Apostila Maratona de Programação

Grupo de Robótica e Maratona de Programação
Universidade Federal de Rondônia

2019

Lista de Figuras

Lista de Algoritmos

Sumário

1 Introdução

A Maratona de Programação é um evento da Sociedade Brasileira de Computação que existe desde o ano de 1996. A Maratona nasceu das competições regionais classificatórias para as finais mundiais do concurso de programação, o International Collegiate Programming Contest, e é parte da regional sulamericana do concurso.

Ela se destina a alunos e alunas de cursos de graduação e início de pós-graduação na área de Computação e afins (Ciência da Computação, Engenharia de Computação, Sistemas de Informação, Matemática, etc). A competição promove nos estudantes a criatividade, a capacidade de trabalho em equipe, a busca de novas soluções de software e a habilidade de resolver problemas sob pressão. De ano em ano temos observado que as instituições e principalmente as grandes empresas da área têm valorizado os alunos que participam da Maratona.

Várias universidades do Brasil desenvolvem concursos locais para escolher os melhores times para participar da Maratona de Programação. Estes times competem na Maratona (e portanto na regional sulamericana) de onde os melhores serão selecionados para participar das Finais Mundiais do evento. No ano de 2018, mais de 50 mil estudantes de mais de 3000 escolas de mais de 100 países competiram em regionais em todo o planeta, e apenas 135 (cerca de 1%) participam das Finais Mundiais do evento, no Porto, Portugal. Seis times brasileiros, dos quase de 800 participantes, estarão presentes nas finais mundiais.

Os times são compostos por três estudantes, que tentarão resolver durante 5 horas o maior número possível dos 10 ou mais problemas que são entregues no início da competição. Estes estudantes têm à sua disposição apenas um computador e material impresso (livros, listagens, manuais) para vencer a batalha contra o relógio e os problemas propostos.

Os competidores do time devem colaborar para descobrir os problemas mais fáceis, projetar os testes, e construir as soluções que sejam aprovadas pelos juízes da competição. Alguns problemas requerem apenas compreensão, outros conhecimento de técnicas mais sofisticadas, e alguns podem ser realmente muito difíceis de serem resolvidos.

O julgamento é estrito. No início da competição os competidores recebem os problemas que devem ser resolvidos. Nos enunciados dos problemas constam exemplos dos dados dos problemas, mas eles não têm acesso às instâncias testadas pelos juízes. A cada submissão incorreta de um problema (ou seja, que deu resposta incorreta a uma das instâncias dos juízes) é atribuída uma penalidade de tempo. O time que conseguir resolver o maior número de problemas (no menor tempo acumulado com as penalidades, caso haja empate) é declarado o vencedor[1].

1.1 Dicas para a competição

Conforme você leu, a maratona de programação geralmente ocorre no período de 5 horas ininterruptas e seu objetivo é a resolução do maior número de problemas. Como a competição ocorre em um longo tempo lhe é aconselhável um preparo físico e mental para que você suporte o estresse da competição. Crie uma rotina de programação. Inicie com alguns minutos, passe para horas e logo você estará programando por horas a fio. Reuniões antes da competição com os integrantes do time auxiliam no entrosamento da equipe. Lembre-se sempre, está é uma competição em equipe. Não tente resolver tudo sozinho pois você irá perder tempo. Existem plataformas online que simulam a competição (também chamada de Contest) ou possuem um grande acervo de problemas relacionados a ela. Podemos mencionar algumas plataformas, tais como:

- Codeforces
- URI Online Judge
- UVA Online Judge

Na maratona de programação cada problema tem uma cor relacionado a ele. Quando você consegue solucionar um problema lhe é dado o balão com a cor do problema. Utilize isso a seu favor. Quando a

[1]Está introdução foi retirada do site oficial da Maratona de Programação. `http://maratona.ime.usp.br/`

competição é iniciada procure olhar os 'Standings' gerais. Busque resolver os problemas mais fáceis. Olhe ao redor e observe a frequência de aparição das cores, observe a figura 1. Este é o método mais fácil para descobrir se um problema é fácil ou não. Se nenhuma equipe conter a cor do balão associado ao exercício o qual você julga ser fácil, revise o problema pois ele pode não ser tão simples assim.

