Introdução

Sumário

Feedback Trabalho 1

Trabalho Prático 2

Tutorial Análise Experimental

Trabalho Prático 1

Trabalho Prático 1

Critérios Correção

Implementação ED	Utilização ED	Boas Práticas	Documentação	Total
3	4	1	2	10

Trabalho Prático 1

Critérios Correção

Inserção/Remoção na Lista
Corner Cases: início e final da lista

Implementação ED	Utilização ED	Boas Práticas	Documentação	Total
3	4	1	2	10

Trabalho Prático 1

Critérios Correção

Usar a estrutura para fazer a medida correta (testes de 1 a 11)

Implementação ED	Utilização ED	Boas Práticas	Documentação	Total
3	4	1	2	10

Trabalho Prático 1

Critérios Correção

Leaks de memória
Modularidade
Nomes de variáveis

Implementação ED	Utilização ED	Boas Práticas	Documentação	Total
3	4	1	2	10

Trabalho Prático 1

Critérios Correção

Explicar decisões (porquê, não como)
Análise assintótica teórica - explicar
Análise assintótica experimental

Implementação ED	Utilização ED	Boas Práticas	Documentação	Total
3	4	1	2	10

Trabalho Prático 1

Nomes de Variáveis

aux, **aux1**, **aux2**, e **auxiliar** são nomes tão genéricos quanto **x**, **y**, **z**

Nos trabalhos entregues, **aux (ou temp)** e suas variantes representam*:

- a lista de **recipientes**
- a lista/fila de **medidas**
- a **soma** de recipiente e medida
- a **sub**tração de medida por recipiente
- iterador para a lista de medidas (**medida**, **medida_i**, **medida_iterator**, **m_i**)
- iterador para a lista de recipientes (**recipiente**, **recipiente_i**, **recipiente_iterator**, **r_i**)

Algumas vezes, em funções mais simples, (inserção, remoção, construtor da lista) não é tão ruim usar esses nomes genéricos nas variáveis, mas evite sempre que possível

* as palavras em **azul** seriam alternativas para nomear essas variáveis

Trabalho Prático 1

Código vs Pseudo-Código na documentação

Quase sempre, não é necessário expor sua implementação da documentação

Quando essencial, prefira utilizar pseudo-códigos para **sumarizar** a idéia da sua implementação

Cuide também da formatação do seu pseudo-código, pois a maioria dos editores de texto não irão criar um layout agradável automaticamente

Trabalho Prático 1

Código vs Pseudo-Código na documentação

```
int min_operations(int q, List *containers) {
  if (q == 0) return 0;
  List *measures = new List();
  measures->push(new ListNode(0, 0));
  while (true) {
    ListNode *measure = measures->pop(),
             *containers_i = containers->first;
    while (containers_i != NULL) { // percorre toda a lista de recipientes
      int sum = measure->value + containers_i->value, // combina medida com
          sub = measure->value - containers_i->value, // o recipiente[i]
          ops = measure->operations + 1; // a combinacao gasta 1 operacao
      if (sum == q || sub == q) {
        delete measure;
        delete measures;
        return ops;
      }
      // adiciona as novas medidas (sum e sub) na lista de medidas realizadas
      measures->push(new ListNode(sum, ops));
      if (sub > 0) measures->push(new ListNode(sub, ops));
      containers_i = containers_i->next;
    }
    delete measure;
  }
  return -1; // nunca chega aqui
}
```

O código fonte contém muita informação que não é essencial para o entendimento do seu algoritmo

Trabalho Prático 1

Código vs Pseudo-Código na documentação

Algorithm 1: Pseudo-código do MinOperations

input : ml: desired ammount, containers: list of available flasks
output: op: operations necessary to measure the desired ammount

```
1  measures = Queue;
2  measures.insert(0, 0);
3  for measure_i ∈ measures do
4      for container_i ∈ containers do
5          sum = measure_i.ml + container_i.ml;
6          sub = measure_i.ml − container_i.ml;
7          op = measure_i.op + 1;
8          if sum == ml || sub == ml then
9              return op;
10         measures.insert_unique(sum, op);
11         measures.insert_unique(sub, op);
```

O pseudo-código não precisa ser 100% fiel ao código (checar se a medida é maior que 0, desalocar memória, etc), e por isso é mais simples de entender

Trabalho Prático 1

Análise Assintótica

É preciso usar notação correta durante sua análise:

Notação correta: $f(n) = O(n^2)$, ou $f(n)$ é $O(n^2)$

Notação errada: $f(n)$ é $O(n) = O(n^2)$

Além disso, é importante explicar seu raciocínio para obter o custo assintótico.

