2nd Recitation

Complexity and Asymptotic Analysis

Part 1 - Essential Concepts of Complexity

Remember that:

- $f(n) \in O(g(n))$ *if there are positive constants* n_0 *and* c *such that* $f(n) \leq c \times g(n)$ *for all* $n \geq n_0$
- $f(n) \in \Omega(g(n))$ *if there are positive constants* n_0 *and* c *such that* $f(n) \geq c \times g(n)$ *for all* $n \geq n_0$
- $f(n) \in \Theta(g(n))$ *if there are positive constants* n_0, c_1, c_2 *such that* $c_1 \times g(n) \leq f(n) \leq c_2 \times g(n)$ *for all* $n \geq n_0$

1. Asymptotic Notation. For the following pairs of functions, indicate whether if $f(n)$ is O, Ω or Θ of $g(n)$. Your answer must be true or false in each empty cell in the following table: (try to justify your options in each case informally).

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

o -> a cima
omega -> a baixo
theta -> igual

f(n) -> real
g(n) -> referência

Verificar em: https://www.desmos.com/calculator

2. Program complexity. Indicate the asymptotic time complexity of each of the following code fragments (try to informally justify your choices in each case)

a) f(n^2)

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

b) f(n*3)=f(n)

```
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) f(n^2)

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

d) f(n*log(n))

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

e) f(n)

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

f) f(log(n))

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

3. Estimated Execution Time. Complete the following table using the information already filled in in each row. Use an estimate based on the ratio between n_1, n_2 and n_3 (ms = milliseconds).

Program	Time Complexity	Execution time (estimated) for:		
		$n_1 = 10$	$n_2 = 20$	$n_3 = 40$
A	$\Theta(1)$	100ms	100ms	100ms
B	$\Theta(n)$	50ms	100ms	200ms
C	$\Theta(n^2)$	10ms	100ms	1000ms
D	$\Theta(n^3)$	(raiz cúbica de 100)ms	100ms	1000000ms
E	$\Theta(2^n)$	102400ms	100ms	2^20 * 100ms

Tempo para n=10: 2^(20-10) * 100ms = 2^10 * 100ms = 1024 * 100ms = 102400ms

Tempo para n=40: 2^(40-20) * 100ms = 2^20 * 100ms

Part 2 - Implementation exercises involving complexity

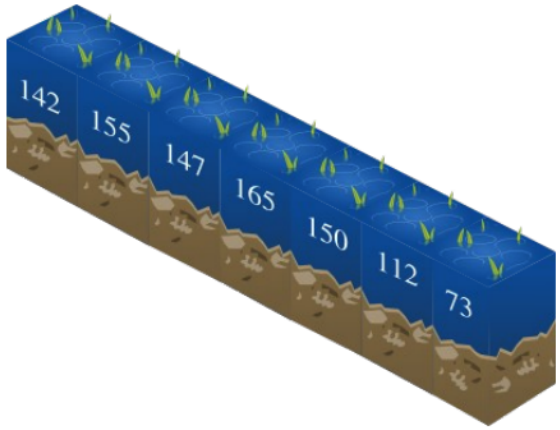
Instructions

- Download the `aed2324_p02.zip` file from the course page and unzip it (it contains the `lib` folder, the `Tests` folder with the `funWithComplexity.cpp`, `funWithComplexity.h`, `Timer.cpp`, `Timer.h` and `tests.cpp` files, the `maxsubarray` and `river` folders, and the `CMakeLists` and `main.cpp` files).
- In CLion, open a project by selecting the folder containing the files from the previous point
- If you can't compile, perform "`Load CMake Project`" on the `CMakeLists.txt` file
- Make your implementation it in `funWithComplexity.cpp`

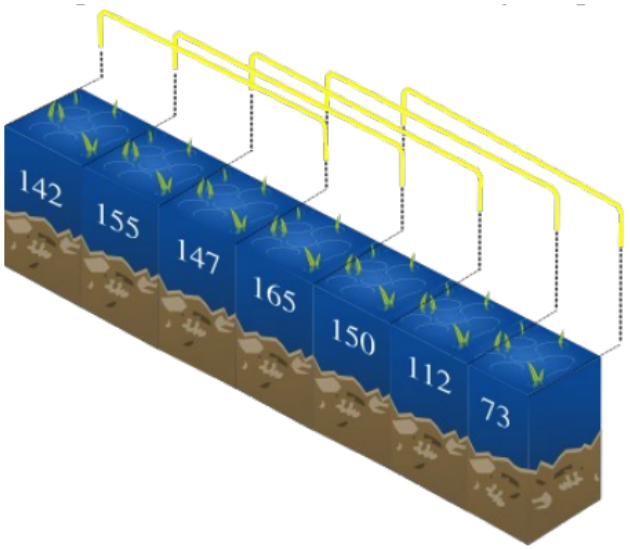
Note: For exercises 4 and 5, you can uncomment the `BENCHMARK` tests (at the end of the `tests.cpp` file) to verify the time complexity.