Figura 1: Balões Maratona de Programação

Fonte: Maratona de Programação do Norte, 2019.

Nas próximas seções serão abordados técnicas e problemas que comumente aparecem na competição e algumas das várias maneiras para aborda-los.

1.2 Análise de Algoritmos

A eficiência dos algoritmos é importante na programação competitiva. Existem diversos problemas em que podemos facilmente supor que ele é um problema fácil. Mas usualmente, é fácil codar[2] um algoritmo com técnicas que resolvam o problema lentamente, mas o real desafio é inventar um algoritmo rápido. Se o algoritmo é muito lento, ele só terá pontos parciais ou nenhum ponto. Porém, ao pensarmos que é um problema fácil acabamos por aborda-lo com uma solução *naive* (ingênua) por não termos o conhecimento suficiente de análise de algoritmos.

A análise de algoritmos, estuda certos problemas computacionais recorrentes, ou seja, problemas que aparecem, sob diversos disfarces, em uma grande variedade de aplicações e de contextos. A análise de um algoritmo para um dado problema trata de:

1. Provar que o algoritmo e a técnica utilizada está correta[3];
2. Estimar o tempo que a execução do algoritmo consome;
3. A estimativa do espaço de memória usado pelo algoritmo também é importante em muitos casos.

Conforme a literatura explana, existem três tipos de notações assintóticas estudadas na computação. Sendo elas:

- Ω - Omega

[2]Significado de codar: O ato criar códigos; programar.
[3]Utilizam-se vários métodos matemáticos, como: Indução; recorrência, etc.

- Θ - Theta
- O - Big O

Onde Ω é conhecido como o melhor caso possível, Θ caso médio, e O como pior caso. Abordaremos somente a notação Big O. Pois a análise via Big O nos permite encontrar o melhor algoritmo possível para o pior caso.

1.2.1 Regras básicas para a perspectiva de cálculo de complexidade

A complexidade de tempo de um algoritmo é denotada como O (...), onde os três pontos representam alguma função. Normalmente, a variável n indica o tamanho da entrada. Por exemplo, se a entrada for uma matriz de números, n será o tamanho da matriz e, se a entrada é uma string, n será o tamanho da string.

Loops

Por exemplo, o tempo de complexidade do código abaixo é O(n):

Algoritmo 1: Loop em O(n)

```
for(int i = 0; i < n; i++){
  //code
}
```

E o tempo de complexidade do código abaixo é O(n^2):

Algoritmo 2: Loop em O(n^2)

```
for(int i = 0; i < n; i++){
  for(int j = 0; j < n; j++){
    //code
  }
}
```

Recursão

A complexidade de tempo de uma função recursiva depende do número de vezes a função é chamada e a complexidade de tempo de uma única chamada. O tempo total complexidade é o produto desses valores.

Por exemplo, considere a seguinte função:

Algoritmo 3: Função f

```
void f(int n){
  if(n == 1) return;
  f(n - 1);
}
```

A chamada f(n) causa n chamadas de função e a complexidade de tempo de cada chamada é O(1). Assim, a complexidade total do tempo é O(n).

Como outro exemplo, considere a seguinte função:

Algoritmo 4: Função g

```
void g(int n){
  if(n == 1) return;
  g(n - 1);
  g(n - 1);
}
```

Neste caso, cada chamada de função gera duas outras chamadas, exceto $n = 1$. Veja o que acontece quando g é chamado com o parâmetro n. A tabela a seguir mostra as chamadas de função produzidas por esta única chamada:

Função de chamada	Número de chamadas
g(n)	1
g(n - 1)	2
g(n - 2)	3
...	...
g(1)	$2n - 1$

Tabela 1: Chamadas da função g

Com base nisso, a complexidade do tempo é: $1 + 2 + 4 + ... + 2n - 1 = 2^n$ - 1 = O(2^n).