O número de loop aninhados não é, necessariamente, uma justificativa correta

Trabalho Prático 1

Análise Assintótica

```
1 int f(n) {
2   int k = 0;
3   for (int i = 0; i < n; ++i)
4     for (int j = 0; j < n; ++j)
5       ++k;
6   return k;
7 }
```

Dois loops aninhados

f(n) é $O(n^2)$

Trabalho Prático 1

Análise Assintótica

```
1 int f(n, m) {
2   int k = 0;
3   for (int i = 0; i < n; ++i)
4     for (int j = 0; j < m; ++j)
5       ++k;
6   return k;
7 }
```

Dois loops aninhados

f(n) é O(n.m)

f(n) **não** é $O(n^2)$,
a não ser que você tenha certeza que n >= m

Trabalho Prático 1

Análise Assintótica

```
1 int f(n) {
2   int k = 0,
3       m = n**2;
4   for (int i = 0; i < n; ++i)
5     for (int j = 0; j < m; ++j)
6       ++k;
7   return k;
8 }
```

Dois loops aninhados

f(n) é O(n.m) ou f(n) é $O(n^3)$

f(n) **não** é $O(n^2)$

aqui, n**2 representa (n ao quadrado), ou n^2, ou pow(n ,2)

Trabalho Prático 1

Análise Assintótica

A maioria das implementações continham 2 ou 3 loops aninhados, porém, a complexidade real do algoritmo não é nem $O(n^2)$ nem $O(n^3)$

Podemos descobrir a complexidade analisando como obtemos as medidas. Veja que existe somente **1** (ou $(2n)^0$) medida possível com 0 operações

A partir dessa medida, fazemos **2n** somas/subtrações para obter as medidas que utilizam 1 operação (**n** somas e **n** subtrações, onde **n** é |recipientes|)

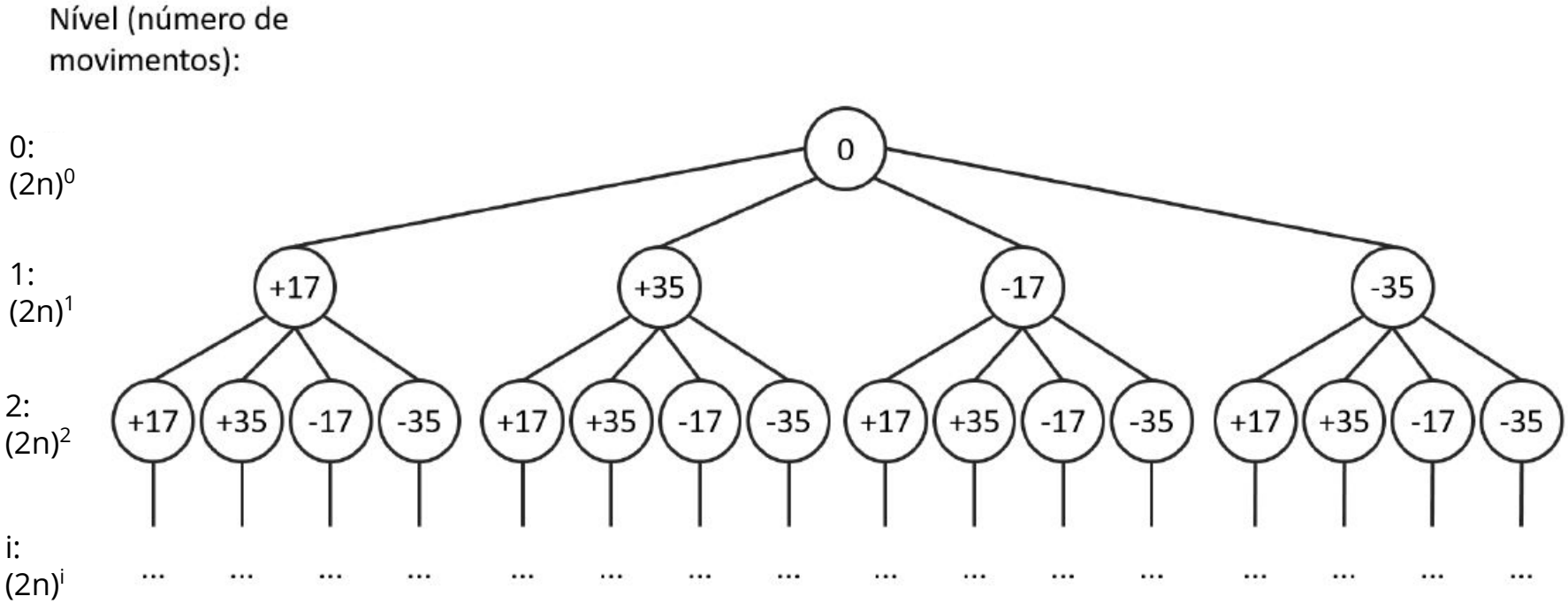
Para cada uma dessas **2n** medidas, fazemos mais **2n** somas/subtrações para obter as medidas de 2 operações, totalizando $\mathbf{(2n)^2}$ somas/subtrações nesse nível

