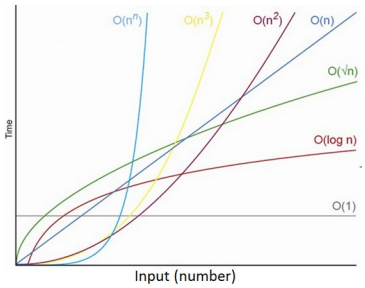
2ª Aula Prática
Complexidade e Análise Assintótica



1ª parte – Conceitos essenciais de Complexidade

(Pode ser útil consultar os slides do ***Capítulo 2 – Complexidade e Análise Assintótica***)

Recorde que:

- $f(n) \in \mathcal{O}(g(n))$ se existem constantes positivas n_0 e c tal que $f(n) \leq c \times g(n)$ para todo o $n \geq n_0$
- $f(n) \in \Omega(g(n))$ se existem constantes positivas n_0 e c tal que $f(n) \geq c \times g(n)$ para todo o $n \geq n_0$
- $f(n) \in \Theta(g(n))$ se existem constantes positivas n_0, c_1 e c_2 tal que $c_1 \times g(n) \leq f(n) \leq c_2 \times g(n)$ para todo o $n \geq n_0$

1. Notação Assintótica. Para os pares de funções seguintes indique se $f(n)$ é $\mathcal{O}, \Omega$ ou Θ de $g(n)$. A sua resposta resposta deve ser um verdadeiro ou falso em cada uma das células vazias da seguinte tabela: *(procure justificar informalmente as suas opções em cada caso)*

	$f(n)$	$g(n)$	$\mathcal{O}$	Ω	Θ
a)	$5n^2 + 100n + 34$	$2n^3 + 42$			
b)	$99n + 23$	$0.5n + 2$			
c)	2^n	$4n^2$			
d)	45	$\log_2 45$			
e)	$\log_2 n$	$\log_3 n$			
f)	$n \log n + n$	n^2			
g)	2^n	2^{n+2}			
h)	$n!$	2^n			

2. Complexidade de programas. Indique a complexidade temporal de cada um dos seguintes fragmentos de código *(procure justificar informalmente as suas opções em cada caso)*

a)
```
for (int i=0; i<n; i++)
   for (int j=i; j<n; j++)
      count++;
```

b)
```
for (int i=0; i<n; i++) count++;
for (int i=0; i<n; i++) count++;
for (int i=0; i<n; i++) count++;
```

c)
```
for (int i=0; i<n; i++)
   for (int j=0; j<n; j+=2)
      for (int k=0; k<10; k++)
         count++;
```

d)
```
for (int i=2; i<n-2; i++)
   for (int i=1; i<n; i*=2)
      count++;
```

e)
```
int f1(int n) {
   if (n <= 0) return 0;
   return 1 + f1(n-1);
}
```

f)
```
int f2(int n) {
   if (n <= 0) return 0;
   return 1 + f2(n/2);
}
```

3. **Previsão de tempo de execução.** Complete a tabela seguinte usando a informação já preenchida em cada linha. Use uma estimativa baseada na proporção entre n_1, n_2 e n_3 (ms = milissegundos).

Programa	**Complexidade Temporal**	**Tempo de execução (estimado) para:**		
		$n_1 = 10$	$n_2 = 20$	$n_3 = 40$
A	$\Theta(1)$		$100ms$	
B	$\Theta(n)$		$100ms$	
C	$\Theta(n^2)$		$100ms$	
D	$\Theta(n^3)$		$100ms$	
E	$\Theta(2^n)$		$100ms$	

2ª parte – Exercícios de implementação envolvendo complexidade

Instruções

- Faça download do ficheiro ***aed2223_p02.zip*** da página da disciplina e descomprima-o (contém a pasta ***lib,*** a pasta ***Tests*** com os ficheiros ***funWithComplexity.cpp, funWithComplexity.h***, ***Timer.cpp***, ***Timer.h*** e ***tests.cpp***, as pastas ***maxsubarray*** e ***river***, e os ficheiros ***CMakeLists*** e ***main.cpp***)
- No CLion, abra um projeto, selecionando 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 *funWithCycles.cpp*
- Note que apenas os primeiros testes de `maxSubArray` (o 1º exercício) estão descomentados; deverá ir descomentando os outros testes à medida que vai fazendo os exercícios.