1.2.2 Classes de complexidade

A lista a seguir contém complexidades de tempo comuns de algoritmos:

- O(1): O tempo de execução de um algoritmo de **tempo constante** não depende do tamanho de entrada. Um algoritmo típico de tempo constante é uma fórmula direta que calcula a resposta.
- O(log n): Um algoritmo **logarítmico** geralmente divide o tamanho da entrada em cada etapa. O tempo de execução de tal algoritmo é logarítmico, porque $\log_2 n$ é igual ao número de vezes n deve ser dividido por 2 para obter 1.
- O($\sqrt{n}$): Um algoritmo de **raiz quadrada** é mais lento que O(log n), mas é mais rápido que O(n). Uma propriedade especial de raízes quadradas é que $\sqrt{n}$ = n / $\sqrt{n}$, então a raiz quadrada $\sqrt{n}$ mentiras, em certo sentido, no meio da entrada.
- O(n): Um algoritmo **linear** passa pela entrada um número constante de vezes. Este é geralmente a melhor complexidade de tempo possível, porque geralmente é necessário acesse cada elemento de entrada pelo menos uma vez antes de relatar a resposta.
- O(n log n): Essa complexidade de tempo geralmente indica que o algoritmo classifica a entrada, porque a complexidade de tempo dos algoritmos de ordenação eficientes é O(n log n). Outra possibilidade é que o algoritmo use uma estrutura de dados onde cada a operação leva tempo O(log n).
- O(n^2): Um algoritmo **quadrático** geralmente contém dois loops aninhados. É possível percorrer todos os pares dos elementos de entrada no tempo O(n^2).
- O(n^3): Um algoritmo **cúbico** geralmente contém três loops aninhados. É possível ir através de todas as trincas dos elementos de entrada no tempo O(n^3).
- O(2^n): Essa complexidade de tempo geralmente indica que o algoritmo itera todos os subconjuntos dos elementos de entrada. Por exemplo, os subconjuntos de 1, 2, 3 são: ∅, 1, 2, 3, 1, 2, 1, 3, 2, 3 e 1, 2, 3.
- O(n!): Essa complexidade de tempo geralmente indica que o algoritmo itera todas as permutações dos elementos de entrada. Por exemplo, as permutações de 1, 2, 3 são: (1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2) e (3 2, 1).

Um algoritmo é polinomial se sua complexidade de tempo for no máximo O(n ^{k}) onde k é uma constante. Todas as complexidades de tempo acima, exceto O(2^n) e O(n!), são polinomiais. Na prática, a constante k é geralmente pequena e, portanto, um tempo polinomial complexidade significa que o algoritmo é eficiente.

A maioria dos algoritmos nesta apostila é polinomial. Ainda assim, existem muitos problemas para os quais nenhum algoritmo polinomial é conhecido, ou seja, ninguém sabe como resolvê-los eficientemente. Os problemas NP-difíceis são um conjunto importante de problemas, qual nenhum algoritmo polinomial é conhecido[4]

1.2.3 Soma máxima de um subvetor

Muitas vezes há vários algoritmos possíveis para resolver um problema de modo que as complexidades do tempo são diferentes. Esta seção discute um problema clássico que tem uma solução direta O(n^3). No entanto, projetando um algoritmo melhor, é possível resolver o problema no tempo O(n^2) e até no tempo O(n).

Dado um vetor de n números, nossa tarefa é calcular o máximo de soma dos raios, isto é, a maior soma possível de uma sequência de valores consecutivos no vetor 2. O problema é interessante quando pode haver valores negativos no vetor. Por exemplo, no vetor: $[-1, 2, 4, -3, 5, 2, -5, 2]$, a soma máxima desse vetor é 10.