De modo geral, no nível **k**, faremos **2n** medidas a partir de $\mathbf{(2n)^{k-1}}$ medidas

Assim, a complexidade de uma medida que necessita de **k** operações pode ser obtida a partir do somatório dos níveis anteriores

Trabalho Prático 1

Análise Assintótica



$$\sum_{i=0}^{k}(2n)^i = \frac{(2n)^k - 1}{2n - 1} = O((2n)^k)$$

A fórmula fechada do somatório é obtida com a fórmula de `soma finita dos termos de uma progressão geométrica de razão 2n`

Trabalho Prático 1

Análise Assintótica

Outro erro recorrente foi dizer que a função é $O(n^k)$, mas vejam que não existe constante c tal que $c.n^k > 2^k n^k$, para $k > k_0$ e $n > n_0$, para algum k_0 e n_0

(não é possível encontrar c e k_0 tais que $c > 2^k$, para todo $k > k_0$)

$$f(n,k) = (2n)^k \quad = O((2n)^k) = O(2^k n^k)$$
$$\neq O(n^k)$$
$$\neq O(n^n)$$

Trabalho Prático 2

Trabalho Prático 2

Contexto

Você deve gerenciar uma agenda de viagens interplanetária

Somente um tempo **T** está alocado para viajar durante cada mês

Cada planeta exige um tempo $\mathbf{t_i}$ de visita, devido ao tempo de viagem

Você deve ordenar um conjunto **P** de planetas, de tal forma que:
- o número de planetas visitados num mês $\mathbf{m_i}$ seja sempre maior que o número de planetas visitados no mês $\mathbf{m_{i+1}}$
- o tempo de visita de um planeta $\mathbf{p_j}$ alocado em $\mathbf{m_i}$ deve ser menor ou igual ao tempo de visitado de um planeta $\mathbf{p_k}$ alocado em $\mathbf{m_{i+1}}$
- os planetas alocados para o mesmo mês devem ser ordenados alfabeticamente entre si

Para maiores informações, leia a especificação

Trabalho Prático 2

Detalhes

O programa consiste basicamente de duas ordenações

A ordenação dos planetas deve ser feita em $O(P \log_2 P)$ e deve ser estável

A ordenação alfabética deve ser feita em O(P.K), onde K é o tamanho do nome dos planetas

A especificação diz que $K < \log_2 P$

Portanto, os algoritmos comparativos que possuem complexidade $O(P \log_2 P)$ não são apropriados

Trabalho Prático 2

Exemplo Passo a Passo

Analisaremos uma entrada aleatória com

- Tempo mensal disponível: 60
- Número de planetas: 15
- Tamanho do nome dos planetas: 4

```
60 15 4
50 rewz
33 pvih
39 xvmt
23 jfee
41 rdlu
44 grpa
40 ezcw
27 jngn
16 qzmf
43 vyth
58 wyvc
17 asle
18 ymzk
38 sqnm
 6 rkse
```

Trabalho Prático 2

Exemplo Passo a Passo

O primeiro passo é ordenar essa lista de acordo com o tempo $\mathbf{t_i}$ necessário para visitar cada planeta

```
 6 rkse
16 qzmf
17 asle
18 ymzk
23 jfee
27 jngn
33 pvih
38 sqnm
39 xvmt
40 ezcw
41 rdlu
43 vyth
44 grpa
50 rewz
58 wyvc
```

