Técnicas (dicas) de Programação

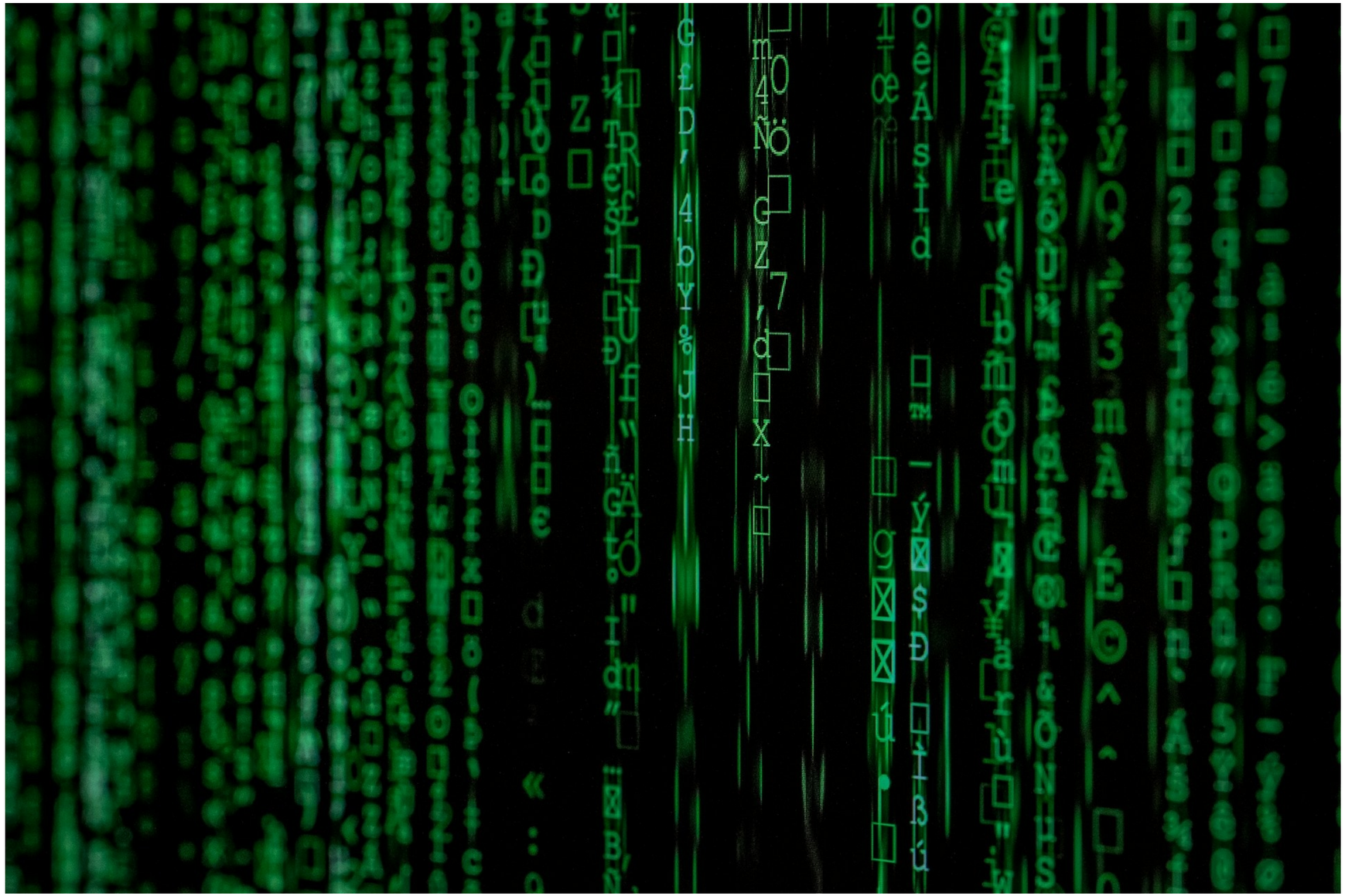

Foto de Markus Spiske no Unsplash (https://unsplash.com/photos/matrix-movie-still-iar-afB0QQw)

COMPUTAÇÃO RAIZ

Vamos revisar algumas dicas de C/C++ que são úteis!

- Templates C/C++
- Input e Output
- Números
- Encurtamento de código
- Matemática

TAREFAS PARA CASA:

- Algoritmos recursivos básicos:
    - Gerar todos os subconjuntos e permutações de um conjunto
- Algoritmos de backtracking
    - Contar coisas (problema das n-Queens)
- Operações bitwise
    - Bitwise na representação de subconjuntos de um conjunto

COMPUTAÇÃO RAIZ

Templates C/C++

Logo no início da maratona um membro do time deve digitar o template da equipe e salvá-lo para poder ser reutilizado várias e várias vezes.

Cada time deve definir qual o melhor template para a linguagem que trabalharão: C ou C++.

As funcionalidades do template devem ser totalmente dominadas pelo time!

```
1  #include <ctype.h>
2  #include <limits.h>
3  #include <math.h>
4  #include <stdbool.h>
5  #include <stdint.h>
6  #include <stdio.h>
7  #include <stdlib.h>
8  #include <string.h>
9  #include <strings.h>
10 #include <time.h>
11
12 typedef long int li;
13 typedef long long int lli;
14 typedef unsigned long int uli;
15 typedef unsigned long long int ulli;
16 typedef long double ld;
17
18 const int INF = 1e9;
19 const lli LLINF = 4e18;
20 const double EPS = 1e-9;
21
22 int main (void)
23 {
24     // SEU CÓDIGO AQUI!!!
25
26     return 0;
27 }
```

COMPUTAÇÃO RAIZ

Templates C/C++

Logo no início da maratona um membro do time deve digitar o template da equipe e salvá-lo para poder ser reutilizado várias e várias vezes.

Cada time deve definir qual o melhor template para a linguagem que trabalharão: C ou C++.