1.2.4 Algoritmo 1

Uma maneira simples de resolver o problema é percorrer todos os sub-argumentos possíveis, calcule a soma dos valores em cada subvetor e mantenha a soma máxima. O código a seguir implementa esse algoritmo:

Algoritmo 5: Algoritmo 1 em O(n^3)

```
#include <bits/stdc++.h>
using namespace std;

int main(){
  int melhor = 0;
  for(int a = 0; a < n; a++){
    for(int b = a; b < n; b++){
      int soma = 0;
      for(int k = a; k <= b; k++){
          soma += vetor[k];
      }
      melhor = max(melhor, soma);
    }
  }
  cout << melhor << "\n";
  return 0;
}
```

As variáveis a e b fixam o primeiro e último índice do subvetor, e o soma dos valores é calculada para a soma variável. A variável melhor contém o soma máxima encontrada durante a pesquisa. A complexidade temporal do algoritmo é O(n^3), porque consiste em três loops aninhados que passam pela entrada.

1.2.5 Algoritmo 2

É fácil tornar o Algoritmo 1 mais eficiente removendo um loop dele. Isto é possível calculando a soma ao mesmo tempo quando a extremidade direita do movimentos subvetor. O resultado é o seguinte código:

Algoritmo 6: Algoritmo 2 em O(n^2)

```
#include <bits/stdc++.h>
using namespace std;

```

[4]Um livro clássico sobre o tema : M. R. Garey's and D. S. Johnson's Computers and Intractability: A Guide to the Theory of NP-Completeness

```
int main(){
  int melhor = 0;
  for(int a = 0; a < n; a++){
    int soma = 0;
    for(int b = a; b < n; b++){
      soma += vetor[b];
      melhor = max(melhor, soma);
    }
  }
  cout << melhor << "\n";
  return 0;
}
```

Após essa alteração, a complexidade do tempo é $O(n^2)$.

1.2.6 Algoritmo 3

Surpreendentemente, é possível resolver o problema em tempo $O(n)$, o que significa que apenas um loop é suficiente. A ideia é calcular, para cada posição do vetor, soma máxima de um subvetor que termina nessa posição. Depois disso, a resposta para o problema é o máximo dessas somas. Considere o subproblema de encontrar o subvetor de soma máxima que termina em posição k. Existem duas possibilidades:

1. O subvetor contém apenas o elemento na posição k;
2. O subvetor consiste em um subvetor que termina na posição k - 1, seguido por o elemento na posição k.

Neste último caso, uma vez que queremos encontrar um subvetor com soma máxima, o subvetor que termina na posição k - 1 também deve ter a soma máxima. Portanto, podemos resolver o problema de forma eficiente, calculando a soma máxima de subvetor para cada posição final da esquerda para a direita. O código a seguir implementa o algoritmo:

Algoritmo 7: Algoritmo 3 em $O(n)$

```
#include <bits/stdc++.h>
using namespace std;

int main(){
  int melhor = 0, soma = 0;
  for(int k = 0; k < n; k++){
    soma = max(vetor[k], soma + vetor[k]);
    melhor = max(melhor, soma);
  }
  cout << melhor << "\n";
  return 0;
}
```

O algoritmo contém apenas um loop que passa pela entrada, então o tempo complexidade é $O(n)$. Esta é também a melhor complexidade de tempo possível, porque qualquer algoritmo para o problema tem que examinar todos os elementos da matriz pelo menos uma vez.

1.2.7 Comparação de eficiência

É interessante estudar como os algoritmos eficientes são na prática. A seguinte tabela mostra os tempos de execução dos algoritmos acima para diferentes valores de n em um computador moderno. Em cada teste, a entrada foi gerada aleatoriamente. O tempo necessário para a leitura a entrada não foi medida.

