Universidade Federal de Uberlândia
Faculdade de Computação

PGC101 – Análise de Algoritmos
Prof. Marcelo Keese Albertini

Proposta: Merge-insertsort

João Batista Ribeiro
joao.b@ufu.br

07/05/2021

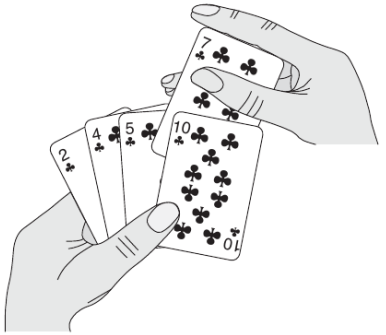
Roteiro

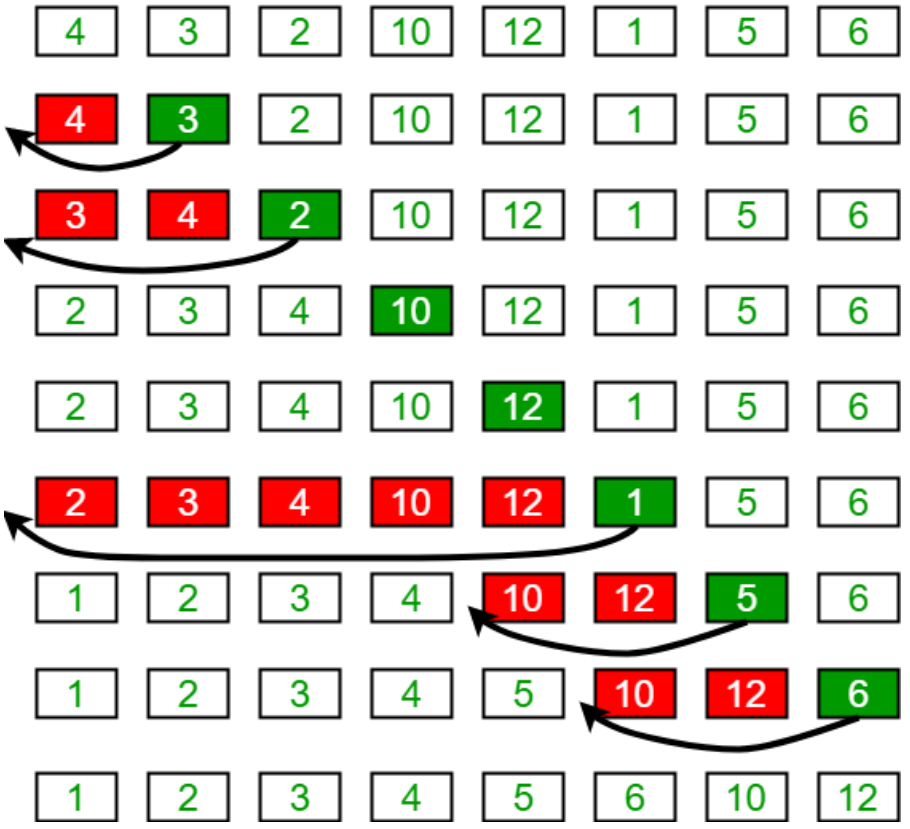
- Proposta
- Insertion sort
- Merge sort (Top Down e Bottom-up)
- Código desenvolvido
- Experimentos e Resultados
- Conclusão