As funcionalidades do template devem ser totalmente dominadas pelo time!

```
1  #include <bits/stdc++.h>
2
3  using namespace std;
4
5  typedef long int li;
6  typedef long long int lli;
7  typedef unsigned long int uli;
8  typedef unsigned long long int ulli;
9  typedef long double ld;
10
11 typedef vector<int> vi;
12 typedef vector<lli> vlli;
13 typedef pair<int, int> pi;
14 typedef vector<pi> vpi;
15
16 const int INF = 1e9;
17 const lli LLINF = 4e18;
18 const double EPS = 1e-9;
19
20 int main (void)
21 {
22     ios::sync_with_stdio(0);
23     cin.tie(0);
24
25     // SEU CÓDIGO AQUI!!!
26
27     return 0;
28 }
```

COMPUTAÇÃO RAIZ

Compilação

NÃO USE A IDE PARA COMPILAR!

PARA COMPILAR, USE SEMPRE O TERMINAL!

Para C:

```
gcc -std=c17 -Wall -Wextra -Wpedantic -Werror -lm -lrt -o programa programa.c
```

Para C++:

```
g++ -std=c++17 -Wall -Wextra -Wpedantic -Werror -lm -lrt -o programa programa.cpp
```

Cada grupo deve, ao longo do curso, verificar quais as flags do compilador e do linker foram necessárias para a resolução dos problemas, e usar o que for adequado durante o treinamento e maratona.

Deve-se descobrir qual o padrão do C/C++ estará disponível!

Compilação: use Makefiles!

```
1  # Escolhe compilador para cada linguagem
2  C = gcc
3  CPP = g++
4
5  # FLAGS para compilador e linker, para linguagem C:
6  CCFLAGS = -std=c17 -Wall -Wextra -Wpedantic -Werror -g3 -O0
7  CLFLAGS = -lm -lrt
8
9  # FLAGS para compilador e linker, para linguagem C++:
10 CPPCFLAGS = -std=c++17 -Wall -Wextra -Wpedantic -Werror -g3 -O0
11 CPPLFLAGS = -lm -lrt
12
13 # Listagem dos programas da maratona (ex.: de A até P)
14 PROGRAMAS = A B C D E F G H I J K L M N O P
15
16 # Compila todos os programas (desnecessário, mas OK)
17 .PHONY: all
18 all: $(PROGRAMAS)
19
20 # Exemplo para linguagem C:
21 A: A.c
22         $(C) $(CCFLAGS) -o A A.c $(CLFLAGS)
23
24 # Exemplo para linguagem C++:
25 B: B.cpp
26         $(CPP) $(CPPCFLAGS) -o B B.cpp $(CPPLFLAGS)
```

Cada time deve padronizar:
- nome dos arquivos fonte
- processo de compilação
- flags do compilador
- flags do links
- diretórios dos arquivos
- etc.

Logo no início da maratona um membro do time deve digitar o template da `Makefile` e salvá-lo para poder ser reutilizado várias e várias vezes.

COMPUTAÇÃO RAIZ

Rotinas de input/output típicas

Qualquer que seja o problema, você deve dominar as rotinas de input de dados e output de resultados. Os problemas apresentam diversos tipos de input/output diferentes, por exemplo:

- **Número de casos de teste na primeira linha e as demais linhas são os casos**

- **Múltiplos casos de teste que são terminados por valores especiais, podendo ter ou não inputs adicionais que devem ser ignorados**

- **Múltiplos casos de teste terminados por EOF ou altum outro terminador especificado**

COMPUTAÇÃO RAIZ

Rotinas de input/output típicas

```c
/* Números de casos de teste na 1ª linha */
int TC;
scanf("%d", &TC);
while (TC--)
{
    int a, b;
    scanf("%d %d", &a, &b);
    printf("%d\n", a + b);
}
```

```
-----------------------------
| 3            | 3
| 1 2          | 12
| 5 7          | 9
| 6 3          |--------------
|--------------|
|              |
-----------------------------
```

```c
/* Casos terminados por valor especial (0) */
int a, b;
while (scanf("%d %d", &a, &b), (a || b))
    printf("%d\n", a + b);
```

```
-----------------------------
| 1 2          | 3
| 5 7          | 12
| 6 3          | 9
| 0 0          |--------------
| 1 1          |
-----------------------------
```

```c
/* Múltiplos casos de teste terminados por EOF */
int x, y;
while (scanf("%d %d", &x, &y) == 2)
    printf("%d\n", x + y);
```

```
-----------------------------
| 1 2          | 3
| 5 7          | 12
| 6 3          | 9
|--------------|--------------
|              |
```

COMPUTAÇÃO RAIZ

Rotinas de input/output típicas

Alguns problemas exigem que o output dos casos de teste seja numerado sequencialmente, com ou sem uma quebra adicional de linha. Por exemplo: se o problema especificar o seguinte:

"`Case [NUMBER]: [ANSWER]`" seguida de linha em branco

```
int a, b, c = 0;
while (scanf("%d %d", &a, &b) == 2)
    printf("Case %d: %d\n\n", ++c, a + b);
```

ATENÇÃO: também imprime uma linha em branco após o último caso de teste! Tenha certeza de que o problema exige isso.

```
-----------------------------
| 1 2          | Case 1: 3
| 5 7          |
| 6 3          | Case 2: 12
|--------------|
|              | Case 3: 9
|              |
|              |--------------
```

COMPUTAÇÃO RAIZ

Rotinas de input/output típicas

Alguns problemas exigem que o output dos casos de teste sejam numerados sequencialmente, com ou sem uma quebra adicional de linha. Por exemplo: se o problema especificar o seguinte:

"`Case [NUMBER]: [ANSWER]`" seguida de linha em branco exceto após a última linha

```
int a, b, c = 0;
while (scanf("%d %d", &a, &b) == 2)
{
    if (c > 0) printf("\n");
    printf("Case %d: %d\n", ++c, a + b);
}
```

```
-----------------------------
| 1 2          | Case 1: 3
| 5 7          |
| 6 3          | Case 2: 12
|--------------|
|              | Case 3: 9
|              |--------------
```

COMPUTAÇÃO RAIZ

Rotinas de input/output típicas

Alguns problemas têm um número variável de inputs por linha. Nesse caso, uma abordagem comum é dizer quantos inputs serão fornecidos no começo da linha, e depois colocar os inputs:

```
int k;
while (scanf("%d", &k) == 1)
{
    int ans = 0, v;
    while (k--)
    {
        scanf("%d", &v);
        ans += v;
        printf("%d\n", ans);
    }
}
```

```
-----------------------------
| 1 1          | 1
| 2 3 4        | 7
| 3 8 1 1      | 10
| 4 7 2 9 3    | 21
| 5 1 1 1 1 1  | 5
|--------------|--------------
```

