

com 4 partições é impossível que todas tenham soma inferior ou igual a 20

a solução de ≤ 22 pode ser a mesma de ≤ 21

≤ 27 pode ser obtido com 3 partições por exemplo do seguinte modo: 7 9 3 8 | 2 2 9 4 3 4 | 7 9 9

com 3 partições é impossível que todas tenham soma inferior ou igual a 26

≤ 18 pode ser obtido com 5 partições por exemplo do seguinte modo: 7 9 | 3 8 2 2 | 9 4 3 | 4 7 | 9 9

com 5 partições é impossível que todas tenham soma inferior ou igual a 17

Dicas (não leia se não quiser spoilers):

- Este exercício pode ser feito com um algoritmo *greedy* cuja ideia principal é ir **extendendo a partição atual enquanto a sua soma for viável** ($\leq x$); se ultrapassar x , então começa-se nova partição; no final basta verificar qual o número de partições mínimo e pode ver-se se k chegavam (*slide 18*).

4.a - partitions

Vamos então à função “principal” (não se esqueça de submeter no Mooshak)

Função a implementar:

```
int FunWithSearch::partitions(const vector<int> & v, int k)
```

Complexidade temporal esperada:

$\Theta(n \log s)$ onde n é o tamanho do vector e s é a soma de todos os elementos do vector.

Esta função deve devolver a resposta ao problema dado, ou seja, deve descobrir a melhor maneira de partir o vector v em k partições de números contíguos, de tal maneira que minimize a soma da pior partição, como explicado. Deve devolver precisamente qual é essa melhor soma de uma pior partição.

Exemplo de chamada e output esperado:

```
cout << FunWithSearch::partitions({7,9,3,8,2,2,9,4,3,4,7,9,9}, 4) << endl;
cout << FunWithSearch::partitions({7,9,3,8,2,2,9,4,3,4,7,9,9}, 3) << endl;
cout << FunWithSearch::partitions({7,9,3,8,2,2,9,4,3,4,7,9,9}, 5) << endl;

21
27
18
```

Explicação: o vetor corresponde ao dado no enunciado.

Com $K=4$ dias, a resposta é 21 e uma partição possível com esse custo é 7 9 3 | 8 2 2 9 | 4 3 4 7 | 9 9

Com $K=3$ dias, a resposta é 27 e uma partição possível com esse custo é 7 9 3 8 | 2 2 9 4 3 4 | 7 9 9

Com $K=5$ dias, a resposta é 18 e uma partição possível com esse custo é 7 9 | 3 8 2 2 | 9 4 3 | 4 7 | 9 9

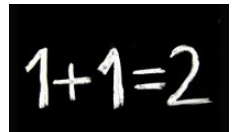
Dicas (não leia se não quiser spoilers):

- Fazer **pesquisa binária da solução** no intervalo $[l, s]$, onde s é a soma de todos os números do vector
- O espaço de procura é de *não não ... sim sim ... sim* como explicado nas aulas e pode verificar se a resposta é *sim...* usando a função do exercício anterior (*slides 19 a 21*)

Exercícios Extra

Os exercícios seguintes permitem-lhe consolidar mais um pouco os seus conhecimentos nesta parte da matéria.

5. Somas mais próximas. (não se esqueça de submeter no *Mooshak*)



Função a implementar:

```
vector<int> FunWithSearch::closestSums(const vector<int> & v, const vector<int> & p)
```

Complexidade temporal esperada:

$\Theta(n^2 \log n^2 + m \log n^2)$ onde n é o tamanho do vector v e m o tamanho do vector p

Para cada um dos elementos p_i do vector p , descobrir qual a soma de dois números diferentes do vector v que está mais próxima de p_i .

Esta função deve devolver um vector a onde a_i é a resposta para a pergunta p_i . Se existirem duas somas à mesma distância, deve ser imprimida a menor dessas somas. É garantido que todos os números de v são distintos.

Exemplo de chamada e output esperado:

```
vector<int> answers = FunWithSearch::closestSums({12,3,17,5,34,33}, {1,51,41,21});
for (int i : answers) cout << i << " " << endl;
cout << endl;

8
51
39
20
```

Explicação: Neste caso temos $S = \{3, 5, 12, 17, 33, 34\}$ e 4 perguntas.

