Algoritmos e Estrutura de Dados II (2018-2)

Trabalho Prático 2

Heitor Lourenço Werneck
heitorwerneck@hotmail.com

8 de Março de 2019

Conteúdo

1 Introdução

O trabalho apresentado nesse documento consiste na avaliação do método de ordenação quicksort (recursivo) considerando as seguintes entradas:

- Os elementos a serem ordenados são inteiros armazenados em um vetor de tamanho N.
- Os elementos a serem ordenados são inteiros armazenados em uma lista duplamente encadeada com N elementos.
- Os elementos a serem ordenados são registros armazenados em um vetor de tamanho N.

Cada registro contém:

- Um inteiro, a chave para ordenação.
- Dez cadeias de caracteres (strings), cada uma como 200 caracteres.
- 1 booleano.
- 4 números reais.

As entradas para o programa são:

1. semente.
2. arquivo de entrada.
3. arquivo de saída.

No arquivo de entrada a primeira linha contém o número de valores que se seguem e os próximos valores são o tamanho das sequencias(N) a serem geradas.

Também foi necessário implementar funções para criação dos conjuntos de elementos aleatórios.

1.1 Funcionamento do algoritmo (Visão geral)

Com o arquivo de entrada, a primeira coisa é ler a primeira linha para saber quantos valores de tamanho de vetor(N's) se seguem.

Apos isso, o programa prossegue lendo os N's, e a cada leitura de um N o programa irá entrar em um laço de repetição que ira ser repetido 5 vezes (definido na especificação do trabalho), e dentro desse laço ocorre a atribuição da semente (a semente se diferencia para cada um das 5 repetições para gerar valores diferentes) a geração de valores randômicos de cada tipo de lista é feita assim como a ordenação, as estatísticas da ordenação são guardadas. No final das 5 repetições é feito uma media de cada algoritmo para cada tipo de lista.

Já no final do algoritmo todos os N's terão suas estatísticas no arquivo de saída.

2 Implementação

2.1 Estruturas de dados

As estruturas utilizadas foram:

1. Vetor de inteiro (`int *vint = malloc(sizeof(int)*N)`);
2. Lista duplamente encadeada de inteiros

```
struct TLista
{
  struct TCelula* inicio;
  int tamanho;
  struct TCelula* fim;
};

struct TCelula
{
  struct TCelula* ant;
  int valor;
  struct TCelula* prox;
};

typedef struct TLista lista;
typedef struct TCelula celula;
```

3. vetor de struct

```
#define numstr 10
#define strsize 200
#define realn 4
#define booltotaln 2
typedef char bool;
typedef struct elem{
  int ch;//Chave para ordenação
  char str[numstr][strsize];//numstr strings, cada uma com strsize caracters
  bool b; // booleano
  float r[realn];//numeros reais
}elem;
```

4. Informações sobre algoritmo (algoinfo) Essa estrutura de dados foi necessária para guardar informações do algoritmo quicksort a ser executado para depois ser possível guardar as estatísticas, essa estrutura guarda:

   - Numero de ativações
   - Copias ou trocas
   - Comparações
   - Tempo de usuário
   - Tempo do sistema
   - Tempo total

2.2 Gerador aleatório

O gerador aleatório se diferencia para cada tipo de dado:

int Para gerar um inteiro aleatório foi feito uma logica simples `rand()%(size*randn)`, isso basta para não ter grande variabilidade de valores para pequenos valores de N e se for maior terá maior variabilidade.

string Para a geração de string é necessário um conjunto de caracteres permitidos na geração aleatória, o conjunto de caracteres utilizado foi o abaixo:

```
const char charset[] =
  "0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

Basta fazer o modulo do tamanho do conjunto de caracteres.

```
charset[rand()%sizeof(charset)];
```

bool a geração do booleano basta utilizar o modulo de 2 para obter-se valores 0 ou 1.

```
rand()%booltotaln;
```

float para gerar um float basta dividir o rand()(um valor inteiro) por um valor float, a lógica seguinte faz com que seja possível um numero float aleatório.

```
(float)rand()/(float)(RAND_MAX/(size*randn));
```

As seguintes funções são dos geradores para cada tipo de estrutura de dados.

```
void vint_random(int *vint, size_t size);
void lint_random(lista *lint, size_t max);
void struct_random(elem *vstruct, size_t size);
```

2.3 Semente

Para a utilização da semente só foi necessário uma logica que durante as 5 repetições tivesse uma diferenciação na semente, como especificado.