COMPUTAÇÃO RAIZ

Rotinas de input/output típicas

Se os problemas têm um número variável de inputs por linha, mas a quantidade de inputs por linha não é informada, temos que verificar se o último caractere é \n:

```
while (1)
{
    int ans = 0, v;
    char l;
    while (scanf("%d%c", &v, &l) == 2)
    {
        ans += v;
        if (l == '\n') break;
    }
    if (feof(stdin)) break;
    printf("%d\n", ans);
}
```

1	1
3 4	7
8 1 1	10
7 2 9 3	21
1 1 1 1 1	5

COMPUTAÇÃO RAIZ

Rotinas de input/output típicas

Em C++, usamos `cin` para o input, por exemplo: se o input é formado por 2 inteiros e 1 string, como por exemplo em

```
123 456 monkey
```

ou

```
123    456
monkey
```

Podemos fazer:

```
int a, b;
string x;
cin >> a >> b >> x;
```

Obs.: scanf também funciona

COMPUTAÇÃO RAIZ

Rotinas de input/output típicas

Em C++, se precisarmos ler uma linha inteira, possivelmente contendo vários espaços em branco, podemos usar:

```
string s;
getline(cin, s);
```

COMPUTAÇÃO
RAIZ

Rotinas de input/output típicas

Em C++, se a quantidade de dados é desconhecida, podemos fazer algo como:

```
int x;
while (cin >> x)
{
    // código
}
```

COMPUTAÇÃO
RAIZ

Rotinas de input/output típicas

Em C++, usamos `cout` para o output, por exemplo:

```
int a = 123, b = 456;
string x = "monkey";
cout << a << " " << b << " " << x << "\n";
```

Obs.: printf também funciona

COMPUTAÇÃO RAIZ

Rotinas de input/output típicas

Em alguns problemas o input será um arquivo e o output também será outro arquivo. Podemos fazer uma "gambiarra técnica" para usar os streams padrão com:

```
freopen("input.txt", "r", stdin);
freopen("output.txt", "w", stdout);
```

COMPUTAÇÃO
RAIZ

Representação de números: sistema decimal

Base do sistema: 10

Valor posicional de um algarismo: $n_i \times 10^i$

- **n = algarismo**
- **i = posição (da direita para esquerda, inicia em 0)**
- **n_i = algarismo na posição i**

Quadro de Valor Posicional

Unidades de Trilhões			Unidades de Bilhões			Unidades de Milhões			Unidades de Milhar			Unidades Simples			← Classes
15ª	14ª	13ª	12ª	11ª	10ª	9ª	8ª	7ª	6ª	5ª	4ª	3ª	2ª	1ª	← Ordens
14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	← Posição
Centenas de trilhões	Dezenas de trilhões	Unidades de trilhões	Centenas de bilhões	Dezenas de bilhões	Unidades de bilhões	Centenas de milhões	Dezenas de milhões	Unidades de milhões	Centenas de milhar	Dezenas de milhar	Unidades de milhar	Centenas	Dezenas	Unidades	
								1	4	8	3	0	0	7	

(1 x 10^6) + (4 x 10^5) + (8 x 10^4) + (3 x 10^3) + (0 x 10^2) + (0 x 10^1) + (7 x 10^0)
1.000.000 + 400.000 + 80.000 + 3.000 + 0 + 0 + 7
1.483.007

Decomposição numérica:

$$(n_i \times 10^i) + (n_{i-1} \times 10^{i-1}) + (n_{i-2} \times 10^{i-2}) + \ldots + (n_0 \times 10^0)$$

COMPUTAÇÃO RAIZ

Representação de números: sistema binário

Base do sistema: 2

Valor posicional de um algarismo: $n_i \times 2^i$

- **n = algarismo**
- **i = posição (da direita para esquerda, inicia em 0)**
- **n_i = algarismo na posição i**

Quadro de Valor Posicional

2								1								← Bytes
16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	← bits
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	← Posição
(32768) Trinta e dois mil setecentos e sessenta e oito	(16384) Dezesseis mil trezentos e oitenta e quatro	(8192) Oito mil dento e noventa e dois	(4096) Quatro mil e noventa e seis	(2048) Dois mil e quarenta e oito	(1024) Mil e vinte e quatro	(512) Quinhentos e doze	(256) Duzentos e cinqüenta e seis	(128) Cento e vinte e oito	(64) Sessenta e quatro	(32) Trinta e dois	(16) Dezesseis	(8) Oito	(4) Quatro	(2) Dois	(1) Um	
						1	0	0	0	0	1	0	1	0	1	

(1 x 2^9) + (0 x 2^8) + (0 x 2^7) + (0 x 2^6) + (0 x 2^5) + (1 x 2^4) + (0 x 2^3) + (1 x 2^2) + (0 x 2^1) + (1 x 2^0)
512 + 0 + 0 + 0 + 0 + 16 + 0 + 4 + 0 + 1
533

Decomposição numérica:

$$(n_i \times 2^i) + (n_{i-1} \times 2^{i-1}) + (n_{i-2} \times 2^{i-2}) + \ldots + (n_0 \times 2^0)$$

COMPUTAÇÃO RAIZ

Representação de números: sistema octal

Base do sistema: 8

Valor posicional de um algarismo: $n_i \times 8^i$

n = algarismo

i = posição (da direita para esquerda, inicia em 0)

n_i = algarismo na posição i

Quadro de Valor Posicional

2								1								← Bytes
16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	← bits
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	← Posição
(35.184.372.088.832) Trinta e cinco trilhões cento ...	(4.398.046.511.104) Quatro trilhões trezentos e ...	(549.755.813.888) Quinhentos e quarenta e nove ...	(68.719.476.736) Sessenta e oito bilhões setecentos ...	(8.589.934.592) Oito bilhões quinhentos e oitenta...	(1.073.741.824) Um bilhão setenta e três milhões...	(134.217.728) Cento e trinta e quatro milhões ...	(16.777.216) Dezesseis milhões setecentos e setenta...	(2.097.152) Dois milhões noventa e sete mil cento ...	(262.144) Duzentos e sessenta e dois mil cento e ...	(32.768) Trinta e dois mil setecentos e sessenta e ...	(4.096) Quatro mil e noventa e seis	(512) Quinhentos e doze	(64) Sessenta e quatro	(8) Oito	(1) Um	
							5	7	0	0	3	4	1	5	3	