Proposta para análise

- Conferir o arquivo:
  - trabalho1-merge-insertion.pdf

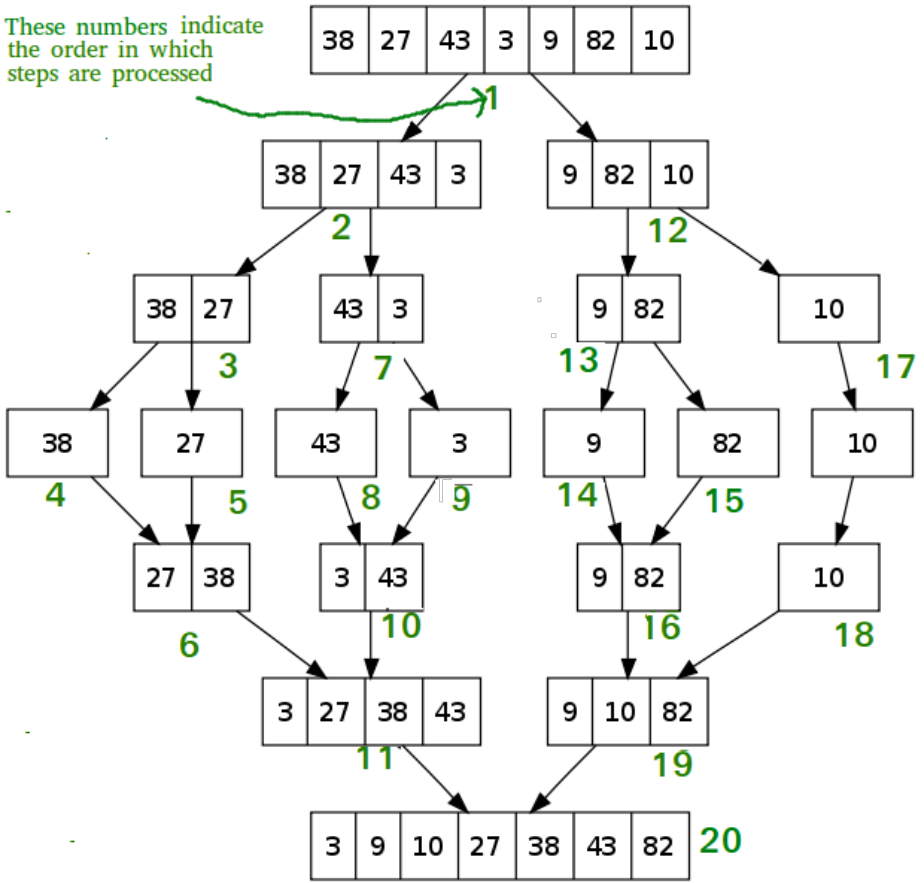
Insertion sort





Fonte: https://www.geeksforgeeks.org/merge-sort/

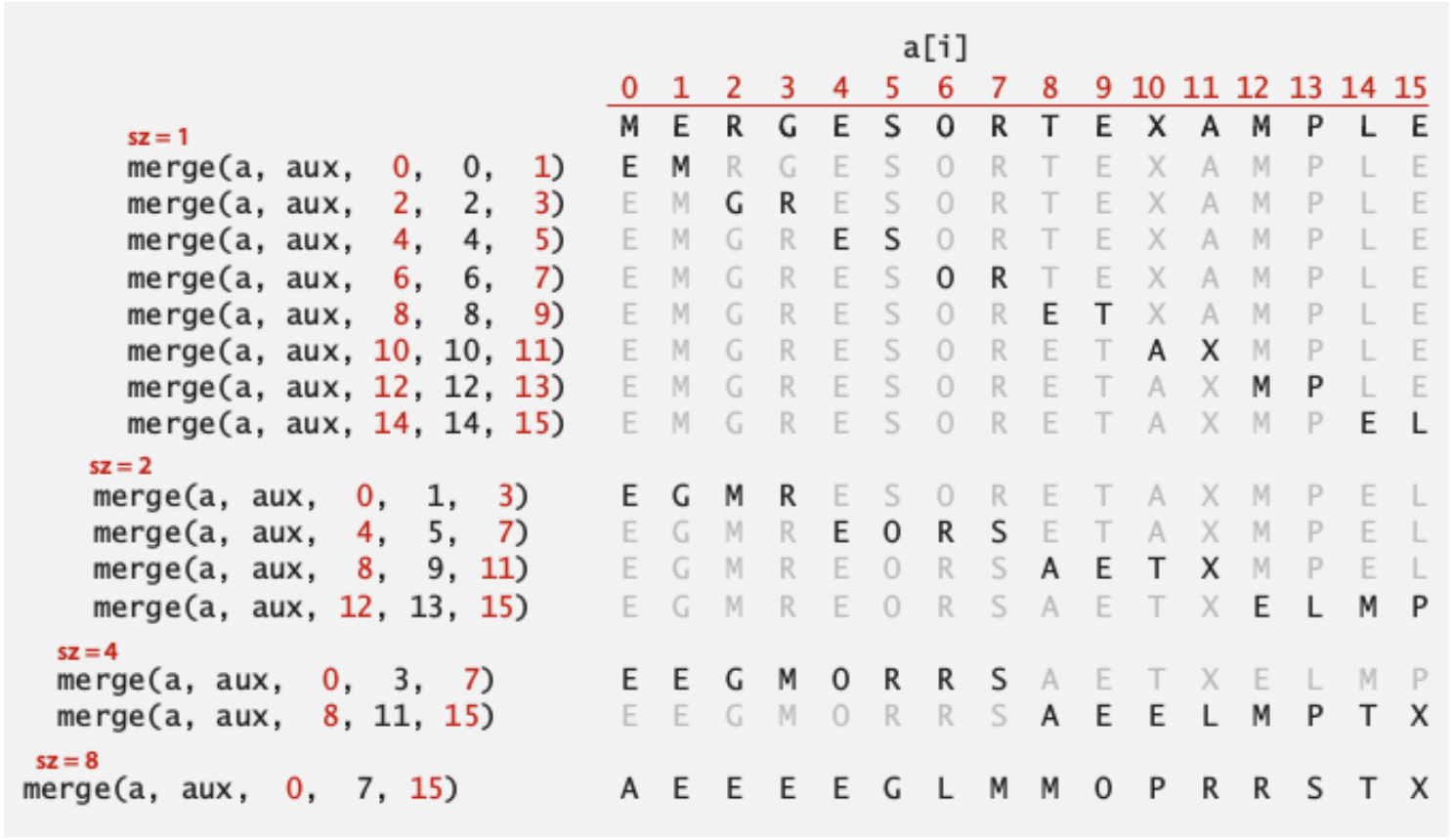
Merge Sort Top Down



Dividir para conquistar

Fonte: https://www.geeksforgeeks.org/merge-sort/

Merge Sort Bottom-up



Dividir para Conquistar

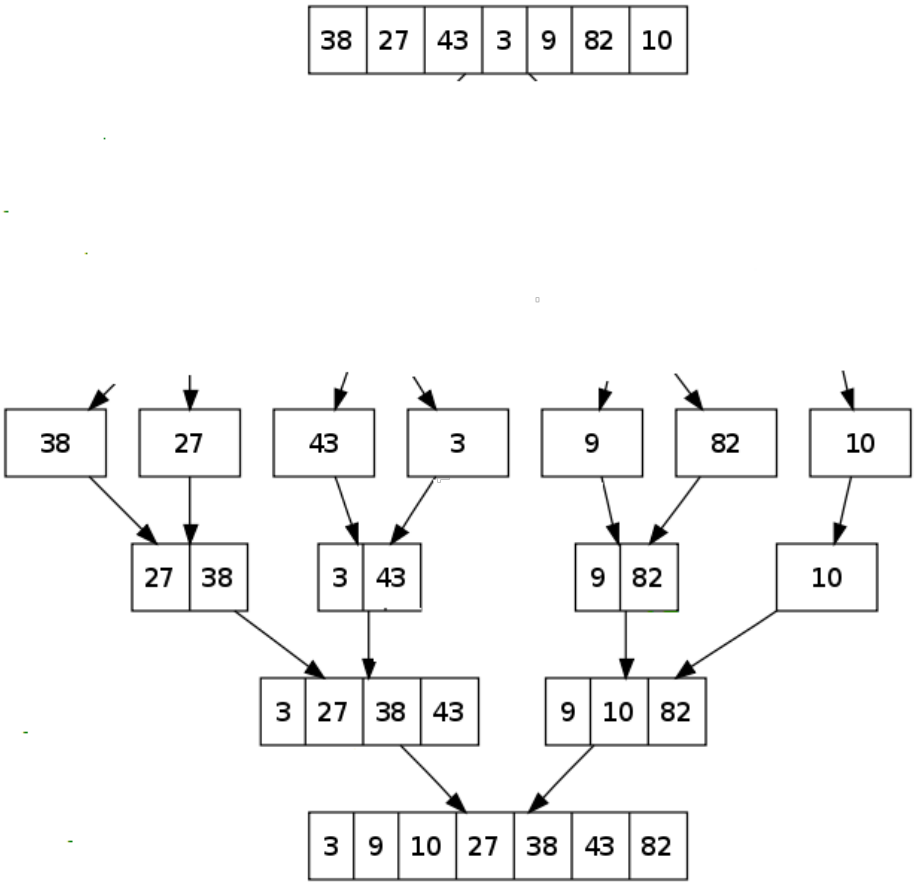
* **Sem Recursão**

*** Sem o "sort(*)"**

*** Laços chamando merge(*)**

Fonte: https://algs4.cs.princeton.edu/22mergesort/

Merge Sort Top Down



Dividir para Conquistar

* **Sem Recursão**

*** Sem o “sort(*)”**

*** Laços chamando merge(*)**

Análise Inicial

	Complexidade de Tempo			Complexidade de espaço
Algoritmo	Melhor caso	Caso médio	Pior caso	Pior caso
Insertion sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Merge sort	O(n log n)	O(n log n)	O(n log n)	O(n)

Hipótese:

- Se que se utilizar **Insertion sort** no **Merge sort** quando o for ordenar um vetor com número de elementos pequeno (menor que 1.000?), o tempo de execução será menor que no **Merge sort** puro?

Objetivo:

1) Confirmar se é verdade
2) Caso sim, qual seria um bom valor para **n** (número de elementos)?

Código desenvolvido

Ver arquivos:

- **code.cpp** // Código em C++ dos algoritmos, testes e geração resultados
- **commands.r** // Para compilar e executar o código C++
- **getDataResults.sh** // Shell script para “limpar” os resultados em “results.txt” e separar os resultados em arquivos específicos

Insersion Sort

```c
void insertionSort(int *vet, int size) {
    for(int j = 1; j < size; j++) {
        int chave = vet[j];
        int i = j - 1;

        // procura lugar de insercao e desloca numero
        while(i >= 0 && vet[i] > chave) {
            vet[i + 1] = vet[i];
            i--;
        }

        vet[i + 1] = chave;
    }
}
```

```c
void insertionSortM(int *vet, int low, int high) {
//     int size;

    for(int j = low + 1; j < high; j++) {
        int chave = vet[j];
        int i = j - 1;

        // procura lugar de insercao e desloca numero
        while(i >= low && vet[i] > chave) {
            vet[i + 1] = vet[i];
            i--;
        }

        vet[i + 1] = chave;
    }
}
```

Mergesort Top Down

```
1  void sort(int *vet, int *aux, int inf, int sup) {
2      if (sup <= inf) {
3          return;
4      }
5
6      int med = inf + (sup - inf) / 2;
7
8      sort(vet, aux, inf, med);
9      sort(vet, aux, med + 1, sup);
10     merge(vet, aux, inf, med, sup);
11 }
12
13 void mergesortTopDown(int *vet, int size) {
14     int *aux = new int[size];
15
16     sort(vet, aux, 0, size - 1);
17
18     delete []aux;
19 }
20
21
22
23
24
25
```

```
1  void sortMI(int *vet, int *aux, int inf, int sup, int threshold) {
2      if (sup <= inf) {
3          return;
4      }
5
6      if ((sup - inf) >= threshold) {
7          int med = inf + (sup - inf) / 2;
8
9          sortMI(vet, aux, inf, med, threshold);
10         sortMI(vet, aux, med + 1, sup, threshold);
11
12         merge(vet, aux, inf, med, sup);
13     } else {
14         insertionSortM(vet, inf, sup + 1);
15     }
16 }
17
18 void mergesortTopDownInsertionSort(int *vet, int size, int threshold)
19     int *aux = new int[size];
20
21     sortMI(vet, aux, 0, size - 1, threshold);
22
23     delete []aux;
24 }
25
```