4. **Sequências.** *(O objectivo deste exercício é ter algoritmos de diferentes complexidades para um mesmo problema e ir verificando a sua eficiência temporal)*

O João e a Maria estão a jogar um novo jogo. Um deles escreve uma sequência de números inteiros (positivos ou negativos) e o outro tem de tentar descobrir qual a subsequência contígua (um ou mais números consecutivos) que dá origem à maior soma possível.

Imagina por exemplo que a Maria escolhe a seguinte sequência de números:

```
-1  4 -2  5 -5  2 -20  6
```

Alguns exemplos de sequências contíguas seriam as seguintes:

```
|-1  4 -2  5 -5  2 -20  6| (soma=-11)
 -1  4 -2  5 -5 |2 -20  6| (soma=-12)
|-1  4 -2  5 -5  2|-20  6  (soma=3)
 -1  4 -2 |5|-5  2 -20  6  (soma=5)
 -1 |4 -2  5|-5  2 -20  6  (soma=7)
```

A última destas sequências corresponde precisamente à melhor sequência contígua possivel que o João poderia escolher, ou seja, a que tem maior soma.

Podes ajudar os dois amigos a jogarem este jogo?

Função a implementar:

```
int FunWithComplexity::maxSubArray(const vector<int> & v)
```

Esta função deve devolver a maior soma de uma subsequência contígua do vector *v*.

Exemplo de chamada e output esperado:

```
cout << FunWithComplexity::maxSubArray({-1, 4, -2, 5, -5, 2, -20, 6}) << endl;
cout << FunWithComplexity::maxSubArray({-2, -1, -3}) << endl;
cout << FunWithComplexity::maxSubArray({2, 4, 6, 8, 10}) << endl;
```

```
7
-1
30
```

Explicação: o primeiro caso é o do exemplo do enunciado e a solução é 7 (4-2+5); no segundo caso todos os números são negativos e por isso a melhor subsequência só tem tem tamanho 1 (contendo o maior número que é -1); na terceira sequência todos os números são positivos e por isso a melhor subsequência é toda a sequência (2+4+6+8+10).

a) Uma primeira solução com "força bruta" com tempo $\Theta(n^3)$

Quais são todas as subsequências contíguas possíveis? São todos os subarrays $v[i \ldots j]$ tal que $0 \leq i \leq j \leq n$. Uma solução exaustiva seria passar por todas estas subsequências e para cada uma delas calcular o valor da respectiva soma, escolhendo a melhor possível. Uma maneira de implementar esta ideia é dada pelo código seguinte:

```
unsigned n = v.size();
int maxSoFar = v[0];  // porque é esta uma boa escolha para a melhor soma inicial?
for (unsigned i=0; i<n; i++) // todas as posicoes iniciais possiveis
   for (unsigned j=i; j<n; j++) { // todas as posicoes finais possiveis
      int sum = 0;
      for (unsigned k=i; k<=j; k++) // calcular soma entre posicoes i e j
         sum += v[k];
      // neste momento sum é a soma dos elementos no intervalo [i,j]
      if (sum > maxSoFar) maxSoFar = sum;
   }
return maxSoFar;
```

Coloque o código dentro da função `maxSubArray` e procure percebê-lo (e porque a sua complexidade temporal é cúbica). Execute o código e veja como passa corretamente nos testes descomentados inicialmente. Note como para um array de tamanho $n = 1\,000$ já demora algum tempo a calcular.

Experimente submeter no Mooshak e veja como obtém um Time Limit Exceeded para os casos maiores *(o seu programa está limitado a 1s de execução na máquina do Mooshak)*.

Usando o que sabe, estime quanto tempo demoraria para um caso de $n = 10\,000$ na sua máquina. Experimente descomentar os casos de teste com $n = 10\,000$ e execute. Demora muito tempo? (se estiver a demorar muito pode parar a sua execução – a sua estimativa deve dar-lhe hipótese de saber quanto tempo teria de esperar...)

b) Melhorando a solução para $\Theta(n^2)$

Intuitivamente, olhando para o código anterior podemos notar que em cada soma estamos a repetir muitos cálculos. Quando passamos do cálculo de `soma(v[i]...v[j])` para `soma(v[i]...v[j+1])` não precisamos de voltar a recalcular tudo (começando novamente em `i`), e basta-nos adicionar `v[j]` à soma anterior!