(5 x 8^8) + (7 x 8^7) + (0 x 8^6) + (0 x 8^5) + (3 x 8^4) + (4 x 8^3) + (1 x 8^2) + (5 x 8^1) + (3 x 8^0)
86.889.080 + 14.680.064 + 0 + 0 + 12.288 + 2.048 + 64 + 40 + 3
98.580.587

Decomposição numérica:

$$(n_i \times 8^i) + (n_{i-1} \times 8^{i-1}) + (n_{i-2} \times 8^{i-2}) + \ldots + (n_0 \times 8^0)$$

COMPUTAÇÃO RAIZ

Representação de números: sistema hexadecimal

Base do sistema: 16

Valor posicional de um algarismo: $n_i \times 16^i$

- **n = algarismo**
- **i = posição (da direita para esquerda, inicia em 0)**
- **n_i = algarismo na posição i**

Quadro de Valor Posicional

2								1								← Bytes
16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	← bits
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	← Posição
(1.152.921.504.606.850.000) Um quintilhão, cento e...	(72.057.594.037.927.900) Setenta e dois quatrilhões...	(4.503.599.627.370.496) Quatro quatrilhões quinhentos...	(281.474.976.710.656) Duzentos e oitenta e um trilhões...	(17.592.186.044.416) Dezessete trilhões quinhentos ...	(1.099.511.627.776) Um trilhão noventa e nove bilhões...	(68.719.476.736) Sessenta e oito bilhões setecentos ...	(4.294.967.296) Quatro bilhões duzentos e noventa...	(268.435.456) Duzentos e sessenta e oito milhões ...	(16.777.216) Dezesseis milhões cetecentos e setenta ...	(1.048.576) Um milhão quarenta e oito mil quinhentos...	(65.536) Sessenta e cinco mil quinhentos e trinta ...	(4.096) Quatro mil e noventa e seis	(256) Duzentos e cinqüenta e seis	(16) Dezesseis	(1) Um	
							1	B	0	0	0	0	A	9	E	

(1 x 16^8) + (11 x 16^7) + (0 x 16^6) + (0 x 16^5) + (0 x 16^4) + (0 x 16^) + (10 x 16^2) + (9 x 16 ^1) + (14 x 16^0)
4.294.967.296 + 2.952.790.016 + 0 + 0 + 0 + 0 + 2.560 + 144 + 14
7.247.760.030

Decomposição numérica:

$$(n_i \times 16^i) + (n_{i-1} \times 16^{i-1}) + (n_{i-2} \times 16^{i-2}) + \ldots + (n_0 \times 16^0)$$

COMPUTAÇÃO RAIZ

Representação de números: sistema de base b

Base do sistema: b

Valor posicional de um algarismo: $n_i \times b^i$

n = algarismo

i = posição (da direita para esquerda, inicia em 0)

n_i = algarismo na posição i

Decomposição numérica:

$$(n_i \times b^i) + (n_{i-1} \times b^{i-1}) + (n_{i-2} \times b^{i-2}) + \ldots + (n_0 \times b^0)$$

COMPUTAÇÃO RAIZ

Representação de números: decimal x binário x hexadecimal = decorar!

Hexadecimal	Decimal	Binário	Octal
0	0	0000	000
1	1	0001	001
2	2	0010	002
3	3	0011	003
4	4	0100	004
5	5	0101	005
6	6	0110	006
7	7	0111	007
8	8	1000	010
9	9	1001	011
A	10	1010	012
B	11	1011	013
C	12	1100	014
D	13	1101	015
E	14	1110	016
F	15	1111	017

COMPUTAÇÃO
RAIZ

Grandezas decimais

Tabela 1: Grandezas de base 10 maiores do que a unidade

Fator	Prefixo	Símbolo	Número	Nome
10^{32}	-	-	100 000 000 000 000 000 000 000 000 000 000	Centena de nonilhão
10^{31}	-	-	10 000 000 000 000 000 000 000 000 000 000	Dezena de nonilhão
10^{30}	quetta	Q	1 000 000 000 000 000 000 000 000 000 000	**Nonilhão**
10^{29}	-	-	100 000 000 000 000 000 000 000 000 000	Centena de octilhão
10^{28}	-	-	10 000 000 000 000 000 000 000 000 000	Dezena de octilhão
10^{27}	ronna	R	1 000 000 000 000 000 000 000 000 000	**Octilhão**
10^{26}	-	-	100 000 000 000 000 000 000 000 000	Centena de septilhão
10^{25}	-	-	10 000 000 000 000 000 000 000 000	Dezena de septilhão
10^{24}	yotta	Y	1 000 000 000 000 000 000 000 000	**Septilhão**
10^{23}	-	-	100 000 000 000 000 000 000 000	Centena de sextilhão
10^{22}	-	-	10 000 000 000 000 000 000 000	Dezena de sextilhão
10^{21}	zetta	Z	1 000 000 000 000 000 000 000	**Sextilhão**
10^{20}	-	-	100 000 000 000 000 000 000	Centena de quintilhão
10^{19}	-	-	10 000 000 000 000 000 000	Dezena de quintilhão
10^{18}	exa	E	1 000 000 000 000 000 000	**Quintilhão**
10^{17}	-	-	100 000 000 000 000 000	Centena de quatrilhão
10^{16}	-	-	10 000 000 000 000 000	Dezena de quatrilhão
10^{15}	peta	P	1 000 000 000 000 000	**Quatrilhão**
10^{14}	-	-	100 000 000 000 000	Centena de trilhão
10^{13}	-	-	10 000 000 000 000	Dezena de trilhão
10^{12}	tera	T	1 000 000 000 000	**Trilhão**
10^{11}	-	-	100 000 000 000	Centena de bilhão
10^{10}	-	-	10 000 000 000	Dezena de bilhão
10^{9}	giga	G	1 000 000 000	**Bilhão**
10^{8}	-	-	100 000 000	Centena de milhão
10^{7}	-	-	10 000 000	Dezena de milhão
10^{6}	mega	M	1 000 000	**Milhão**
10^{5}	-	-	100 000	Centena de milhar
10^{4}	-	-	10 000	Dezena de milhar
10^{3}	kilo	k	1000	**Milhar**
10^{2}	hecto	h	100	Centena
10^{1}	deca	da	10	Dezena
10^{0}	-	-	1	**Unidade**