Ambiente de experimentos

Máquina: Thinkpad X200

Processador: Intel Core 2 Duo (sem Turbo Boost e sem Hyper-Threading), com 2 núcleos e 2 threads

Gcc: g++ (GCC) 10.2.0

Sistema Operacional: Slackware 14.2 + (Current), 64 bits

Quantidade de RAM: 6 GiB

Experimento 1

Ordenar 10 vetores com:

- 6.5 * 10 ^ 6 (6.500.000) valores cada
- Valores inteiros (int)
- Vetor com valores
  - 1 aleatórios
  - 2 decrescentes
  - 3 crescentes (ordenado)

10 execuções cada

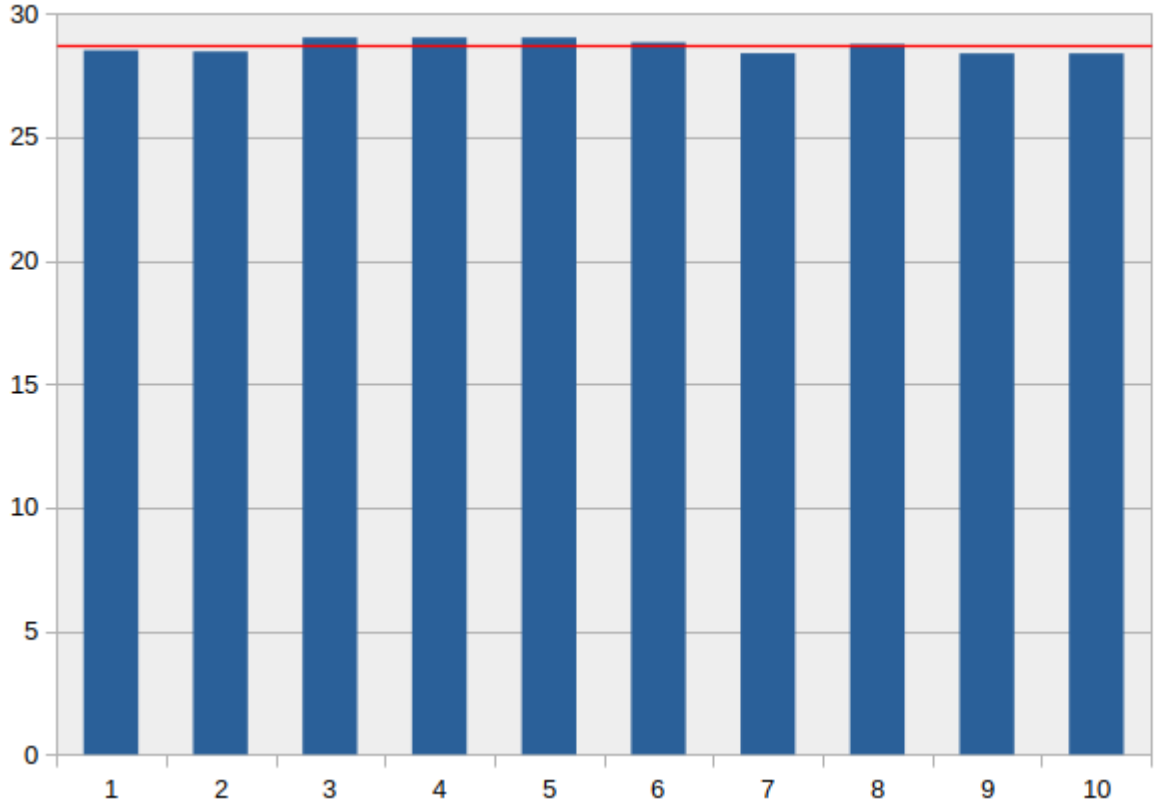
Experimento 1

Consumo do processador: 50 % do total (100 % de um núcleo)

Consumo de RAM (no pior/maior caso): ~520 MiB, (10 * 2 + 1) * 6.5 * 10^6 * 32) / (8 * 1024 ^ 2) =~ 520 MiB

Tempo para executar: ~ 2 h

- real 115m16.586s; user 115m9.096s; sys 0m1.857s
- **Pode variar bastante dependendo da máquina que for executado, principalmente pelo processador**



Execução do Merge Sort Top Down

Análise do tempo de execução

Run	1	2	3	4	5	6	7	8	9	10
ms	28513	28469	29031	29032	29031	28838	28394	28771	28393	28391
s	28,513	28,469	29,031	29,032	29,031	28,838	28,394	28,771	28,393	28,391

Resultados - Números **aleatórios**

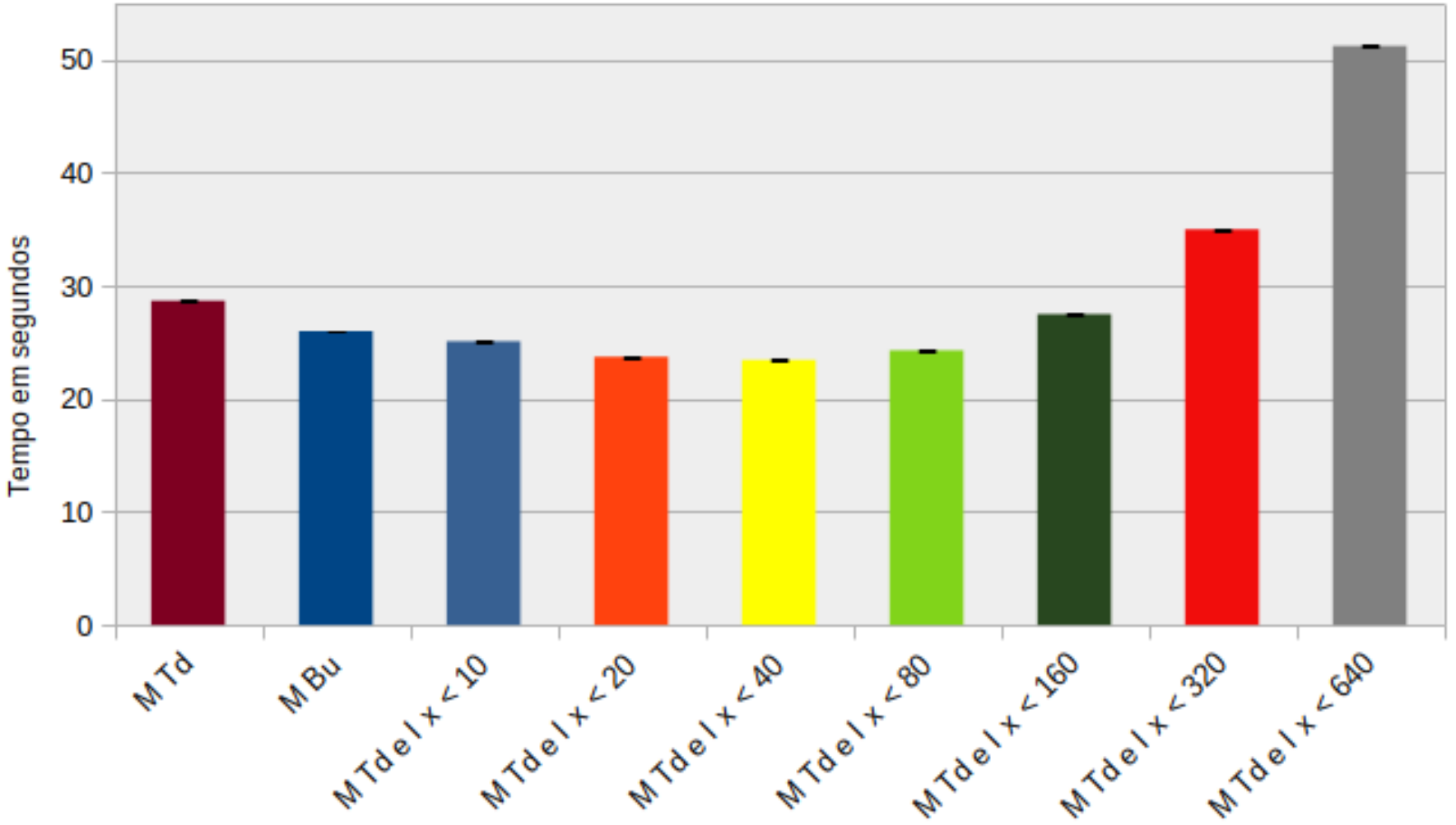
	Tempo em Segundos		
Algoritmo	**Média**	**Desvio Padrão**	**Erro Padrão**
M Td	28,686	0,283	0,090
M Bu	25,999	0,261	0,083
M Td e I x < 10	25,107	0,238	0,075
M Td e I x < 20	23,722	0,227	0,072
M Td e I x < 40	23,442	0,191	0,060
M Td e I x < 80	24,291	0,243	0,077
M Td e I x < 160	27,494	0,258	0,082
M Td e I x < 320	35,004	0,291	0,092
M Td e I x < 640	51,227	0,365	0,115