Tamanho de n	Algoritmo 1	Algoritmo 2	Algoritmo 3
10^2	0.0 s	0.0 s	0.0 s
10^3	0.1 s	0.0 s	0.0 s
10^4	$>$ 10.0 s	0.1 s	0.0 s
10^5	$>$ 10.0 s	5.3 s	0.0 s
10^6	$>$ 10.0 s	$>$ 10.0 s	0.0 s
10^7	$>$ 10.0 s	$>$ 10.0 s	0.0 s

Tabela 2: Tempo de execução dos algoritmos

A comparação mostra que todos os algoritmos são eficientes quando o tamanho da entrada é insumos pequenos, mas maiores, trazem diferenças notáveis nos tempos de execução dos algoritmos. Algoritmo 1 torna-se lento quando $n = 10^{\ 4}$ e Algoritmo 2 torna-se lento quando $n = 10^5$. Apenas o Algoritmo 3 é capaz de processar até mesmo os maiores entradas instantaneamente.

Exercícios[5]

- **1221** - Primo Rápido
- **1424** - Problema Fácil de Rujia Liu?
- **1805** - Soma Natural

[5]Os exercícios encontram-se disponíveis na plataforma URI Online Judge. `https://www.urionlinejudge.com.br`

2 Ad hoc

2.1 Técnica dos Dois Ponteiros

2.2 Problema de Josephus

3 Estrutura de Dados

3.1 Decomposição da Raiz Quadrada

Problema: Dado um vetor de n elementos precisamos responder consultas que retornam a soma dos elementos no intervalo l para r no vetor. Além disso, o vetor não é estático, permitindo que os valores possam ser alterados por meio de alguma consulta de atualização.

Algoritmo 8: Algoritmo Decomposição da Raiz Quadrada

```cpp
#include <bits/stdc++.h>

using namespace std;

#define MAXN 10000
#define SQRSIZE 100

int arr[MAXN];       // vetor original
int block[SQRSIZE];    // vetor decomposto
int blk_sz;          // tamanho do bloco

// Complexidade : O(1)
void update(int idx, int val){
  int blockNumber = idx / blk_sz;
  block[blockNumber] += val - arr[idx];
  arr[idx] = val;
}

// Complexidade : O(sqrt(n))
int query(int l, int r){
  int sum = 0;
  while (l<r and l%blk_sz!=0 and l!=0){
    // Caminhando pelo limite esquerdo
    sum += arr[l];
    l++;
  }
  while (l+blk_sz <= r){
    // Caminhando pelos blocos menores do que o limite direito
    sum += block[l/blk_sz];
    l += blk_sz;
  }
  while (l<=r){
    // Caminhando pelo bloco do limite direito
    sum += arr[l];
    l++;
  }
  return sum;
}

// Preenche os valores na variavel input
void preprocess(int input[], int n){
  // Iniciando o index do bloco
  int blk_idx = -1;

  // Calculo do tamanho do bloco
  blk_sz = sqrt(n);
```

```
47
48    // Criando o vetor decomposto
49    for (int i=0; i<n; i++){
50      arr[i] = input[i];
51      if (i%blk_sz == 0){
52        // Entrando no proximo bloco
53        // Incrementando o valor do indice
54        blk_idx++;
55      }
56      block[blk_idx] += arr[i];
57    }
58  }
59
60
61  int main(){
62
63    int input[] = {1, 5, 2, 4, 6, 1, 3, 5, 7, 10};
64    int n = sizeof(input)/sizeof(input[0]);
65
66    preprocess(input, n);
67
68    cout << "query(3,8) : " << query(3, 8) << endl;
69    cout << "query(1,6) : " << query(1, 6) << endl;
70    update(8, 0);
71    cout << "query(8,8) : " << query(8, 8) << endl;
72    return 0;
73  }
```