Dito de outro modo, `soma(v[i]...v[j+1]) = soma(v[i]...v[j]) + v[j+1]`. Podemos utilizar isto para remover o terceiro ciclo com `k` que tínhamos na solução anterior.

Implemente esta nova solução e teste com $n = 10\,000$. Quanto tempo demorou agora? Submeta a solução no Mooshak e veja como obtém mais pontos por passar em mais testes (mas continua a ter Time Limit Exceeded nos casos ainda maiores).

Descomente o caso com $n = 100\,000$. Antes de executar tente prever o tempo que vai demorar e veja se realmente o tempo de execução é perto do esperado (pode aguardar que termine a execução se o seu programa é genuinamente quadrático no tempo). Precisamos ainda de baixar o tempo de execução para menos de 1s...

c) Melhorando a solução para $\Theta(n)$

Uma solução quadrática ainda não é suficiente e temos de a melhorar para tempo linear. Para isso vamos usar o algoritmo de Kadane.

Considere que `best[i]` representa o melhor subarray que termina na posição `i`. Sabemos como "caso base" que `best(0) = v[0]` (é o único subarray possível que termina na primeira posição).

Se soubermos o valor de `best(i)`, como calcular o valor de `best(i+1)`?

- Se `best(i)` for um valor positivo, então `best(i+1) = best(i) + v[i+1]` pois como os subarrays têm de ser contíguos, se aproveitarmos algo "de trás", terá que ser o melhor possível a terminar na posição anterior. Imagine por exemplo o array [4, -2, 1, 5]:

i	0	1	2	3
v[i]	4	-2	1	5
best(i)	4	2	3	8

`best(2) = 3`, obtido através do subarray `[4, -2, 1]`
`best(3) = best(2) + v[3] = 3 + 5 = 8`, obtido através do subarray `[4, -2, 1, 5]` *(estendeu-se o array anterior)*

- Se `best(i)` for um valor negativo, então `best(i+1)=v[i+1]`, pois o que está "para trás" só pode fazer decrescer a soma do array a terminar na posição `i+1` . Imagine por exemplo o array [-5, 2, -3, 4]:

i	0	1	2	3
v[i]	-5	2	-3	4
best(i)	-5	2	-1	4

`best(2) = -1`, obtido através do subarray `[2, -3]`
`best(3) = v[3] = 4`, obtido através do subarray `[4]` *(não se estendeu o array anterior)*

No final de tudo, o melhor subarray é o melhor valor de um qualquer `best(i)` (o melhor pode terminar em qualquer posição).

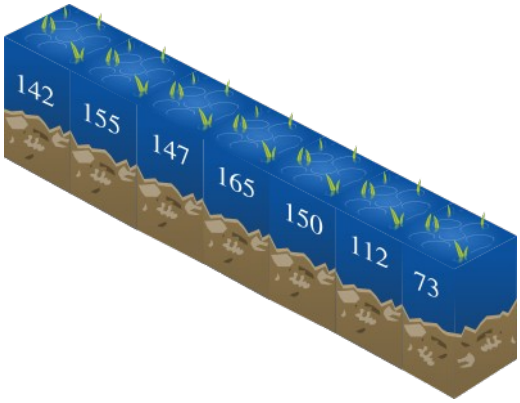
Para calcular isto basta-nos percorrer uma única vez o array e em cada iteração calcular em tempo constante o valor da melhor soma a terminar na posição actual, usando a melhor soma a terminar na posição anterior. A complexidade temporal fica portanto linear.

Implemente esta solução, execute para ver o tempo de execução para $n = 100\,000$ e depois de confirmar a correção, submeta-a no Mooshak, verificando que finalmente obtém o seu Accepted!

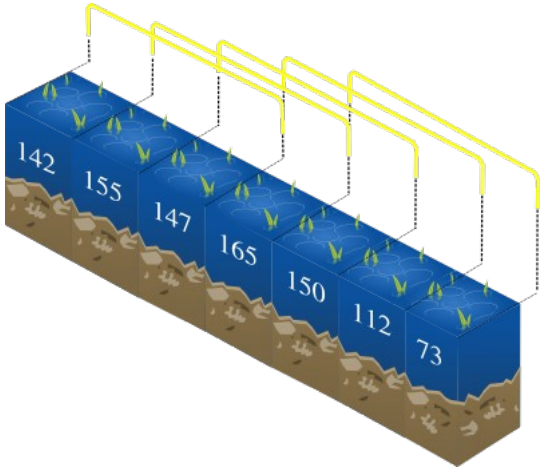
5. Analisando um rio.