M	Merge sort
I	Insertion Sort
x	Limiar para M Td + I

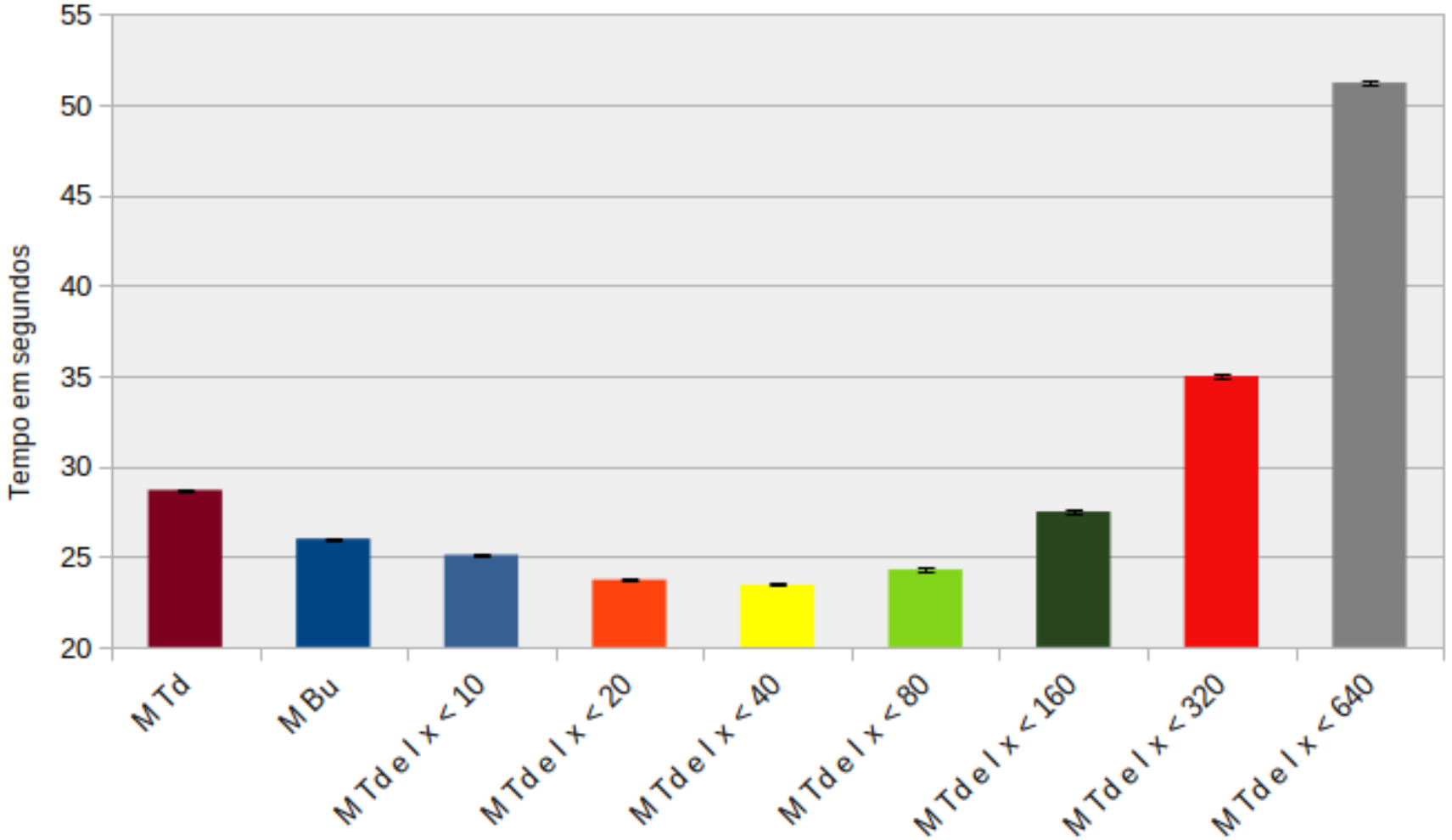
Bu	Bottom-up
Td	Top down

Erro Padrão: DP/n^1/2

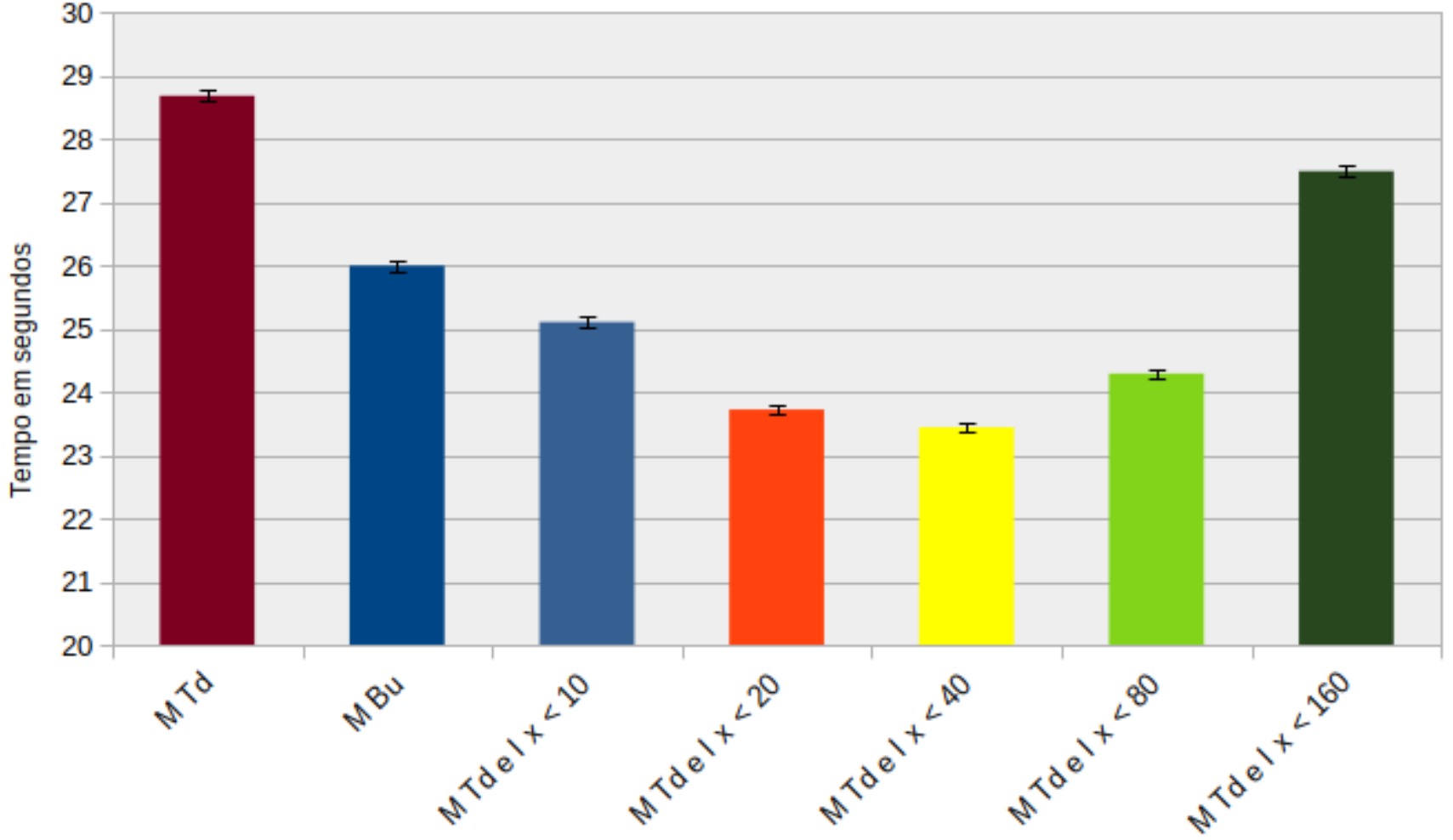
Resultados - Números **aleatórios**



Resultados - Números **aleatórios**



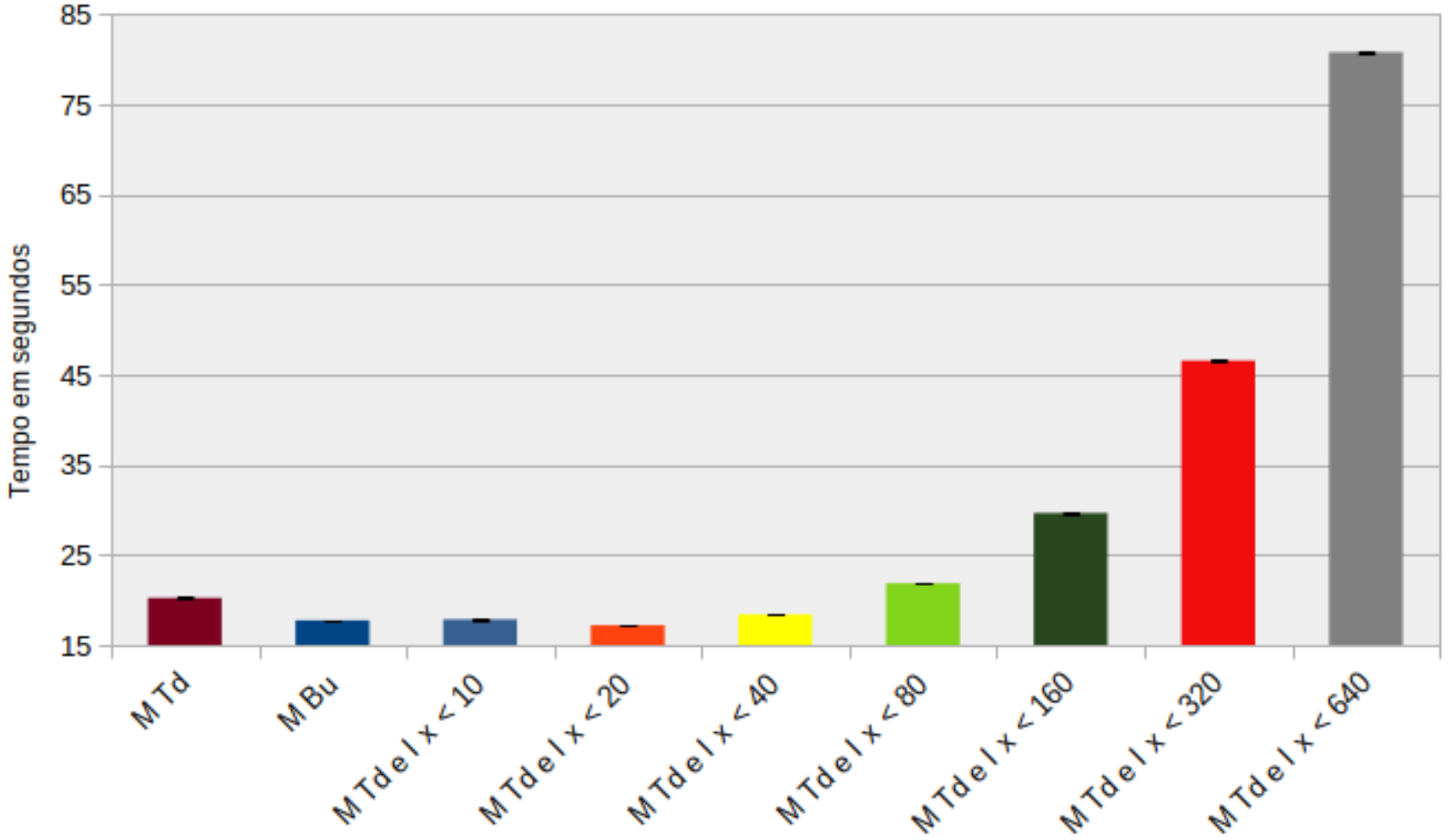
Resultados - Números **aleatórios**



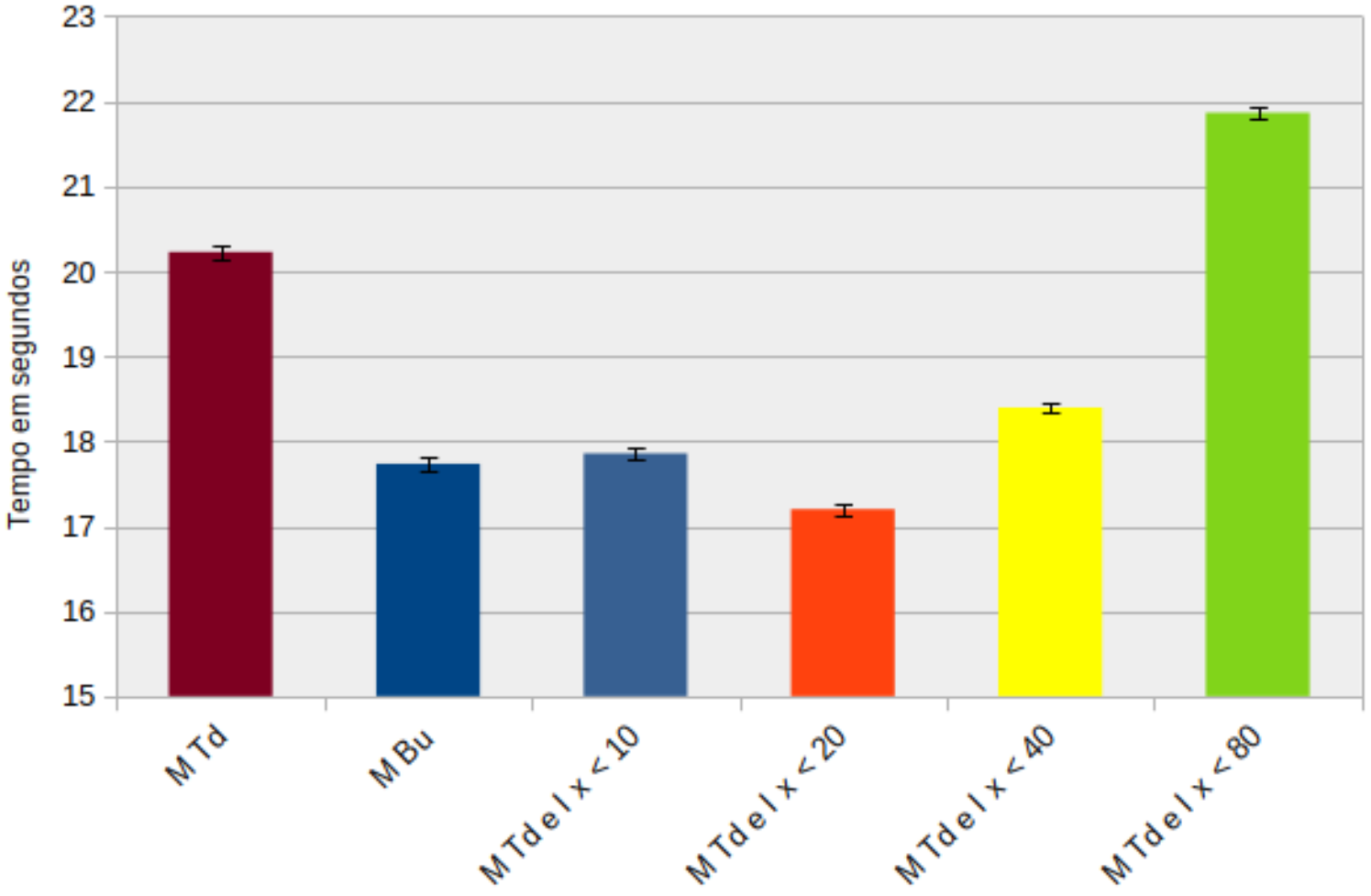
Resultados - Números **decrescentes**

	Tempo em Segundos		