Exercício

- **2800** - Brincando de Consultas

3.2 Arvore de Segmentos

Algoritmo 9: Algoritmo Arvore de Segmentos

```
#include <bits/stdc++.h>
#define endl '\n'
#define IOS ios_base::sync_with_stdio(false); cin.tie(NULL);

using namespace std;
const int x = 100000;

int l[4*x+1],h[4*x+1],delta[4*x+1],mi[4*x+1];

void init(int i,int a,int b);
void increment(int i, int a, int b, int val);
void prop(int i);
void update(int  i);
int minimum(int i, int a, int b);


int main(){
  IOS;
  int n;
  cin >> n;
  init(1, 0,n-1);

  return 0;
}

void init(int i , int a, int b){
  l[i] = a;
  h[i] = b;

  if(a == b)
    return;

  int m = (a+b)/2;
  init(2*1,a,m);
  init(2*1+1,m+1,b);

}

void prop(int i){

  delta[2*i] += delta[i];
  delta[2*i+1] += delta[i];
  delta[i] = 0;

}
void update(int i){

  mi[i] = min(mi[2*i] + delta[2*i], mi[2*i+1] + delta[2*i+1]);

}

void increment(int i, int a, int b,int val){
```

```
53
54    if(b < l[i] or a > h[i])
55      return;
56
57    if(a <= l[i]  and h[i] <= b){
58      delta[i] += val;
59      return;
60    }
61
62    prop(i);
63
64    increment(2*i,a,b,val);
65    increment(2*i+1,a,b,val);
66
67    update(i);
68
69  }
70
71  int minimum(int i, int a,int b){
72
73    if(l[i] < b or a > h[i])
74      return INT_MAX;
75
76    if(a <= l[i] and b >= h[i])
77      return mi[i]+ delta[i];
78
79    prop(i);
80    int minLeft = minimum(2*i,a,b);
81    int minRight = minimum(2*i+1,a,b);
82    update(i);
83    return min(minLeft,minRight);
84  }
```

Exercício

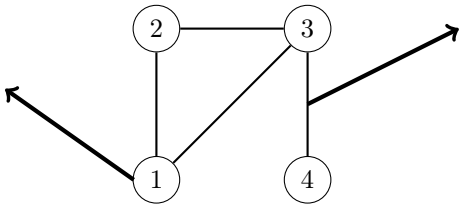
- **1301** - Produto do Intervalo

4 Grafos

Grafos são estruturas matemáticas que permitem codificar relacionamentos entre pares de objetos. As estruturas que podem ser representadas por grafos estão em toda parte e muitos problemas de interesse prático podem ser formulados como questões sobre certos grafos. Por exemplo, uma rota do metrô, e um labirinto.

Um grafo é composto por um conjunto de vértices e arestas conectando cada par de vértices com um relacionamento. Os vértices são os objetos da estrutura, que são representados por círculos, e as arestas são os relacionamentos existentes entre os vértices, sendo representadas por uma linha. Abaixo existem algumas figuras demonstrando a composição de um grafo.

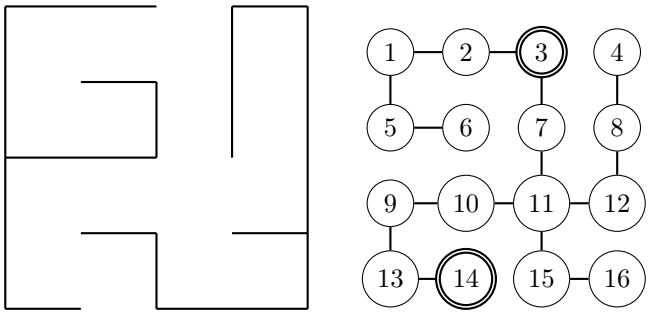
Figura 2: Exemplo de um grafo



Fonte: Elaborado pelo autor

O grafo acima pode estar demonstrando o caminho de uma cidade para outra, sendo cada vértice uma cidade e as arestas as rodovias interligando cada cidade. As arestas presentes nesse grafo são: (1, 2), (1, 3), (2, 3), (3, 4).

Figura 3: Representando um labirinto em um grafo



Fonte: Elaborado pelo autor

Podemos observar, através do grafo acima, que para sair do labirinto será necessário passar pelas arestas: (3, 7), (7, 11), (11, 10), (10, 9), (9, 13), (13, 14).