(exercício baseado num problema da fase de qualificação das Olimpíadas Nacionais de Informática 2019)

Encomendaram-lhe o estudo da geografia de um rio, que pode ser pensado como uma sequência de ***N*** localizações, referindo para cada uma delas a sua profundidade, em unidades de comprimento. Por exemplo, um rio com ***N=7*** localizações poderia ser o seguinte:

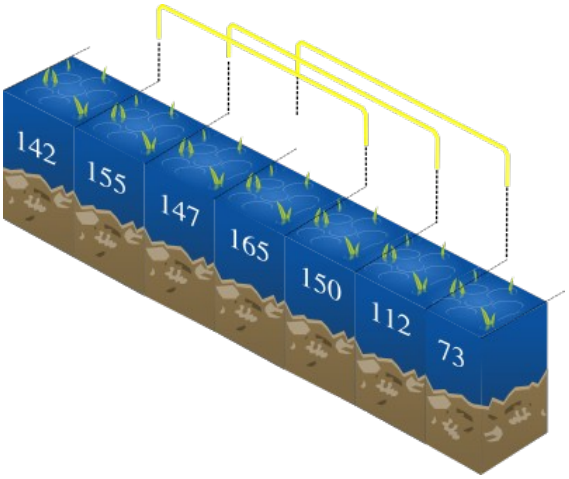


Queremos analisae regiões de ***K*** localizações consecutivas, que podem ter sobreposição entre si. Por exemplo, se ***K=3***, então existem 5 regiões possíveis, indicadas na imagem seguinte:



Diz-se que uma região é **satisfatória** se **pelo menos metade** das suas *K* localizações tiver uma profundidade **maior ou igual** a ***T***.

Para o exemplo da figura, se *K=3* e *T=150* temos que, existem 3 regiões que seguem a propriedade mencionada, tal com indicado na figura seguinte:



Para um caso geral, quantas regiões são satisfatórias?

Função a implementar:

```
int FunWithComplexity::river(const vector<int> & v, int k, int t)
```

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

Esta função deve devolver a quantidade de regiões de *k* localizações contínuas de *v* onde pelo menos metade das localizações têm profundidade maior ou igual a *t*.

Exemplo de chamada e output esperado:

```
cout << FunWithComplexity::river({142,155,147,165,150,112,73}, 3, 150) << endl;
cout << FunWithComplexity::river({5,10,12,10,9,14,5,7,9,11,3,3}, 4, 10) << endl;
```

```
3
4
```

Explicação: o primeiro caso foi o explicado no enunciado; no segundo caso procuramos regiões de 4 localizações consecutivas onde a profundidade é maior ou igual a 10.

Para este problema, uma solução "bruta" que tente todos os intervalos e para cada um deles percorra todos os seus elemento para verificar quantos são maiores ou iguais a T pode demorar $\Theta(n^2)$ (basta imaginar o caso onde $k = n/2$: teremos nesse caso $n/2 + 1$ intervalos, cada um com tamanho $n/2$ que temos de verificar). Isto não é suficiente para passar no tempo, e precisamos de uma solução melhor...

*Sugestão (não leia se não quiser **spoilers**):*
- Imagine que já processou o intervalo $[a, b]$ (entre as posições a e b, tendo obtido Q_a, a quantidade de elementos maiores ou iguais a T nesse intervalo.

- O próximo intervalo a considerar é o intervalo $[a + 1, b + 1]$. O que muda nesse intervalo? Apenas desaparece a posição a e é inserida a posição $b + 1$. O (novo) valor de Q_{a+1} é portanto igual a Q_a subtraíndo 1 caso o valor da posição a seja $\geq T$e somando 1 caso o valor da posição $b + 1$ seja $\geq T$.

- Para cada novo intervalo precisamos então apenas de um número constante de operações (considerar o elemento a retirar e considerar o elemento a inserir). Em termos de eficiência, isto corresponde a um ciclo inicial para o primeiro intervalo de tamanho e depois percorrer os restantes intervalos, gastando em cada um tempo constante. Outra maneira de pensar no mesmo algoritmo é ver que cada elemento é adicionado uma vez ao intervalo *actual*, e retirado uma vez. Esta é portanto uma solução com tempo $\Theta(n)$ que passa em todos os casos e dá os **100** pontos e Accepted no Mooshak.