Para a pergunta 1, a resposta é 8 (3+5), a soma de um par mais próxima de 1.

Para a pergunta 51, a resposta é 51 (17+34).

Para a pergunta 41, a resposta é 39 (5+34).

Finalmente, para a pergunta 21, a resposta é 20 (3+17) e 22 (5+17), ambos à mesma distância de 21.

Dicas (não leia se não quiser spoilers):

- Vamos começar por calcular todos as somas possíveis de pares: $v[i] + v[j], 0 \leq i < j < n$
 - . Exactamente quantos pares existem? (use o seu conhecimento sobre somatórios e progressões aritméticas obtido no capítulo anterior de matéria). Qual a sua ordem de grandeza em termos assintóticos?
 - . Implemente dois ciclos para guardar estes pares num novo vector *somas* e confirme que a quantidade de pares é a que calculou.
- Para cada uma das p perguntas vamos querer agora pesquisar no vector *pares* procurando a soma mais próxima
 - . Uma pesquisa sequencial é lenta de mais e não passaria no tempo
 - . O array de somas é igual para todas as perguntas pelo que podemos pré-processar para depois melhorar a pesquisa. Sendo assim, podemos precisamente... ordená-lo! Use o `sort` de C++.
 - . Depois de ter ordenado, podemos aplicar... **pesquisa binária**! É no entanto necessário cuidado porque o número que procuramos pode não estar no array. Como podemos usar o *lowerBound* para nos ajudar?
 - . A soma mais próxima pode ser menor... ou maior que o número da pergunta
- No final, o esquema geral do algoritmo para este problema é o seguinte:
 - . Calcular a soma de todos os pares - em tempo $\Theta(n^2)$.
 - . Ordenar as somas dos pares - em tempo $\Theta(n^2 \log n^2)$.
 - . Para cada uma das p perguntas, fazer uma pesquisa binária - em tempo $\Theta(\log n^2)$.

6. Um jogo de strings.

A pequena Maria tem um novo passatempo: ela gosta de retirar letras de uma palavra para obter outra palavra. É no entanto complicado para ela e o seu irmão João quer sempre ajudá-la.



O João dá à Maria uma palavra **A** para obter a palavra **B** a partir dela. A Maria remove as letras de **A** numa certa ordem que é indicada por uma permutação **P** dos índices da palavra **A**. Por exemplo, se palavra **A**="MARIA" e permutação **P**= $\{4,1,3,2,5\}$, então a Maria as remoções de letras fazem a seguinte sequência a palavras: MARIA → MAR**I**A → **M**AR**I**A → **M**AR**I**A → **M**AR**I**A → **M**AR**I**A.

O João conhece a permutação. O seu objetivo é parar a sua irmã em alguma fase e continuar ele próprio a retirar as letras até chegar à palavra **B**. No entanto, como a Maria gosta tanto deste passatempo, ele quer parar a irmã o mais tarde possível. A tua tarefa é determinar quantas letras pode a Maria remover até ser parada pelo João.

Função a implementar:

```
int FunWithSearch::stringGame(const string & a, const string & b, const vector<int> & p)
```

Complexidade temporal esperada:

$\Theta(n \log n)$ onde n é o tamanho da palavra **A**

A função deve devolver o maior número de letras que a Maria pode retirar de **A** (a) pela ordem indicada pela permutação **P** (p) de tal modo que ainda seja possível no final obter a palavra **B** (b). É garantido que a palavra **B** pode ser obtida a partir de **A** e que **P** contém uma permutação válida (ou seja, tem todos os números entre 1 e $|A|$ uma única vez). As palavras são sempre constituídas por letras minúsculas.