4.1 Conceitos Básicos de um Grafo

4.1.1 Grafos Direcionados

Em um grafo direcionado as relações representadas pelas arestas possuem sentido definido. Isso significa que as arestas só podem ser seguidas em uma única direção. Em grafos direcionados, as arestas são pares ordenados de vértices, saindo de um, a origem, e indo para o outro, o destino.

4.1.2 Grafos Não-Direcionados

4.1.3 Grafos Ponderados

4.2 Representação de um Grafo

Uma questão importante é como representar um grafo no computador, para isso, existem dois tipos principais de representações, Matriz de Adjacência e Lista de Adjacência.

4.2.1 Matriz de Adjacência

4.2.2 Lista de Adjacência

4.3 Busca em Largura

4.4 Busca em Profundidade

Exercícios

- **1076** - Desenhando Labirintos
- **1082** - Componentes Conexos

5 Miscelânea

5.1 Tabela ASCII

Figura 4: Tabela ASCII

Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex
(nul)	0	0000	0x00	(sp)	32	0040	0x20	@	64	0100	0x40	`	96	0140	0x60
(soh)	1	0001	0x01	!	33	0041	0x21	**A**	**65**	**0101**	**0x41**	**a**	**97**	**0141**	**0x61**
(stx)	2	0002	0x02	"	34	0042	0x22	**B**	**66**	**0102**	**0x42**	**b**	**98**	**0142**	**0x62**
(etx)	3	0003	0x03	#	35	0043	0x23	**C**	**67**	**0103**	**0x43**	**c**	**99**	**0143**	**0x63**
(eot)	4	0004	0x04	$	36	0044	0x24	**D**	**68**	**0104**	**0x44**	**d**	**100**	**0144**	**0x64**
(enq)	5	0005	0x05	%	37	0045	0x25	**E**	**69**	**0105**	**0x45**	**e**	**101**	**0145**	**0x65**
(ack)	6	0006	0x06	&	38	0046	0x26	**F**	**70**	**0106**	**0x46**	**f**	**102**	**0146**	**0x66**
(bel)	7	0007	0x07	'	39	0047	0x27	**G**	**71**	**0107**	**0x47**	**g**	**103**	**0147**	**0x67**
(bs)	8	0010	0x08	(	40	0050	0x28	**H**	**72**	**0110**	**0x48**	**h**	**104**	**0150**	**0x68**
(ht)	9	0011	0x09	)	41	0051	0x29	**I**	**73**	**0111**	**0x49**	**i**	**105**	**0151**	**0x69**
(nl)	10	0012	0x0a	*	42	0052	0x2a	**J**	**74**	**0112**	**0x4a**	**j**	**106**	**0152**	**0x6a**
(vt)	11	0013	0x0b	+	43	0053	0x2b	**K**	**75**	**0113**	**0x4b**	**k**	**107**	**0153**	**0x6b**
(np)	12	0014	0x0c	,	44	0054	0x2c	**L**	**76**	**0114**	**0x4c**	**l**	**108**	**0154**	**0x6c**
(cr)	13	0015	0x0d	-	45	0055	0x2d	**M**	**77**	**0115**	**0x4d**	**m**	**109**	**0155**	**0x6d**
(so)	14	0016	0x0e	.	46	0056	0x2e	**N**	**78**	**0116**	**0x4e**	**n**	**110**	**0156**	**0x6e**
(si)	15	0017	0x0f	/	47	0057	0x2f	**O**	**79**	**0117**	**0x4f**	**o**	**111**	**0157**	**0x6f**
(dle)	16	0020	0x10	**0**	**48**	**0060**	**0x30**	**P**	**80**	**0120**	**0x50**	**p**	**112**	**0160**	**0x70**
(dc1)	17	0021	0x11	**1**	**49**	**0061**	**0x31**	**Q**	**81**	**0121**	**0x51**	**q**	**113**	**0161**	**0x71**
(dc2)	18	0022	0x12	**2**	**50**	**0062**	**0x32**	**R**	**82**	**0122**	**0x52**	**r**	**114**	**0162**	**0x72**
(dc3)	19	0023	0x13	**3**	**51**	**0063**	**0x33**	**S**	**83**	**0123**	**0x53**	**s**	**115**	**0163**	**0x73**
(dc4)	20	0024	0x14	**4**	**52**	**0064**	**0x34**	**T**	**84**	**0124**	**0x54**	**t**	**116**	**0164**	**0x74**
(nak)	21	0025	0x15	**5**	**53**	**0065**	**0x35**	**U**	**85**	**0125**	**0x55**	**u**	**117**	**0165**	**0x75**
(syn)	22	0026	0x16	**6**	**54**	**0066**	**0x36**	**V**	**86**	**0126**	**0x56**	**v**	**118**	**0166**	**0x76**
(etb)	23	0027	0x17	**7**	**55**	**0067**	**0x37**	**W**	**87**	**0127**	**0x57**	**w**	**119**	**0167**	**0x77**
(can)	24	0030	0x18	**8**	**56**	**0070**	**0x38**	**X**	**88**	**0130**	**0x58**	**x**	**120**	**0170**	**0x78**
(em)	25	0031	0x19	**9**	**57**	**0071**	**0x39**	**Y**	**89**	**0131**	**0x59**	**y**	**121**	**0171**	**0x79**
(sub)	26	0032	0x1a	:	58	0072	0x3a	**Z**	**90**	**0132**	**0x5a**	**z**	**122**	**0172**	**0x7a**
(esc)	27	0033	0x1b	;	59	0073	0x3b	[	91	0133	0x5b	{	123	0173	0x7b
(fs)	28	0034	0x1c	<	60	0074	0x3c	\	92	0134	0x5c	|	124	0174	0x7c
(gs)	29	0035	0x1d	=	61	0075	0x3d	]	93	0135	0x5d	}	125	0175	0x7d
(rs)	30	0036	0x1e	>	62	0076	0x3e	^	94	0136	0x5e	~	126	0176	0x7e
(us)	31	0037	0x1f	?	63	0077	0x3f	_	95	0137	0x5f	(del)	127	0177	0x7f