Trabalho Prático 2

Exemplo Passo a Passo

Depois, devemos agrupar esses planetas em cada mês, respeitando o tempo máximo de visita por mês (60, nesse exemplo)
(cada final de mês é representado pelo separador =======)

```
 6 rkse ( 6)
16 qzmf (22)
17 asle (39)
18 ymzk (57)
============
23 jfee (23)
27 jngn (50)
============
33 pvih (33)
============
38 sqnm (38)
============
39 xvmt (39)
============
40 ezcw (40)
============
41 rdlu (41)
============
43 vyth (43)
============
44 grpa (44)
============
50 rewz (50)
============
58 wyvc (58)
```

Trabalho Prático 2

Exemplo Passo a Passo

Por fim, os planetas em cada mês devem ser ordenados alfabeticamente

```
17 asle
16 qzmf
 6 rkse
18 ymzk
========
23 jfee
27 jngn
========
33 pvih
========
38 sqnm
========
39 xvmt
========
40 ezcw
========
41 rdlu
========
43 vyth
========
44 grpa
========
50 rewz
========
58 wyvc
```

Análise Experimental

Análise Experimental

Motivação

É interessante ter evidências empíricas da sua análise assintótica teórica

Muitos dos que erraram a análise teórica no TP1, poderiam ter acertado caso observassem o comportamento experimental do programa

Assim, os alunos da turma **TN** deverão apresentar um gráfico na documentação do **TP2** com a análise experimental do tempo de execução do programa

Aqui, apresentaremos uma maneira de medir o tempo usando comandos do Bash (disponível em sistemas Unix e Windows 10).

Caso não tenha acesso ao Bash, envie uma mensagem para o monitor

Análise Experimental

Medindo o tempo de execução com bash

O bash possui uma keyword reservada chamada **time** que mede o tempo de execução de um programa

A maioria das distribuições também possui um executável com mesmo nome, **time** (/usr/bin/time) que faz a mesma coisa porém tem o comportamento um pouco diferente

Ambos dividem a medição em:

- **real**: tempo do relógio gasto do início até a execução do programa
- **user**: tempo que o processador gastou executando o programa
- **sys**: tempo que o processador gastou executando chamadas de sistema

Estamos interessados no tempo de **user** (o mesmo programa com a mesma entrada pode medir diferentes tempo **real** dependendo da carga do processador).

Estou apresentando as duas opções pois é possível que uma distribuição diferente contenha somente um dos dois

Análise Experimental

Usando time

time

```
$ time ./tp1 < tests/00.in > /dev/null

real    0m0,005s
user    0m0,004s
sys     0m0,001s
$ time ./tp1 < tests/29.in > /dev/null

real    0m0,865s
user    0m0,788s
sys     0m0,076s
```

/usr/bin/time

```
$ /usr/bin/time -p ./tp1 < tests/00.in > /dev/null
real 0.00
user 0.00
sys 0.00
$ /usr/bin/time -p ./tp1 < tests/29.in > /dev/null
real 0.88
user 0.81
sys 0.07
```

./tp1 > /dev/null redireciona o *stdout* do programa ./tp1 para o arquivo /dev/null que é um “buraco negro” onde a saída é descartada
./tp1 < exemplo.txt redireciona o *stdin* do programa ./tp1 para o arquivo exemplo.txt

Análise Experimental

Filtrando somente a saída do tempo **user**

time

```
$ TIMEFORMAT="%3U"
$ time ./tp1 < tests/29.in > /dev/null
0,820
```

/usr/bin/time

```
$ /usr/bin/time -f "%U" ./tp1 < tests/29.in > /dev/null
0.85
```

mais opções para **time** nos comandos `help time`e `man bash`
mais opções para **/usr/bin/time** no comando `man /usr/bin/time`

Análise Experimental

Gerando saídas aleatórias

Queremos fazer uma curva que mostra o tempo que o programa gasta para várias entradas diferentes

Para isso, precisamos executar o programa para entradas aleatórias de diferentes tamanhos

Vamos estudar a partir de agora um exemplo onde iremos medir o tempo de execução do BubbleSort experimentalmente