```
for(int nsed=0;nsed<repeat;nsed++){
  srand(atoi(argv[seed])+nsed*randn);
  ...
    }
```

2.4 Contador de tempo

Para contar o tempo que cada quicksort utiliza foi necessário a criação da seguinte função:
`void timeusage(struct rusage *resources,char start,algoinfo *ai);`.

- a estrutura rusage é a estrutura que vai guardar o tempo na forma "original".
- o carácter start é uma variável utilizada para saber se é para começar a contar ou parar, se start for verdadeiro então começara a contar o tempo, se for falso ira parar de contar.
- a estrutura algoinfo irá guardar o tempo total, de usuário e do sistema.

2.5 Quicksort

Como há 3 estruturas de dados diferentes será necessário 3 algoritmos de quicksort diferentes.

2.5.1 Vetor de inteiro

```
int particiona_int(int vetor[], int inicio, int fim,algoinfo* ai);
void quick_int(int vetor[], int inicio, int fim, algoinfo* ai);
```

Aqui foi feito o quicksort padrão que normalmente é feito para vetor de inteiros. A função particiona_int reorganiza o vetor em torno de um pivô escolhido, que no caso dessa implementação o pivô é o primeiro numero, e apos isso vai dividindo e reorganizando o vetor ate chegar em elementos individuais e no final o resultado será o vetor organizado.

Na reorganização é utilizado duas variáveis para movimentação pelo vetor, que são as variáveis **esq** e **dir** e a variável **esq** inicia com o índice inicial e a variável **dir** inicia com o índice final, um laço é repetido enquanto a variável **esq** for menor que a **dir**.

Dentro desse laço existe um laço para movimentação da variável **esq** que enquanto o valor na posição **esq** do vetor for menor ou igual ao pivô e enquanto a **esq** for menor que **fim** o laço irá movimentar uma posição positivamente na variável **esq** ou seja aumentar 1 na **esq**.

Dentro desse laço também existe um laço para movimentação da variável **dir** que enquanto o valor na posição **dir** do vetor for maior que o pivô o laço irá movimentar uma posição negativamente na variável **dir** ou seja diminuir 1 em **dir**.

Depois se o **esq** ainda for menor que **dir** então o valor nesses índices terão valores trocados.

Então quando acaba todo o while e **esq** é maior ou igual a **dir** então a posição inicial recebe o valor no índice **dir** e o valor no índice **dir** recebe o valor do pivô. E no final o retorno será **dir** que é o índice que irá dividir o vetor.

Durante todo esse processo as informações do algoritmo são armazenadas na variável **ai**.

```
int particiona_int(int vetor[], int inicio, int fim,algoinfo* ai)
{
  int esq, dir;
  int pivo, aux;
  esq = inicio;
  dir = fim;
  pivo = vetor[inicio];
  while(esq<dir)
  {
    while(vetor[esq] <= pivo && esq<fim){   // vetor[esq] <= pivo
      esq++;
      ai->comparacoes++;
    }

    while(pivo < vetor[dir]){    //  vetor[dir] > pivo
      dir--;
      ai->comparacoes++;
    }

    if(esq < dir)
    {
      aux = vetor[esq];          // troca vetor[esq] com vetor[dir]
      vetor[esq] = vetor[dir];
      vetor[dir] = aux;
      ai->troca++;
    }

    ai->comparacoes+=comparasionnumber;
  }
```

```
  ai->comparacoes++;
  vetor[inicio] = vetor[dir];
  vetor[dir] = pivo;
  return dir;                    //retorna dir, que é o índice que vai dividir o vetor
}


void quick_int(int vetor[], int inicio, int fim, algoinfo* ai)
{
  ai->ativacoes++;
  int pivo;
  if(inicio < fim)
  {
    pivo = particiona_int(vetor,inicio,fim,ai); // encontra um pivo que "divide" o vetor em dois
    quick_int(vetor, inicio, pivo-1, ai); // realiza a partição para a parte da esquerda
    quick_int(vetor, pivo+1, fim, ai);  // e realiza a partição para a parte de direita
  }
}
```

2.5.2 Lista duplamente encadeada de inteiro

```
celula* particiona_int_l(celula* inicio, celula* fim,algoinfo* ai);
void quick_int_l(celula* inicio,celula* fim,algoinfo* ai);
```

O quicksort para lista duplamente encadeada encontra algumas pequenas diferenças como nos tipos de dados utilizados para se fazer a ordenação. A função particiona_int_l reorganiza a lista em torno de um pivô escolhido, que no caso dessa implementação o pivô é o primeiro numero, e apos isso vai dividindo e reorganizando o vetor ate chegar em elementos individuais e no final o resultado será o vetor organizado.

Para a equivalência de **esq**<**dir** como no quicksort para vetor de inteiros foi feito um macro que irá resolver o problema.