Exemplo de chamada e output esperado:

```
cout << FunWithSearch::stringGame("ababcba", "abb", {5,3,4,1,7,6,2}) << endl;
cout << FunWithSearch::stringGame("bbbabb", "bb", {1,6,3,4,2,5}) << endl;
```

3
4

Explicação: No primeiro caso temos ababcba → abab**e**ba → abab**e**ba → abab**e**ba

Aqui a Maria não pode continuar pois seria impossível obter abb a partir de abab**e**ba

No segundo caso temos bbbabb → **b**bbabb → **b**bbabb → **b**bbabb → **b**bbabb

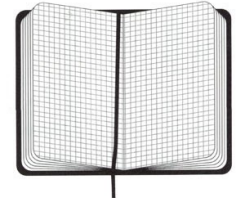
Aqui a Maria não pode continuar pois seria impossível obter bb a partir de **b**bbabb

Dicas (não leia se não quiser spoilers):

- Consegue fazer uma função que diga se é possível a Maria remover k letras? Isto pode ser feito em tempo $\Theta(n)$ numa única passagem “ao mesmo tempo” por **A** e **B**
(pode acrescentar funções diretamente no ficheiro `funWithSearch.cpp` sem as colocar no `.h`, pois neste problema apenas poderá submeter um ficheiro no Mooshak)
- Tendo a função anterior a funcionar, o espaço de procura de k é de não não ... sim sim ... sim e pode usar **pesquisa binária na resposta**. E quais são os k possíveis?

Exercício de Desafio

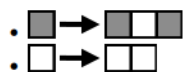
(exercício substancialmente mais difícil para alunos que querem ter desafios adicionais com problemas algoritmicamente mais complexos)



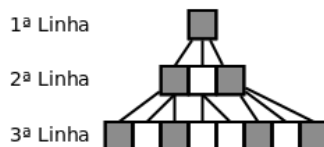
8. Caderno Quadriculado.

(exercício baseado num problema da fase de seleção das Olimpíadas Nacionais de Informática 2016)

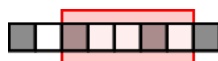
Como sabes da aula anterior, a Sara adora o seu caderno quadriculado e arranja todo o tipo de maneiras de passar tempo com ele. Desta vez resolveu começar a pintar com o seu lápis algumas quadrículas. A partir da maneira como preenche uma linha do caderno, ela faz as seguintes transformações a cada quadrícula:



A Sara começa por pintar na primeira linha apenas uma quadrícula. A partir daí pinta nas linhas sucessivas usando as regras indicadas. A figura seguinte ilustra a maneira como ficariam as 3 primeiras linhas do caderno:



A Sara achou que o caderno estava a ficar com um padrão muito bonito! Como adora contar, resolveu seleccionar uma parte de uma das linhas e contar quantas quadrículas estão pintadas. Por exemplo, entre a 3ª e a 7ª posição da 3ª linha existem duas quadrículas pintadas:



A Sara rapidamente percebeu que ia dar muito trabalho contar quadrículas para as linhas seguintes e precisa da tua ajuda!

Função a implementar:

```
long long FunWithSearch::rules(int k, long long a, long long b)
```

Sabendo que a Sara usa as regras atrás descritas começando com uma única quadrícula pintada na primeira linha, a função deve devolver o número que responde à seguinte pergunta: na linha nº k , quantas quadrículas estão pintadas entre as posições a e b (inclusive)? É garantido que $a \leq b$.

Para passar no tempo pode ter que responder a 1000 perguntas destas num segundo, com $1 \leq a \leq b \leq 10^{16}$ e $K \leq 1000$.

Qual a magnitude da complexidade temporal e espacial que deverá ter?

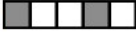
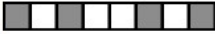


Como é um desafio não vamos para já dar mais pistas mas podem entrar em contacto com @Pedro Ribeiro no Slack para pedir dicas.

Exemplo de chamada e output esperado:

```
cout << FunWithSearch::rules(3, 3, 7) << endl;  
cout << FunWithSearch::rules(3, 1, 8) << endl;  
cout << FunWithSearch::rules(2, 1, 1) << endl;  
cout << FunWithSearch::rules(5, 27, 41) << endl;  
cout << FunWithSearch::rules(4, 9, 12) << endl;
```

```
2  
4  
1  
5  
0
```

Explicação:

- **3 3 7**: posições 3 a 7 da 3ª linha (2 quadriculas pintadas) 
- **3 1 8**: posições 1 a 8 da 3ª linha (4 quadriculas pintadas) 
- **2 1 1**: posições 1 a 1 da 2ª linha (1 quadricula pintada) 
- **5 27 41**: posições 27 a 41 da 5ª linha (5 quadriculas pintadas) 
- **4 9 12**: posições 9 a 12 da 4ª linha (0 quadriculas pintadas) 