4. Analyzing a River
(*exercise based on a problem from the qualifying round of the National Computer Olympics 2019*)

They commissioned you to study the geography of a river, which can be thought of as a sequence of `N` locations, with each one indicating its depth in units of length. For example, a river with `N=7` locations could look like this:

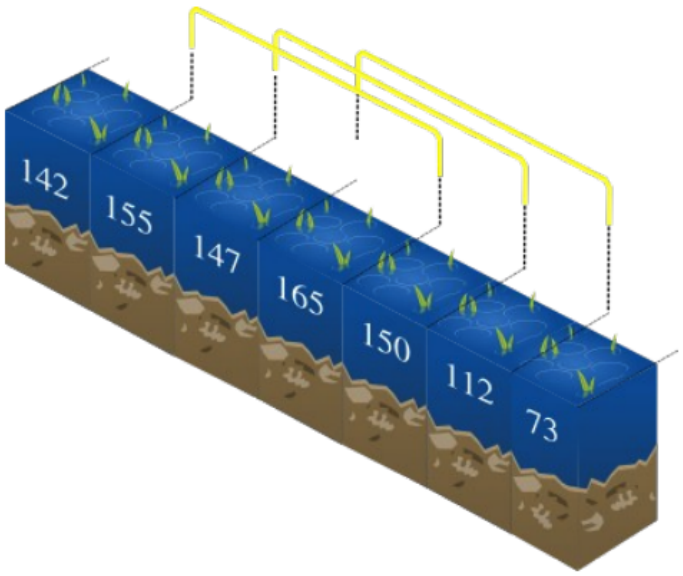


We want to analyze regions of `K` consecutive locations, which may overlap with each other. For example, if `K=3`, then there are 5 possible regions, indicated in the following image:



A region is said to be **satisfactory** if **at least half** of its `K` locations have a depth greater than or equal to T.

For the example in the figure, if `K=3` and `T=150` then there are 3 regions that follow the above property, as shown in the following figure:



For a general case, how many regions are satisfactory?

Function to implement:

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

Expected time complexity: $\Theta(n)$ where n is the size of the vector.

This function should return the number of regions of k continuous locations of v where at least half of the locations have a depth greater than or equal to t.

Example call and expected output:

```
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
```

Explanation: the first case was the one explained in the statement; in the second case we looked for regions of 4 consecutive locations where the depth is greater than or equal to 10.

Suggestion: Imagine that you have already processed the interval $[a, b]$ (between the positions a and b, having obtained Q_a, the number of elements greater than or equal to T in that interval.

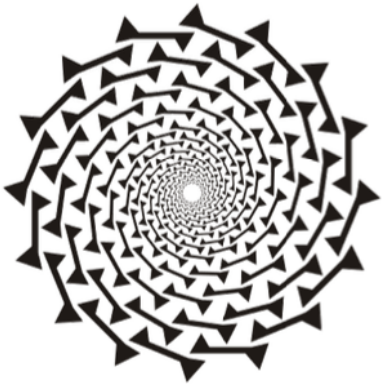
- The next interval to consider is $[a + 1, b + 1]$. What changes in this interval? Only the position a disappears and the position $b + 1$ is inserted. The (new) value of Q_{a+1} is therefore equal to Q_a subtracting 1 if the a position value is $\geq T$ and adding 1 if the $b + 1$ position value is $\geq T$.
- For each new interval, we only need a constant number of operations (consider the element to be removed and consider the element to be inserted). In terms of efficiency, this corresponds to an initial cycle for the first size interval and then going through the remaining intervals, spending a constant amount of time on each one. Another way of thinking about the same algorithm is to see that each element is added once to the *current* interval, and removed once. This is therefore a solution with time $\Theta(n)$ that passes all tests.

This idea of reusing the result of the previous interval to calculate the current interval more quickly is called sliding window, because we are "sliding" the current window along the input.

Extra Exercises

5. Spiral of Numbers. Imagine the following spiral of numbers (it starts at 01 and goes in ascending order):