Análise Experimental

Gerando saídas aleatórias - Código BubbleSort e Gerador Aleatório

./bubble

```c
#include <stdio.h>
#include <stdlib.h>
void swap(int *a, int *b) {
  int c = *a; *a = *b; *b = c;
}
void bubble_sort(int *v, int n) {
  for (int i = 0; i < n; ++i)
    for (int j = i + 1; j < n; ++j)
      if (v[j] < v[i])
        swap(v + i, v + j);
}
void print_v(int *v, int n) {
  for (int i = 0; i < (n - 1); ++i)
    printf("%d ", v[i]);
  printf("%d\n", v[n - 1]);
}
int main() {
  int n;
  scanf("%d", &n);
  int *v = (int *)malloc(n*sizeof(int));
  for (int i = 0; i < n; ++i) scanf("%d", v + i);
  bubble_sort(v, n);
  print_v(v, n);
  free(v);
}
```

./rand_gen

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main(int argc, char *argv[]) {
  if (argc <= 1) {
    printf("parametro int(x) necessario\n");
    return 0;
  }
  int x = atoi(argv[1]);
  srand(time(NULL));
  printf("%d\n", x);
  for (int i = 0; i < x; ++i) {
    // random number from 0 to 199
    int rand_number = rand()%200;
    printf("%d ", rand_number);
  }
  printf("\n");
  return 0;
}
```

Análise Experimental

Saída rand_gen

rand_gen cria entradas aleatórias para que *bubble* ordene

```
$ ./rand_gen 3
3
27 84 89
$ ./rand_gen 5
5
196 34 116 26 172
$ ./rand_gen 10
10
136 99 150 100 188 65 147 69 25 161
```

Podemos redirecionar a saída de *rand_gen* para a entrada de *bubble* usando o pipe `|` da seguinte maneira:

```
$ ./rand_gen 3 | ./bubble
74 88 144
$ ./rand_gen 5 | ./bubble
23 60 150 163 183
$ ./rand_gen 10 | ./bubble
20 39 62 84 95 127 152 169 177 193
```

Análise Experimental

Gerando várias entradas

A grande vantagem desse processo é que agora podemos automatizar a construção de entradas para o programa bubble

Usando um simples loop, conseguimos executar o programa bubble para entradas aleatórias de vários tamanhos diferentes

```
$ for i in {5..25..5}; do
>     echo "Ordenando entrada de tamanho $i"
>     ./rand_gen $i | ./bubble
> done
Ordenando entrada de tamanho 5
33 37 39 49 105
Ordenando entrada de tamanho 10
5 33 37 39 41 49 82 96 105 107
Ordenando entrada de tamanho 15
5 29 33 37 39 41 41 49 59 82 96 105 107 132 170
Ordenando entrada de tamanho 20
5 8 29 33 37 39 41 41 43 49 59 82 96 105 107 117 132 137 170 172
Ordenando entrada de tamanho 25
3 5 8 29 33 37 39 41 41 43 46 49 59 60 82 96 105 107 113 117 132 137 170 172 191
```

A sintaxe do for é {valor_inicial..valor_final..tamanho_passo}

Análise Experimental

Medindo tempo de várias entradas

Por fim, basta adicionar a medição de tempo com o comando **time**

```
$ TIMEFORMAT="%U"
$ for i in {5000..25001..10000}; do
>     echo "Medindo tempo de ordenação de um vetor com tamanho $i"
>     ./rand_gen $i | { time ./bubble > /dev/null; }
> done
Medindo tempo de ordenação de um vetor com tamanho 5000
0.080
Medindo tempo de ordenação de um vetor com tamanho 15000
0.472
Medindo tempo de ordenação de um vetor com tamanho 25000
1.276
```

o comando ./bubble > /dev/null redireciona a saída de bubble para o arquivo /dev/null, que é uma espécie de arquivo “buraco-negro”. Isto é feito para que o vetor ordenado (a saída de bubble) não ocupe espaço na tela, e torne mais difícil nossa tarefa de ver o tempo

Análise Experimental

Medindo tempo de várias entradas

Por fim, basta adicionar a medição de tempo com o comando **/usr/bin/time**