Tabela 2: Grandezas de base 10 menores do que a unidade

Fator	Prefixo	Símbolo	Número	Nome
10^{0}	-	-	1	**Unidade**
10^{-1}	deci	d	0,1	Décimo
10^{-2}	centi	c	0,01	Centésimo
10^{-3}	mili	m	0,001	**Milésimo**
10^{-4}	-	-	0,0001	Décimo de milésimo
10^{-5}	-	-	0,000 01	Centésimo de milésimo
10^{-6}	micro	µ	0,000 001	**Milionésimo**
10^{-7}	-	-	0,000 000 1	Décimo de milionésimo
10^{-8}	-	-	0,000 000 01	Centésimo de milionésimo
10^{-9}	nano	n	0,000 000 001	**Bilionésimo**
10^{-10}	-	-	0,000 000 000 1	Décimo de bilionésimo
10^{-11}	-	-	0,000 000 000 01	Centésimo de bilionésimo
10^{-12}	pico	p	0,000 000 000 001	**Trilionésimo**
10^{-13}	-	-	0,000 000 000 000 1	Décimo de trilionésimo
10^{-14}	-	-	0,000 000 000 000 01	Centésimo de trilionésimo
10^{-15}	femto	f	0,000 000 000 000 001	**Quatrilionésimo**
10^{-16}	-	-	0,000 000 000 000 000 1	Décimo de quatrilionésimo
10^{-17}	-	-	0,000 000 000 000 000 01	Centésimo de quatrilionésimo
10^{-18}	atto	a	0,000 000 000 000 000 001	**Quintilionésimo**
10^{-19}	-	-	0,000 000 000 000 000 000 1	Décimo de quintilionésimo
10^{-20}	-	-	0,000 000 000 000 000 000 01	Centésimo de quintilionésimo
10^{-21}	zepto	z	0,000 000 000 000 000 000 001	**Sextilionésimo**
10^{-22}	-	-	0,000 000 000 000 000 000 000 1	Décimo de sextilionésimo
10^{-23}	-	-	0,000 000 000 000 000 000 000 01	Centésimo de sextilionésimo
10^{-24}	yocto	y	0,000 000 000 000 000 000 000 001	**Septilionésimo**
10^{-25}	-	-	0,000 000 000 000 000 000 000 000 1	Décimo de septilionésimo
10^{-26}	-	-	0,000 000 000 000 000 000 000 000 01	Centésimo de septilionésimo
10^{-27}	ronto	r	0,000 000 000 000 000 000 000 000 001	**Octilionésimo**
10^{-28}	-	-	0,000 000 000 000 000 000 000 000 000 1	Décimo de octilionésimo
10^{-29}	-	-	0,000 000 000 000 000 000 000 000 000 01	Centésimo de octilionésimo
10^{-30}	quecto	q	0,000 000 000 000 000 000 000 000 000 001	**Nonilionésimo**
10^{-31}	-	-	0,000 000 000 000 000 000 000 000 000 000 1	Décimo de nonilionésimo
10^{-32}	-	-	0,000 000 000 000 000 000 000 000 000 000 01	Centésimo de nonilionésimo

COMPUTAÇÃO RAIZ

Grandezas decimais e binárias

Tabela 3: Resumo de ordens de grandeza padronizadas no SI

Fator	Prefixo	Símbolo	Nome
10^{30}	quetta	Q	Nonilhão
10^{27}	ronna	R	Octilhão
10^{24}	yotta	Y	Septilhão
10^{21}	zetta	Z	Sextilhão
10^{18}	exa	E	Quintilhão
10^{15}	peta	P	Quatrilhão
10^{12}	tera	T	Trilhão
10^{9}	giga	G	Bilhão
10^{6}	mega	M	Milhão
10^{3}	kilo	k	Milhar
10^{2}	hecto	h	Centena
10^{1}	deca	da	Dezena
10^{0}	-	-	Unidade
10^{-1}	deci	d	Décimo
10^{-2}	centi	c	Centésimo
10^{-3}	mili	m	Milésimo
10^{-6}	micro	µ	Milionésimo
10^{-9}	nano	n	Bilionésimo
10^{-12}	pico	p	Trilionésimo
10^{-15}	femto	f	Quatrilionésimo
10^{-18}	atto	a	Quintilionésimo
10^{-21}	zepto	z	Sextilionésimo
10^{-24}	yocto	y	Septilionésimo
10^{-27}	ronto	r	Octilionésimo
10^{-30}	quecto	q	Nonilionésimo

Algarismos Significativos (https://archive.org/details/aasf_sigfigs_20210126)

Tabela 4: Ordens de grandeza para potências de base 2

Fator	Prefixo	Símbolo
2^{0}	-	-
2^{10}	kibi	Ki
2^{20}	mebi	Mi
2^{30}	gibi	Gi
2^{40}	tebi	Ti
2^{50}	pebi	Pi
2^{60}	exbi	Ei
2^{70}	zebi	Zi
2^{80}	yobi	Yi
2^{90}	robi	Ri
2^{100}	quebi	Qi

Algarismos Significativos (https://archive.org/details/aasf_sigfigs_20210126)

COMPUTAÇÃO RAIZ

Grandezas decimais e binárias

	Decimal		
	Fator	**Prefixo (símbolo)**	**Valor**
Unidade	10^{0}		1
Milhar	10^{3}	kilo (k)	1.000
Milhão	10^{6}	mega (M)	1.000.000
Bilhão	10^{9}	giga (G)	1.000.000.000
Trilhão	10^{12}	tera (T)	1.000.000.000.000
Quatrilhão	10^{15}	peta (P)	1.000.000.000.000.000
Quintilhão	10^{18}	exa (E)	1.000.000.000.000.000.000
Sextilhão	10^{21}	zetta (Z)	1.000.000.000.000.000.000.000
Septilhão	10^{24}	yotta (Y)	1.000.000.000.000.000.000.000.000
Octilhão	10^{27}	ronna (R)	1.000.000.000.000.000.000.000.000.000
Nonilhão	10^{30}	quetta (Q)	1.000.000.000.000.000.000.000.000.000.000