Algoritmo	**Média**	**Desvio Padrão**	**Erro Padrão**
M Td	20,227	0,177	0,056
M Bu	17,734	0,180	0,057
M Td e I x < 10	17,856	0,181	0,057
M Td e I x < 20	17,201	0,137	0,043
M Td e I x < 40	18,396	0,164	0,052
M Td e I x < 80	21,862	0,182	0,057
M Td e I x < 160	29,657	0,278	0,088
M Td e I x < 320	46,577	0,334	0,106
M Td e I x < 640	80,786	0,611	0,193

Resultados - Números **decrescentes**



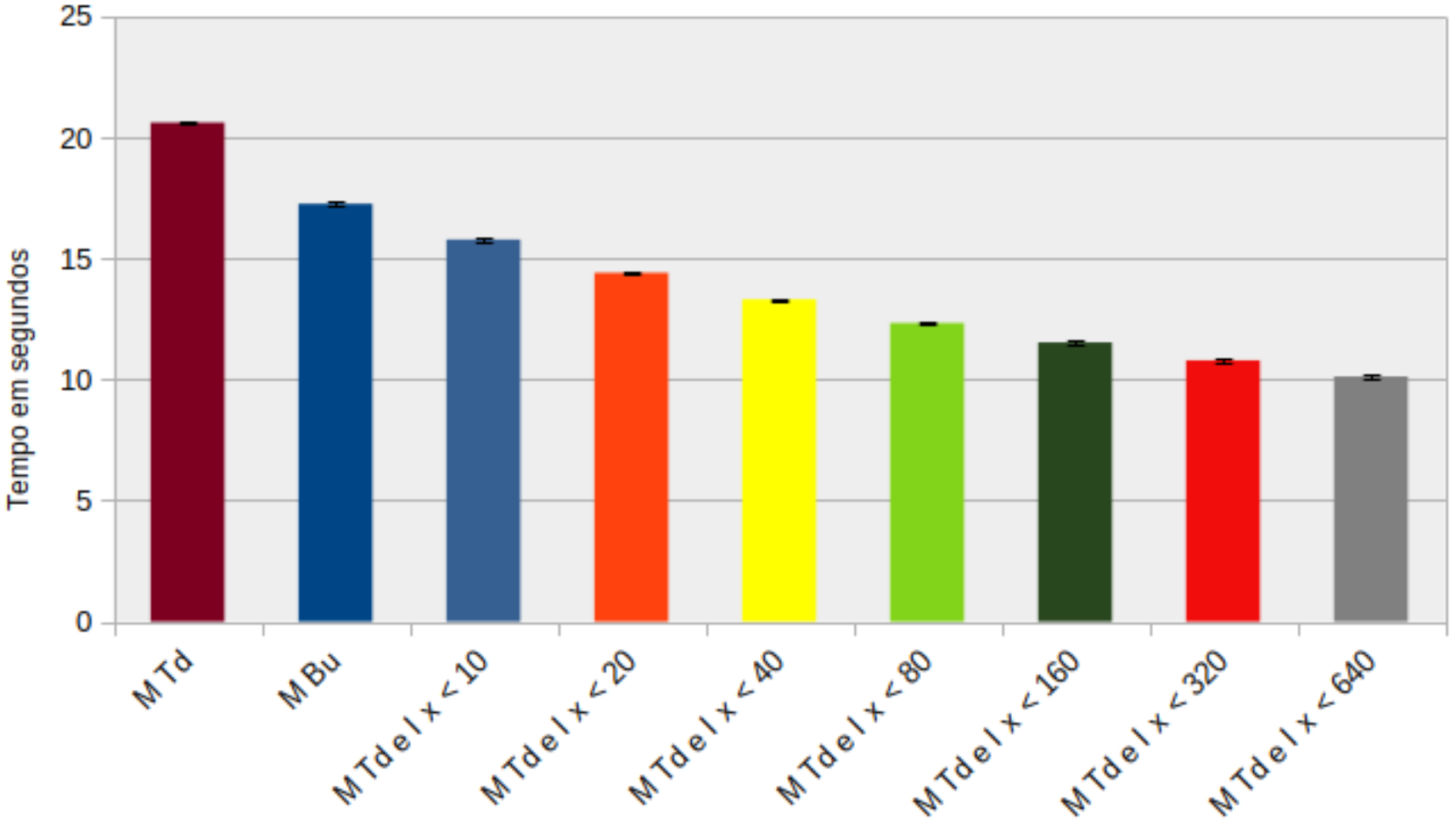
Resultados - Números **decrescentes**



Resultados - Números **crescentes**

	Tempo em Segundos		
Algoritmo	**Média**	**Desvio Padrão**	**Erro Padrão**
M Td	20,613	0,117	0,037
M Bu	17,256	0,067	0,021
M Td e I x < 10	15,787	0,036	0,011
M Td e I x < 20	14,413	0,079	0,025
M Td e I x < 40	13,312	0,026	0,008
M Td e I x < 80	12,332	0,078	0,025
M Td e I x < 160	11,535	0,058	0,018
M Td e I x < 320	10,793	0,013	0,004
M Td e I x < 640	10,127	0,056	0,018

Resultados - Números **crescentes**



Experimento 2

Ordenar 10 vetores com:

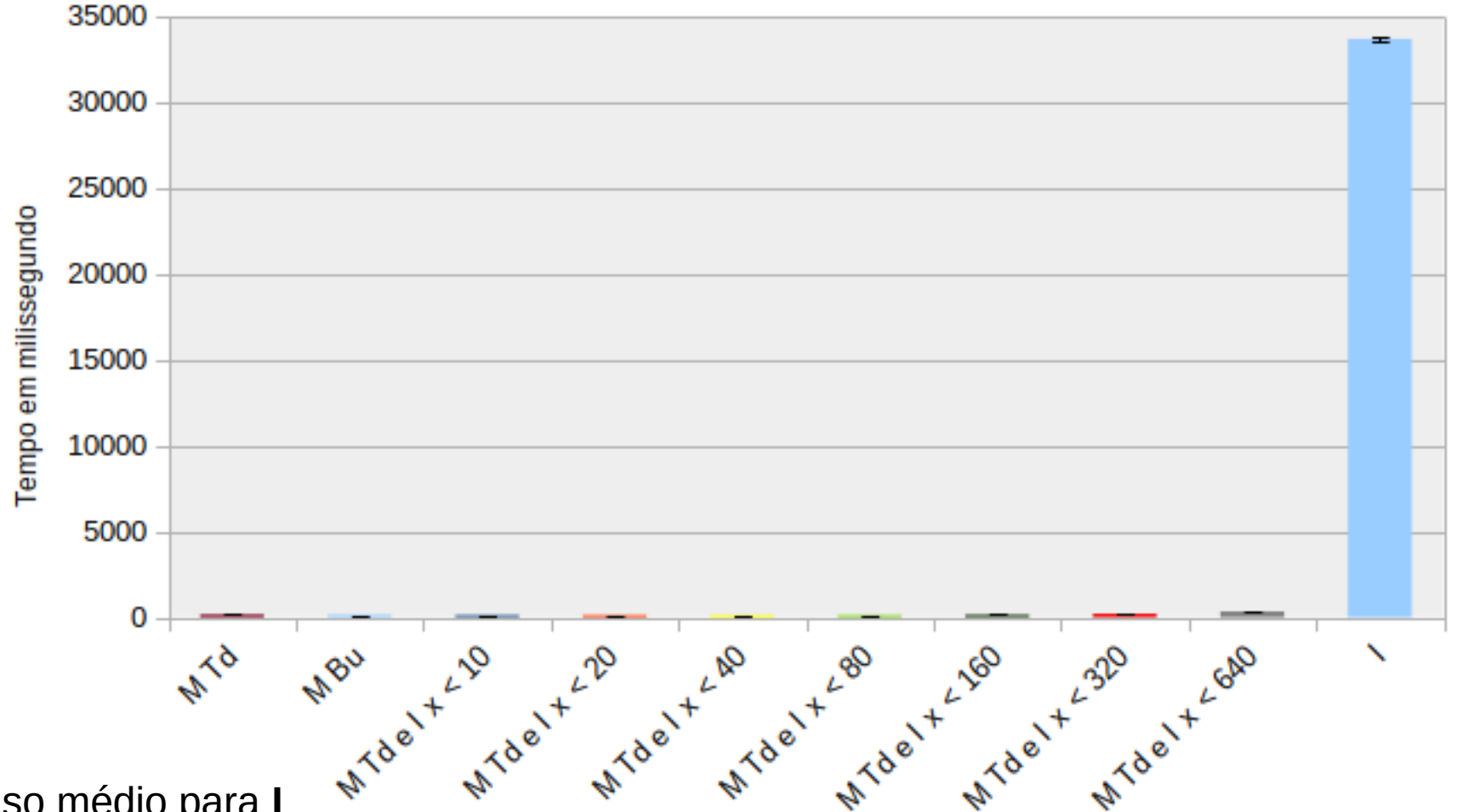
- 5 * 10 ^ 4 (50.000) valores cada
- Valores inteiros (int)
- Vetor com valores
  - 1 aleatórios
  - 2 decrescentes
  - 3 crescentes (ordenado)

10 execuções cada

Resultados - Números **aleatórios**

	Tempo em Milissegundos		
Algoritmo	**Média**	**Desvio Padrão**	**Erro Padrão**
M Td	161,333	1,323	0,418
M Bu	137,300	1,160	0,367
M Td e I x < 10	133,300	1,252	0,396
M Td e I x < 20	122,400	0,966	0,306
M Td e I x < 40	119,600	1,174	0,371
M Td e I x < 80	126,400	0,966	0,306
M Td e I x < 160	150,200	1,317	0,416
M Td e I x < 320	208,000	1,886	0,596
M Td e I x < 640	331,800	3,327	1,052
I	33688,400	252,280	79,778

Resultados - Números **aleatórios**

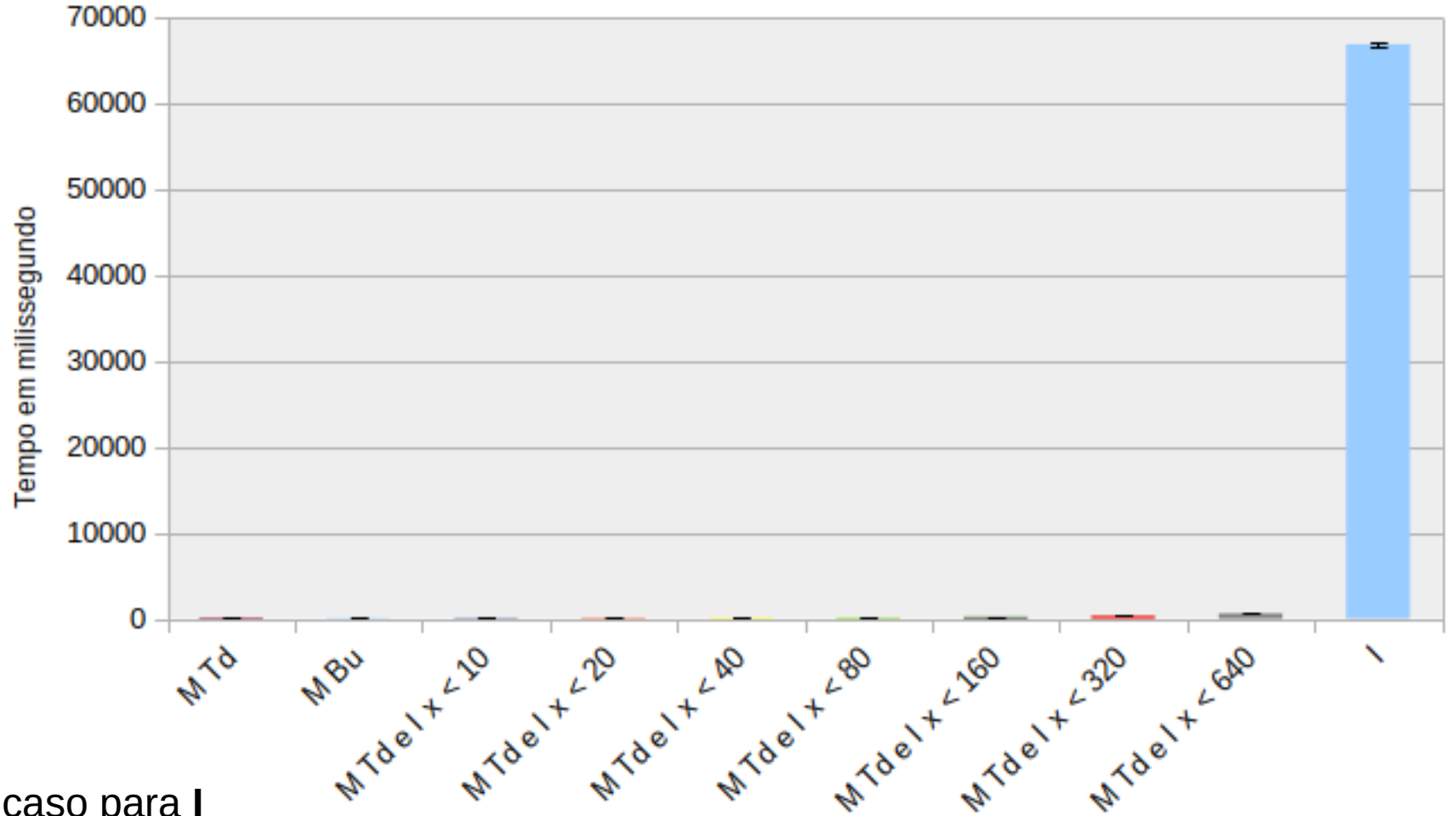


~ caso médio para **I**

Resultados - Números **decrescentes**

	Tempo em Milissegundos		
Algoritmo	**Média**	**Desvio Padrão**	**Erro Padrão**
M Td	114,667	1,000	0,316
M Bu	94,600	0,843	0,267
M Td e I x < 10	97,900	1,197	0,379
M Td e I x < 20	92,800	1,033	0,327
M Td e I x < 40	101,100	1,197	0,379
M Td e I x < 80	127,100	1,449	0,458
M Td e I x < 160	186,500	1,958	0,619
M Td e I x < 320	312,400	3,098	0,980
M Td e I x < 640	569,700	6,129	1,938
I	66834,600	538,729	170,361