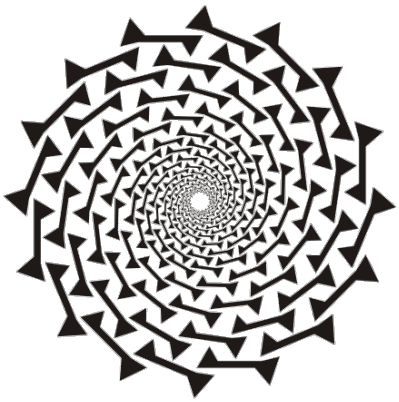
Esta ideia de reutilizar o resultado do intervalo anterior para calcular mais rapidamente o intervalo atual chama-se sliding window, pois estamos a fazer "deslizar" a janela corrente ao longo do input.

Exercícios Extra

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

6. Espiral de Números. Imagina a seguinte espiral de números (começa em 01 e vai andando por ordem crescente):

```
|       +--+--+--+--+--+--+
|  +2   |21|22|23|24|25|26|
|       +--+--+--+--+--+--+
|  +1   |20|07|08|09|10|27|
        +--+--+--+--+--+--+
Y   0   |19|06|01|02|11|28|
        +--+--+--+--+--+--+
|  -1   |18|05|04|03|12|29|
|       +--+--+--+--+--+--+
|  -2   |17|16|15|14|13|30|
|       +--+--+--+--+--+--+
                        ...
          -2 -1  0 +1 +2 +3
        ------- X --------
```



Imagina agora um sistema de coordenadas (X, Y), onde a origem seja o número 01. Desse modo, podemos dizer que 01 está na posição (0,0) , 02 está na posição (1,0), 03 na posição (1,-1), 04 na posição (0,-1), 05 na posição (-1,-1), e por aí adiante.

Consegues fazer um programa para calcular as coordenadas de um número da espiral?

Função a implementar:

```
pair<int, int> FunWithComplexity::spiral(int n)
```

Complexidade temporal esperada: $\mathcal{O}(\sqrt{n})$.
(o *seu programa será testado até* $n = 10^9$ *pelo que uma solução linear "naive" não passará no tempo*)

Esta função deve devolver um par de inteiros indicado as coordenadas na espiral do número n. O primeiro elemento do par deve ser o X e o segundo elemento o Y.

Exemplo de chamada e output esperado:

```
pair<int, int> p1 = FunWithComplexity::spiral(5);
cout << p1.first << "," << p1.second << endl;
pair<int, int> p2 = FunWithComplexity::spiral(25);
cout << p2.first << "," << p2.second << endl;
pair<int, int> p3 = FunWithComplexity::spiral(3);
cout << p3.first << "," << p3.second << endl;
```

```
-1,-1
2,2
1,-1
```

Explicação: as coordenadas do 5 são X=Y=-1; as coordenadas do 25 são X=Y=2,
as coordenadas do 3 são X=1, Y=-1

Uma solução bruta que passe por todos números até *n* é $\Theta(n)$ e não vai passar no tempo em todos os testes. Consegue ver algum padrão na espiral que possa ser aproveitado? Por exemplo, alguma das linhas diagonais segue um padrão caractertístico?

7. **Complexidade de funções recursivas e recorrências.** Indique uma recorrência que represente o tempo de execução de cada uma das funções recursivas seguintes, bem como sua a complexidade temporal. *(procure justificar informalmente as suas opções em cada caso – ex: desenhando a árvore de recursão)*

a)

```
int g1(int n) {
   if (n<=0) return 0;
   int count = 0;
   for (int i=0; i<n;i++) count++;
   return count + g1(n/2) + g1(n/2);
}
```

b)

```
int g2(int n) {
   if (n<=0) return 0;
   return 1 + g2(n/2) + g2(n/2);
}
```

c)

```
int g3(int n) {
   if (n<=0) return 0;
   return 1 + g3(n/3);
}
```

d)

```
int g4(int n) {
   if (n<=0) return 0;
   return 1 + g4(n-1) + g4(n-1);
}
```

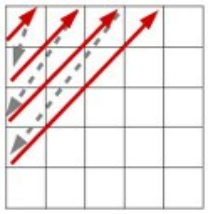
Exercício de Desafio

(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 qualificação das Olimpíadas Nacionais de Informática 2016)