	Binário		
	Fator	**Prefixo (símbolo)**	**Valor**
Unidade	2^{0}		1
Milhar	2^{10}	kibi (Ki)	1.024
Milhão	2^{20}	mebi (Mi)	1.048.576
Bilhão	2^{30}	gibi (Gi)	1.073.741.824
Trilhão	2^{40}	tebi (Ti)	1.099.511.627.776
Quatrilhão	2^{50}	pebi (Pi)	1.125.899.906.842.624
Quintilhão	2^{60}	exbi (Ei)	1.152.921.504.606.846.976
Sextilhão	2^{70}	zebi (Zi)	1.180.591.620.717.411.303.424
Septilhão	2^{80}	yobi (Yi)	1.208.925.819.614.629.174.706.176
Octilhão	2^{90}	robi (Ri)	1.237.940.039.285.380.274.899.124.224
Nonilhão	2^{100}	quebi (Qi)	1.267.650.600.228.229.401.496.703.205.376

COMPUTAÇÃO
RAIZ

Inteiros Unsigned x Inteiros Signed

Unsigned (sem sinal):

São números binários sem nenhuma informação sobre o sinal (+ ou -), ou seja, nenhum dos bits do número representa o sinal, todos os bits são utilizados para os números. A interpretação é simples: são números não negativos.

Signed (com sinal):

São números binários com informação sobre o sinal (+ ou -), ou seja, um dos bits do número representa o sinal e os outros bits são utilizados para os números. A interpretação desses números depende da NOTAÇÃO utilizada. Sem saber qual a notação, não é possível identificar qual é o número. Notações:

- Sinal/Magnitude
- Complemento de 1
- Complemento de 2
- Notação Enviesada (notação de exesso)

COMPUTAÇÃO RAIZ

Bit mais/menos significativo; Byte mais/menos significativo

msb: **m**ost **s**ignificant **b**it (bit mais significativo)
É o bit mais à **esquerda** em um número binário,
pois tem o **maior valor posicional**.

lsb: **l**east **s**ignificant **b**it (bit menos significativo)
É o bit mais à **direita** em um número binário,
pois tem o **menor valor posicional**.

msB: **m**ost **s**ignificant **B**yte (byte mais significativo)
É o byte mais à **esquerda** em um número binário
com mais de 1 byte, pois tem o **maior valor posicional**.

lsB: **l**east **s**ignificant **B**yte (byte menos significativo)
É o byte mais à **direita** em um número binário
com mais de 1 byte, pois tem o **menor valor posicional**.

01010101

01010101 00110011 11110000 11001100

Total de números e faixa numérica

O total de números que podem ser representados com n bits, e a faixa numérica (menor e maior) desses números, depende se estamos considerando binários unsigned ou signed.

Para binários unsigned com n bits, temos o seguinte:

- **Total de números:** 2^n
- **Faixa numérica:** $[0; 2^n - 1]$

Para binários signed com n bits, depende da notação:

- **Para Complemento de 2:**
  - **Total de números:** 2^n
  - **Faixa numérica:** $[-2^{n-1} \; ; \; 2^{n-1} - 1]$

COMPUTAÇÃO RAIZ

Tipos de dados inteiros

$$(\text{short int}) \leq (\text{int}) \leq (\text{long int}) \leq (\text{long long int})$$

```
signed short int:
    tamanho (bits): 16
    valor mínimo  : -32768
    valor máximo  : +32767

signed int:
    tamanho (bits): 32
    valor mínimo  : -2147483648
    valor máximo  : +2147483647

signed long int:
    tamanho (bits): 64
    valor mínimo  : -9223372036854775808
    valor máximo  : +9223372036854775807

signed long long int:
    tamanho (bits): 64
    valor mínimo  : -9223372036854775808
    valor máximo  : +9223372036854775807
```

```
unsigned short int:
    tamanho (bits): 16
    valor mínimo  : 0
    valor máximo  : 65535

unsigned int:
    tamanho (bits): 32
    valor mínimo  : 0
    valor máximo  : 4294967295

unsigned long int:
    tamanho (bits): 64
    valor mínimo  : 0
    valor máximo  : 18446744073709551615

unsigned long long int:
    tamanho (bits): 64
    valor mínimo  : 0
    valor máximo  : 18446744073709551615
```

COMPUTAÇÃO RAIZ

Tipos de dados inteiros

A biblioteca <`limits.h`> define diversas constantes simbólicas úteis que podem ser usadas nos programas. Os limites MÍNIMOS do padrão C estão abaixo; a implementação utiliza os valores "reais".

```
#define CHAR_BIT                          8
#define CHAR_MAX    UCHAR_MAX or SCHAR_MAX
#define CHAR_MIN             0 or SCHAR_MIN
#define INT_MAX                     +32767
#define INT_MIN                     -32767
#define LONG_MAX               +2147483647
#define LONG_MIN               -2147483647
#define LLONG_MAX     +9223372036854775807
#define LLONG_MIN     -9223372036854775807
#define MB_LEN_MAX                        1
#define SCHAR_MAX                      +127
#define SCHAR_MIN                      -127
#define SHRT_MAX                     +32767
#define SHRT_MIN                     -32767
#define UCHAR_MAX                       255
#define USHRT_MAX                     65535
#define UINT_MAX                      65535
#define ULONG_MAX               4294967295
#define ULLONG_MAX   18446744073709551615
```

COMPUTAÇÃO RAIZ

Aritmética Modular

Nós denotamos por "`x mod m`" o resto `r` da divisão de `x`, o dividendo, por *m*, o divisor, sendo `q` o quociente da divisão.

Aritmética modular é muito importante! Veremos isso em muito mais profundiade.

$$x = mq + r$$

Algumas vezes estamos trabalhando com números muito grandes e o problema pede para que a solução seja "`mod m`", o resto quando a resposta é dividida por `m`. Algumas propriedades:

```
(a + b) mod m = (a mod m + b mod m) mod m
(a - b) mod m = (a mod m - b mod m) mod m
(a * b) mod m = (a mod m * b mod m) mod m
```

Na adição, subtração e multiplicação, o resto pode ser tomado antes da operação e asim os números nunca serão muito grandes e será suficiente usar int ou long long int.

COMPUTAÇÃO RAIZ

Números de ponto flutuante

Podemos usar `float`, `long` e `long double`.

A maioria dos números reais não possui representação exata com os tipos de ponto flutuante e, portanto, não é possível compará-los diretamente (muito risco de erros de exatidão). A melhor maneira de comparar números de ponto flutuante é considerar que eles são iguais se a diferença entre eles for menor do que um valor EPSILON, onde EPSILON é um número muito pequeno (geralmente 1e-9).