Fonte: Macacário Maratona de Programação - Instituto Tecnológico de Aeronáutica

5.2 Merge Sort

Algoritmo 10: Algoritmo Merge Sort

```
#include <bits/stdc++.h>
#define INF 0x3f3f3f3f

using namespace std;

int MergeSort(int *A,int p, int r);
void Merge(int *A, int p, int q, int r);

int main(){

  int A[] = {3,5,2,1,4,6};
  int p = 0 , r = (int)(sizeof(A)/sizeof(A[0]));

```

```
14    for(int i = 0 ; i < r ; i++)
15      cout << A[i] << ' ';
16    cout << '\n';
17
18    MergeSort(A,p,r);
19
20    for(int i = 0 ; i < r ; i++)
21      cout << A[i] << ' ';
22    cout << '\n';
23
24    return 0;
25  }
26
27  int MergeSort(int *A, int p, int r){
28    if(p < r){
29      int q = (p+r)/2;
30      MergeSort(A,p,q);
31      MergeSort(A,q+1,r);
32      Merge(A,p,q,r);
33    }
34    return 0;
35  }
36
37  void Merge(int *A, int p, int q, int r){
38
39    int n1 = q-p+1, n2 = r-q;
40
41    int L[n1+1], R[n2+1];
42
43    for(int i = 1 ; i <= n1 ; i++)
44      L[i] = A[p+i-1];
45
46    for(int i = 1 ; i <= n2 ; i++)
47      R[i] = A[q+i];
48
49
50    L[n1+1] = abs(INF);
51    R[n2+1] = abs(INF);
52
53    int i = 1 , j = 1;
54
55    for(int k = p ; k <= r ; k++)
56      if(L[i] <= R[j])
57        A[k] = L[i++];
58        else
59          A[k] = R[j++];
60
61  }
```