A Sara adora o seu caderno quadriculado de matemática. Para passar o tempo começou a escrever os números inteiros consecutivamente. Achou, contudo, que fazê-lo de esquerda para a direita e de cima para baixo era muito aborrecido! Resolveu por isso preencher os números pelas diagonais usando o seguinte padrão:



Isto resulta num preenchimento das quadrículas como representado a seguir, onde se vê parte do caderno da Sara (as outras linhas e colunas que não aparecem na figura estão preenchidas com números também):

1	3	6	10	15	21	28	36
2	5	9	14	20	27	35	44
4	8	13	19	26	34	43	53
7	12	18	25	33	42	52	63
11	17	24	32	41	51	62	74
16	23	31	40	50	61	73	86

A Sara acha que ficou um padrão muito giro! Como na sua escola está na fase de aprender a operação de adição, resolveu começar a somar os números contidos dentro de retângulos. Por exemplo, se considerasse o retângulo da figura seguinte (com cantos nos números 13 e 51), a soma seria igual a 358 (13+19+26+34+18+25+33+42+24+32+41+51):

1	3	6	10	15	21	28	36
2	5	9	14	20	27	35	44
4	8	13	19	26	34	43	53
7	12	18	25	33	42	52	63
11	17	24	32	41	51	62	74
16	23	31	40	50	61	73	86

A Sara gostava de poder verificar se as somas estão corretas. **Claro que ela gostava de saber a soma de um qualquer retângulo dado.** Será que podes ajudá-la?

Função a implementar:

```
long long FunWithComplexity::gridSum(int a, int b)
```

Dados dois inteiros *a* e *b,* indicando respetivamente os cantos superior esquerdo e inferior direito do retângulo a considerar, deve devolver a soma dos números no retângulo correspondente.

Exemplo de chamada e output esperado:

```
cout << FunWithComplexity::gridSum(13, 51) << endl;
cout << FunWithComplexity::gridSum(5, 31) << endl;
cout << FunWithComplexity::gridSum(28, 44) << endl;
cout << FunWithComplexity::gridSum(25, 245) << endl;
```

```
358
160
143
7480
```

Explicação: os quatro retângulos do exemplo correspondem aos retângulos das figuras seguintes

- o primeiro (a vermelho) tem cantos 13 e 51 e uma soma de 358;
- o segundo (a verde) tem cantos 5 e 31 e uma soma de 160;
- o terceiro (a azul) tem cantos 28 e 44 e uma soma de 143;
- o quarto (a amarelo) tem cantos 25 e 245 e uma soma de 7480.

1	3	6	10	15	21	28	36
2	5	9	14	20	27	35	44
4	8	13	19	26	34	43	53
7	12	18	25	33	42	52	63
11	17	24	32	41	51	62	74
16	23	31	40	50	61	73	86

1	3	6	10	15	21	28	36
2	5	9	14	20	27	35	44
4	8	13	19	26	34	43	53
7	12	18	25	33	42	52	63
11	17	24	32	41	51	62	74
16	23	31	40	50	61	73	86

1	3	6	10	15	21	28	36
2	5	9	14	20	27	35	44
4	8	13	19	26	34	43	53
7	12	18	25	33	42	52	63
11	17	24	32	41	51	62	74
16	23	31	40	50	61	73	86

1	3	6	10	15	21	28	36	45	55	66	78	91	105	120
2	5	9	14	20	27	35	44	54	65	77	90	104	119	135
4	8	13	19	26	34	43	53	64	76	89	103	118	134	151
7	12	18	25	33	42	52	63	75	88	102	117	133	150	168
11	17	24	32	41	51	62	74	87	101	116	132	149	167	186
16	23	31	40	50	61	73	86	100	115	131	148	166	185	205
22	30	39	49	60	72	85	99	114	130	147	165	184	204	225
29	38	48	59	71	84	98	113	129	146	164	183	203	224	246
37	47	58	70	83	97	112	128	145	163	182	202	223	245	268
46	57	69	82	96	111	127	144	162	181	201	222	244	267	291

Poderá usar 256MB de memória e para ter Accepted terá de conseguir responder em 1s a 1 000 chamadas a `gridSum`, sendo que *a* e *b* podem ir até 10^9 (os retângulos a considerar não têm mais do que 10 000 linhas ou colunas). 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.