Os problemas geralmente indicam a exatidão desejada!

```c
#include <stdio.h>
#include <math.h>

#define EPS 1e-9

int main (void)
{
    double d = (0.3 * 3) + 0.1;

    if (d == 1.0)
        printf("Iguais\n");
    else
        printf("Diferentes\n");

    /**
     * Em <math.h>:
     *   FABSL: para long double
     *   FABS : para double
     *   FABSF: para float
     */
    if (fabs(d - 1.0) < EPS)
        printf("Iguais\n");
    else
        printf("Diferentes\n");
}
```

Obs.: as funções ABS, para inteiros, estão em **`<stdlib.h>`**

COMPUTAÇÃO RAIZ

Números de ponto flutuante

-A biblioteca <`floats.h`> define diversas constantes simbólicas para trabalhar com ponto flutuante mas, em geral, não são necessárias. O tamanhos abaixo são importantes:

$$(\text{float}) \leq (\text{double}) \leq (\text{long double})$$

Tipo	Menor Negativo	Maior Negativo	Zero	Zero	Menor Positivo	Maior Positivo
float	$-3.4028234664 \times 10^{38}$	$-1.1754943508 \times 10^{-38}$	-0.0	0.0	$1.1754943508 \times 10^{-38}$	$3.4028234664 \times 10^{38}$
double	$-1.7976931349 \times 10^{308}$	$-2.2250738585 \times 10^{-308}$	-0.0	0.0	$2.2250738585 \times 10^{-308}$	$1.7976931349 \times 10^{308}$
long double	$-1.1897314954 \times 10^{4932}$	$-3.3621031431 \times 10^{-4932}$	-0.0	0.0	$3.3621031431 \times 10^{-4932}$	$1.1897314954 \times 10^{4932}$

COMPUTAÇÃO RAIZ

Números inteiros em ponto flutuante

Em geral é suficiente usar `long long int` para os números inteiros mas em algumas situações onde temos que trabalhar com números inteiros muito grandes, podemos usar o tipo `double` para essa tarefa.

Inteiros até aproximadamente 2^(53) podem ser representados corretamente usando doubles.

Encurtamento de código

Importante pois o código deve ser digitado o mais rápido possível. Algumas técnicas:

a) Uso de `typedef`: já vimos (templates C/C++)

b) Uso de `macros`: instruções ao pré-processador, que nos permite criar:

- **macros de objeto (onstantes simbólicas)**
- **macros de função**
- **macros variádicas**
- **macros de controle condicional**
- **etc**

COMPUTAÇÃO RAIZ

Encurtamento de código

```c
#include <stdio.h>

// Macros de objetos
#define PI 3.14159265358979323846
#define EPS 1e-9

// Macros de função
#define REP(a, b) for (int i = a; i < b; i++)
#define AOQ(a) ((a)*(a))
#define MAX(a, b) ((a) > (b) ? (a) : (b))

// Macros variádicas
#define LOG(fmt, ...) printf("LOG: " fmt "\n", __VA_ARGS__)

// Macros de controle condicional
#ifdef DEBUG
  #define DBG_PRINT(msg) printf("DEBUG: %s\n", msg)
#else
  #define DBG_PRINT(msg) /* nada */
#endif

int main (void)
{
    REP(0, 10){
        printf("%d\n", AOQ(i));
    }

    LOG("%s %s", "ERRO!", "Deu tudo errado.");
}
```

Cuidado com a expansão que o pré-processador faz!
É importante colocar os parênteses ao redor da expressão inteira e ao redor dos parâmetros.

DESVANTAGENS:

- **não verifica tipo**
- **difícil debugar**
- **legibilidade**
- **escopo global**

COMPUTAÇÃO RAIZ

Algumas fórmulas matemáticas importantes

Somatórios:

Todo somatório da forma

$$\sum_{x=1}^{n} x^k = 1^k + 2^k + 3^k + \ldots + n^k$$

onde k é um inteiro positivo, tem uma fórmula fechada que é um polinômio de grau k+1. Exemplos:

$$\sum_{x=1}^{n} x = 1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2}$$

$$\sum_{x=1}^{n} x^2 = 1^2 + 2^2 + 3^2 + \ldots + n^2 = \frac{n(n+1)(2n+1)}{6}$$

COMPUTAÇÃO RAIZ

Algumas fórmulas matemáticas importantes

Progressão aritmética:
É uma seqüência de números onde a diferença entre quaisquer dois números consecutivos é constante. Exemplo:

3, 7, 11, 15

O somatório de uma progressão aritmética entre a e b (incluindo esses números) pode ser obtido usando-se a fórmula:

$$\underbrace{a + \cdots + b}_{n \text{ numbers}} = \frac{n(a+b)}{2}$$

a = primeiro número
b = último número
n = quantidade de números

COMPUTAÇÃO RAIZ

Algumas fórmulas matemáticas importantes

Progressão geométrica:
É uma seqüência de números onde a razão entre quaisquer dois números consecutivos é constante. Exemplo:

3, 6, 12, 24

O somatório de uma progressão geométria entre a e b (incluindo esses números) pode ser obtido usando-se a fórmula:

$$a + ak + ak^2 + \cdots + b = \frac{bk - a}{k - 1}$$

a = primeiro número
b = último número
k = razão

Caso especial:
$1 + 2 + 4 + 8 + \ldots + 2^{n-1} = 2^n - 1$

COMPUTAÇÃO RAIZ

Algumas fórmulas matemáticas importantes

Logarítmo:

O logarítmo de x na base k é o expoente que se deve elevar a base para obter x.

$$\log_k(x) = a \qquad\qquad k^a = x$$

Propriedades interessantes:

- É o número de vezes que devemos dividir x por k até obter 1

$$\log_2(32) = 5 \qquad 32 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$$

- Outras fórmulas:

$$\log_k(ab) = \log_k(a) + \log_k(b) \qquad \log_k(x^n) = n \cdot \log_k(x)$$

$$\log_k\left(\frac{a}{b}\right) = \log_k(a) - \log_k(b) \qquad \log_u(x) = \frac{\log_k(x)}{\log_k(u)}$$

COMPUTAÇÃO RAIZ

Algumas fórmulas matemáticas importantes

Lógica:

Não se esqueça da lógica proposicional:

A	B	$\neg A$	$\neg B$	$A \wedge B$	$A \vee B$	$A \Rightarrow B$	$A \Leftrightarrow B$
0	0	1	1	0	0	1	1
0	1	1	0	0	1	1	0
1	0	0	1	0	1	0	0
1	1	0	0	1	1	1	1

Não se esqueça da lógica de predicados:

- se x, y, a, b são números inteiros, e P(x) é verdadeiro se x for primo:

$$\forall x(\exists y(y < x))$$

$$\forall x((x > 1 \wedge \neg P(x)) \Rightarrow (\exists a(\exists b(a > 1 \wedge b > 1 \wedge x = ab))))$$

COMPUTAÇÃO RAIZ

Funções importantes

Piso e Teto:

- Piso: é o maior inteiro menor do que x $\lfloor x \rfloor$
- Teto: é o menor inteiro maior do que x $\lceil x \rceil$

Cuidado com o piso e o teto de números negativos!

$$\lfloor 3/2 \rfloor = 1 \qquad \lceil 3/2 \rceil = 2$$

Min e Max:

- precisa explicar?

Fatorial:

- O fatorial de n pode ser definido de forma iterativa ou recursiva:

Cuidado com overflow ao calcular fatorial de números grandes!

$$\prod_{x=1}^{n} x = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$$

$$\begin{aligned} 0! &= 1 \\ n! &= n \cdot (n-1)! \end{aligned}$$

COMPUTAÇÃO RAIZ

Funções importantes

Fibonacci:

- f(n) é definido recursivamente:

$$
\begin{aligned}
f(0) &= 0 \\
f(1) &= 1 \\
f(n) &= f(n-1) + f(n-2)
\end{aligned}
$$

mas pode ser implementado através de algoritmos iterativos também, já que a implementação recursiva simples é O(2^n).

- f(n) pode ser encontrado diretamente pela fórmula de Binet:

$$
f(n) = \frac{(1+\sqrt{5})^n - (1-\sqrt{5})^n}{2^n\sqrt{5}}
$$

COMPUTAÇÃO RAIZ

Conjuntos

É uma coleção de elementos não ordenados e não repetidos.

$$X = \{2, 4, 7\}$$

Importante:

$\emptyset$	**conjunto vazio**		
$\lvert S \rvert$	**tamanho do conjunto (quantidade de elementos)**		
$x \in S$	**pertence**	$x \notin S$	**não pertence**
$A \cap B$	**interseção**	$A \cup B$	**união**
$\bar{A}$	**complemento**	$A \setminus B = A \cap \bar{B}$	**diferença**
$A \subset S$	**está contido**		
$2^{\lvert S \rvert}$	**quantidade de subconjuntos (inclui vazio e S). Ex.:** $\{2, 4, 7\}$		

$$\emptyset,\ \{2\},\ \{4\},\ \{7\},\ \{2,4\},\ \{2,7\},\ \{4,7\},\ \{2,4,7\}$$

COMPUTAÇÃO RAIZ

Habilidades BÁSICAS de processamento de conjuntos

Muitos problemas da maratona (e da computação em geral) exigem que você consiga processar e obter todos os subconjuntos ou permutações dos elementos de um conjunto.

Solucione todas as 4 tarefas que são correspondentes às 4 habilidades básicas do processamento de conjuntos.

NÃO CONTINUE ANTES DE REALIZAR TODAS ESSAS TAREFAS NA ORDEM EXATA EM QUE ESTÃO. DOMINE ESSE CONHECIMENTO!

Tente fazer por conta própria, se esforce e, se não conseguir, peça ajuda.

Habilidades BÁSICAS de processamento de conjuntos

HABILIDADE 1:

Na computação é muito comum o problema de ***gerar todos os subconjuntos*** de um determinado conjunto com n elementos. Sabemos que a quantidade de subconjuntos é igual a $2^{|n|}$, ou seja, 2 elevado ao número de elementos do conjunto (incluindo o conjunto vazio e o próprio conjunto como um todo). Sua tarefa é descobrir, para C++ ou C (a linguagem de sua escolha), um algoritmo recursivo capaz de gerar todos os subconjuntos de um conjunto de n elementos. Por exemplo: dado o conjunto A = {1, 2, 3}, como $n = 3$, temos $2^3 = 8$ subconjuntos, que são:

```
{}
{1}
{2}
{3}
{1, 2}
{1, 3}
{2, 3}
{1, 2, 3}
```

Programe e execute um algoritmo recursivo para isso e verifique se:

a) Seu programa está funcionando corretamente
b) Você sabe explicar como o algoritmo funciona
c) Você sabe a complexidade (Big-O) do algoritmo

HABILIDADE 2:

Outra habilidade importante na computação é a capacidade de ***gerar todas as permutações*** de uma coleção de n elementos. Suponha que você tenha a coleção (1, 2, 3) e quer gerar todas as permutações dessa coleção. Fazendo manualmente encontramos 6 permutações:

```
(1, 2, 3)
(1, 3, 2)
(2, 1, 3)
(2, 3, 1)
(3, 1, 2)
(3, 2, 1)
```

Ocorre que há um problema: ao fazer as permutações temos que separar 2 casos diferentes: a) se a coleção não tem elementos repetidos, e b) se a coleção tem elementos repetidos.

- Se a coleção **não tem** elementos repetidos, o número total de permutações é dado por $P(n)$, onde:

$$P(n) = n!$$

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1$$

- Se a coleção **tem um ou mais elementos repetidos**, o número total de permutações é dado por $P(n; n_1, n_2, \ldots, n_k)$, onde:

$$P(n; n_1, n_2, \ldots, n_k) = \frac{n!}{n_1! \times n_2! \times \cdots \times n_k!}$$

n = número total de elementos
$n_1, n_2, \ldots, n_k$ = quantidades de repetições de cada elemento

Por exemplo: a coleção (AABB) terá apenas 6 permutações:
(AABB), (ABAB), (ABBA), (BAAB), (BABA), (BBAA), (ABBA)

COMPUTAÇÃO RAIZ

Dúvidas?

Foto de Towfiqu Barbhuiya, no Unsplah (https://unsplash.com/photos/a-blue-question-mark-on-a-pink-background-oZuBNC-6E2s)

COMPUTAÇÃO RAIZ