```
$ for i in {5000..25001..10000}; do
>     echo "Medindo tempo de ordenação de um vetor com tamanho $i"
>     ./rand_gen $i | /usr/bin/time -f %U ./bubble > /dev/null
> done
Medindo tempo de ordenação de um vetor com tamanho 5000
0.08
Medindo tempo de ordenação de um vetor com tamanho 15000
0.47
Medindo tempo de ordenação de um vetor com tamanho 25000
1.28
```

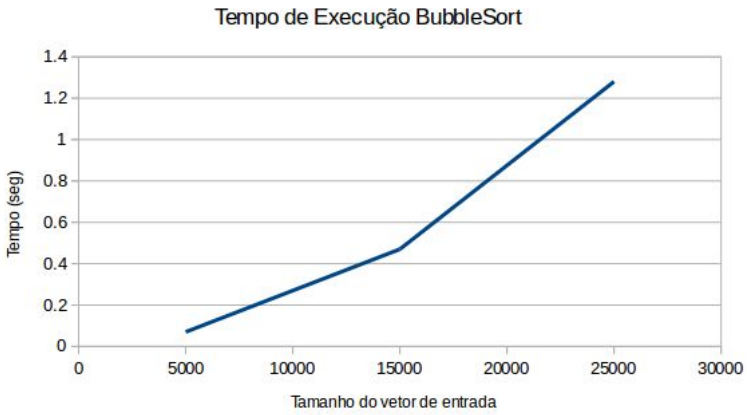
o comando ./bubble > /dev/null redireciona a saída de bubble para o arquivo /dev/null, que é uma espécie de arquivo “buraco-negro”. Isto é feito para que o vetor ordenado (a saída de bubble) não ocupe espaço na tela, e torne mais difícil nossa tarefa de ver o tempo

Análise Experimental

Criando um arquivo csv e um gráfico

Podemos criar um arquivo csv* com duas colunas, tamanho de entrada e tempo de execução através da simples troca da string impressa no comando echo

```
$ for i in {5000..25001..10000}; do
>     echo -n "$i;"
>     ./rand_gen $i | /usr/bin/time -f %U ./bubble > /dev/null
> done &> tempos.csv
$ cat tempos.csv
5000;0.07
15000;0.47
25000;1.28
```



O comando &> exemplo.txt redireciona a saída stderr (utilizada pelo time) e a saída stdout (utilizada pelo echo) para o arquivo exemplo.txt

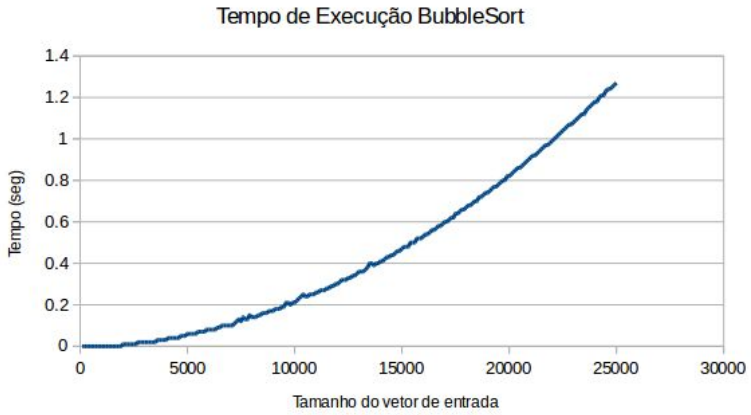
* utilizei **;** como separador ao invés de **,** pois alguns sistemas podem usar a vírgula como seperador de cassas decimais do tempo

Análise Experimental

Criando um arquivo csv e um gráfico decente

O gráfico anterior utilizava somente 3 pontos, e por isso, não ficava claro o comportamento da curva. Uma simples alteração do loop for gera um gráfico melhor

```
$ for i in {100..25001..100}; do
>     echo -n "$i;"
>     ./rand_gen $i | /usr/bin/time -f %U ./bubble > /dev/null
> done &> tempos_250_amostras.csv
```



Use pelo menos 10 pontos para fazer o gráfico de sua documentação

Análise Experimental

Considerações finais

O arquivo csv pode ser lido pelo Excel, Libre Office, e Google Docs[1]

Esses programas também são capazes de criar os gráficos

Vocês devem criar um gerador de entradas personalizado para o TP2 (deve gerar também a string com o nome dos planetas, além de imprimir no ínicio os valores P, T, x)
Esse gerador não precisa ser entregue no Moodle

Em casos de dificuldades, envie um e-mail para matheusad95 [at] gmail.com

1: sheets.google.com