```
|       +--+--+--+--+--+--+
|  +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 --------
```



Now consider a (X,Y) coordinate system where the origin is the number 01. In this way, we can say that 01 is at position (0,0), 02 is at position (1,0), 03 at position (1,-1), 04 at position (0,-1), 05 at position (-1,-1), and so on.

Can you make a program to calculate the coordinates of a number on the spiral?

Function to implement:

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

Expected time complexity: $\Theta(\sqrt{n})$.

(*your program will be tested until* $n = 10^9$*, so a "naive" linear solution will take a long time*)

This function must return a pair of integers indicating the coordinates on the spiral of the number. The first element of the pair must be X and the second element Y.

Example call and expected output:

```
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
```

Explanation: the coordinates of 5 are X=Y=-1; the coordinates of 25 are X=Y=2, the coordinates of 3 are X=1, Y=-1.

A brute force solution that passes through all the numbers up to n is $\Theta(n)$ and will not pass in time in all the tests. Can you observe any patterns in the spiral that could be used? For example, do any of the diagonal lines follow a characteristic pattern?

6. Complexity of Recursive Functions and Recursions.
Indicate a recurrence that represents the execution time of each of the following recursive functions, as well as their time complexity. (*Try to informally justify your choices in each case - e.g. by drawing the recursion tree*)

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)

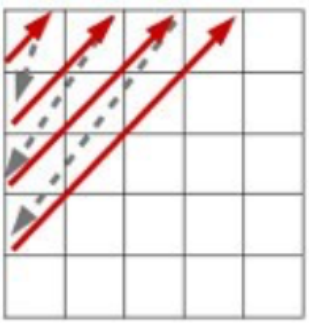
```
int g4(int n) {
  if (n<=0) return 0;
  return 1 + g4(n-1) + g4(n-1);
 }
```

Challenge Exercise

(*substantially more difficult exercise for students who want to have additional challenges with more algorithmically complex problems*)

7. Squared Notebook.
(*exercise based on a problem from the qualifying round of the National Computer Olympics 2016*)

Sara loves her math notebook. To pass the time, she started writing the whole numbers consecutively. However, she found that doing it from left to right and top to bottom was too boring! So, she decided to fill in the numbers along the diagonals using the following pattern:



This results in the squares being filled in as shown below, where you can see part of Sara's notebook (the other rows and columns that don't appear in the picture are also filled in with numbers):

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

Sara thinks it is a nice pattern! As she is learning addition at school, she decided to start adding up the numbers inside the rectangles. For example, if she took the rectangle in the following picture (with corners at 13 and 51), the sum would be 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

Sara would like to be able to check that the sums are correct. **Of course, she'd like to know the sum of any given rectangle**. Can you help her?

Function to implement:

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

Given two integers a and b, indicating respectively the top left and bottom right corners of the rectangle to be considered, return the sum of the numbers in the corresponding rectangle.

Example call and expected output:

```
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
```

Explanation: the four rectangles in the example correspond to the rectangles in the following figures

- the first (in red) has corners 13 and 51 and a sum of 358;
- the second (in green) has corners 5 and 31 and a sum of 160;
- the third (in blue) has corners 28 and 44 and a sum of 143;
- the fourth (in yellow) has corners 25 and 245 and a sum of 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

You can use 256MB of memory and it must be able to respond in 1s to 1,000 calls to `gridSum`, where a and b can go up to 10^9 (the rectangles to be considered have no more than 10,000 rows or columns). How much time and space complexity should it have?