Resultados - Números **decrescentes**

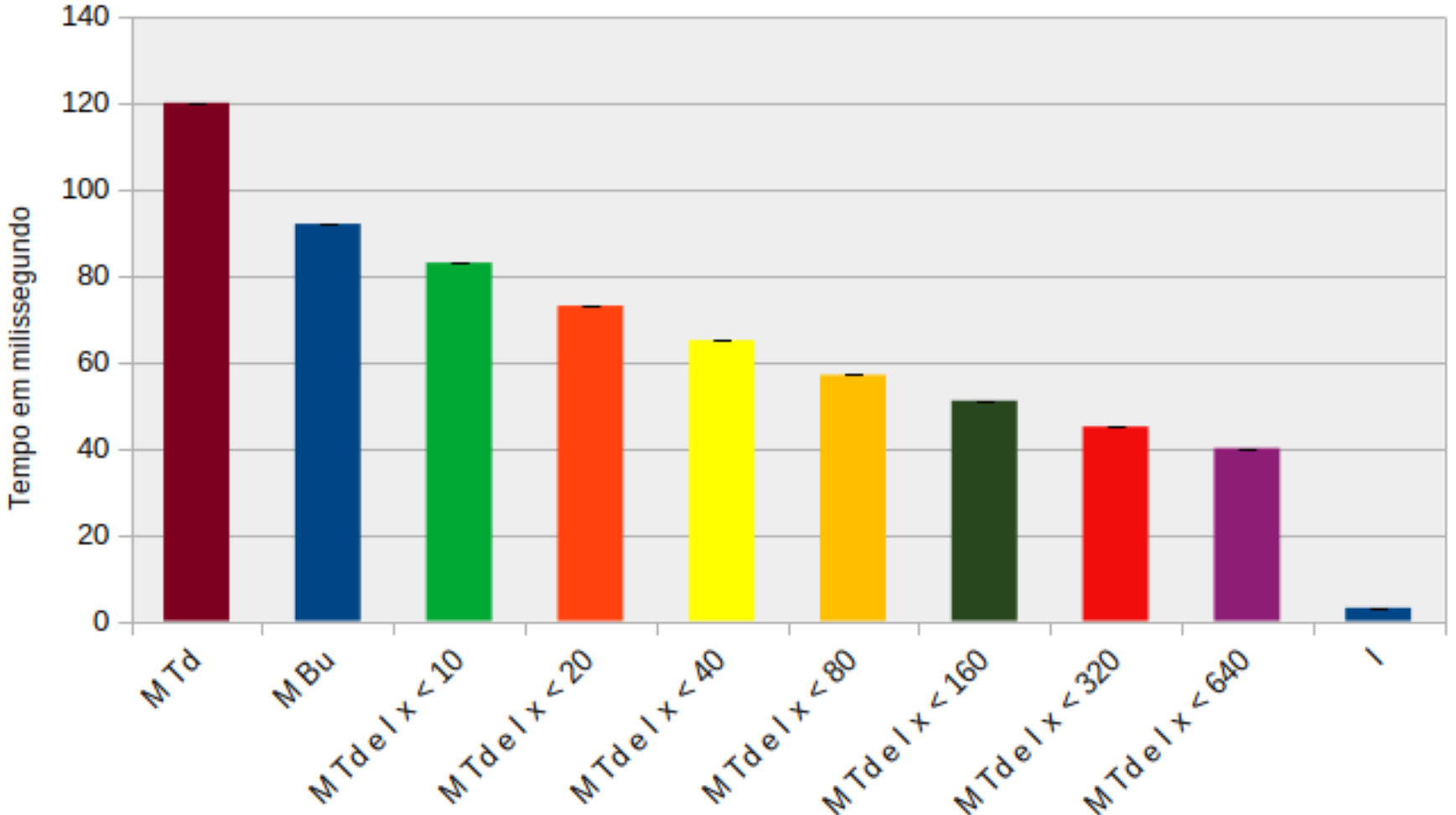


pior caso para **I**

Resultados - Números **crescentes**

	Tempo em Milissegundos		
Algoritmo	**Média**	**Desvio Padrão**	**Erro Padrão**
M Td	120,000	0,000	0,000
M Bu	92,000	0,000	0,000
M Td e I x < 10	83,000	0,000	0,000
M Td e I x < 20	73,000	0,000	0,000
M Td e I x < 40	65,000	0,000	0,000
M Td e I x < 80	57,000	0,000	0,000
M Td e I x < 160	51,000	0,000	0,000
M Td e I x < 320	45,000	0,000	0,000
M Td e I x < 640	40,000	0,000	0,000
I	3,000	0,000	0,000

Resultados - Números **crescentes**



melhor caso para **I**

Conclusão

Merge sort (top down) + **Insertion sort** tem resultado um pouco melhor que **Merge sort** puro

Insertion sort tem bom resultado com vetores com número de elementos (**n**) pequenos

- Um bom **n** fica entre 20 e 80 (melhor resultado para números aleatórios)
- Rápido para entradas com valores quase ordenados
- Está de acordo com resultado de desenvolvedores e pesquisadores =>

Conclusão

Timsort utilizam uma combinação de **Merge sort** + **Insertion sort** e **busca binária** (para encontrar onde inserir o valor no Insertion sort). Usado no Python, Java e outras linguagens

TimSort – descrição e detalhes (Utiliza **n** entre 32 e 64):

https://hg.python.org/cpython/file/tip/Objects/listsort.txt

https://github.com/python/cpython/blob/master/Objects/listobject.c

Java JDK 8, n = MIN_MERGE = 32

http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/687fd7c7986d/src/share/classes/java/util/TimSort.java

Trabalhos futuros

- Testar para outros valores de **n** mais próximos de 32 e 64
- Testar Mege sort bottom-up com Insertion Sort
- Testar para outros tipos de valores (double, string etc)
- Comparar e validar a estabilidade do tempo de execução para variados tamanhos de vetores
- Analisar para cada algoritmo o custo em questão de operações (número de comparações, atribuições)

Principais Referências

Material das aulas de Análise de Algoritmos do Prof. Marcelo Keese Albertini,Faculdade de Computação Universidade Federal de Uberlândia. http://www.facom.ufu.br/~albertini/ada/index.html

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. **Introduction to Algorithms**, Third Edition (3rd. ed.). The MIT Press.

Geeksforgeeks, **Merge Sort**, https://www.geeksforgeeks.org/merge-sort/. Aceso em 28/04/2021

Geeksforgeeks, **Insertion Sort**, https://www.geeksforgeeks.org/insertion-sort/. Aceso em 29/04/2021

Algorithms, 4th Edition. Booksite: https://algs4.cs.princeton.edu/. Aceso em 28/04/2021

Obrigado pela atenção! :-)

joao.b@ufu.br

Código desenvolvido:

https://github.com/ryuuzaki42/merge-insertsort