```
#define LLTR(left, right)  (right != NULL && left != right && left != right->prox)
```

Esse macro irá receber dois ponteiros de celula e irá dizer se o ponteiro da esquerda é menor que o segundo em relação a posição na lista.

Na reorganização é utilizado duas variáveis para movimentação pela lista, que são as variáveis **esq** e **dir** e a variável **esq** inicia apontando para a celula inicial e a variável **dir** inicia apontando para a celula final, um laço é repetido enquanto a variável **esq** for menor que a *dir*(`LLTR(esq,dir)`).

Dentro desse laço existe um laço para movimentação da variável **esq** que enquanto o valor na **esq** for menor ou igual ao pivô e enquanto a **esq** for menor que **fim** a **esq** irá para a próxima celula.

Dentro desse laço também existe um laço para movimentação da variável **dir** que enquanto o valor de **dir** for maior que o pivô o laço irá para a celula anterior a **dir**.

Depois se o **esq** ainda for menor que **dir** então o valor nessas celulas serão trocados.

Então quando acaba todo o while e **esq** é maior ou igual a **dir** então a celula **inicio** recebe o valor de **dir** e o valor de **dir** recebe o valor do pivô. E no final o retorno será **dir** que é a celula que irá dividir a lista.

Durante todo esse processo as informações do algoritmo são armazenadas na variável **ai**.

```
celula* particiona_int_l(celula* inicio, celula* fim,algoinfo* ai){
  int pivo = inicio->valor,aux;
  celula* dir = fim,*esq = inicio;
  while(LLTR(esq,dir)){
    while(esq->valor <= pivo && LLTR(esq,fim)){
      esq=esq->prox;
```

```
      ai->comparacoes++;
    }
    while(pivo < dir->valor){
      dir=dir->ant;
      ai->comparacoes++;
    }
    if(LLTR(esq,dir))
    {
      aux = dir->valor;
      dir->valor = esq->valor;
      esq->valor = aux;
      ai->troca++;
    }
    ai->comparacoes+=comparasionnumber;
  }
  ai->comparacoes++;
  inicio->valor = dir->valor;
  dir->valor = pivo;
  return dir;
}

void quick_int_l(celula* inicio,celula* fim,algoinfo* ai){
  ai->ativacoes++;
  if (LLTR(inicio,fim)){
    celula* pivo =  particiona_int_l(inicio,fim,ai);
    quick_int_l(inicio, pivo->ant,ai);
    quick_int_l(pivo->prox, fim,ai);
  }
}
```

2.5.3 Vetor de registro

```
int particiona_struct(elem* rgs, int inicio, int fim,algoinfo* ai);
void quick_struct(elem* rgs, int inicio, int fim, algoinfo* ai);
```

Este quicksort é bem parecido com o padrão que normalmente é feito para vetor de inteiros. A função particiona_struct reorganiza o vetor em torno de um pivô escolhido, que no caso dessa implementação o pivô é a chave do primeiro elemento do vetor, e apos isso vai dividindo e reorganizando o vetor ate chegar em elementos individuais e no final o resultado será o vetor organizado.

Para fazer o quicksort em uma struct basta selecionar um de seus elementos e utilizá-lo como um valor de referencia para ordenação. Nesse caso o valor utilizado foi o campo **ch** que é um inteiro.

Quase tudo ira ser como na ordenação de vetor de inteiros. Agora para acessar um índice em certa posição basta também utilizar o acesso ao elemento **ch** que será o valor de referencia.

Na troca de valores o registro deverá ser trocado completamente e necessitara de um auxiliar do tipo **elem** para a troca ser efetuada. Diferente da troca das outras estruturas que só necessitavam de trocar um inteiro.

Durante todo esse processo as informações do algoritmo são armazenadas na variável **ai**.

```
int particiona_struct(elem* rgs, int inicio, int fim,algoinfo* ai)
{
  int esq, dir;
  elem pivo,aux;
  esq = inicio;
```

```
  dir = fim;
  pivo = rgs[inicio];
  while(esq<dir)
  {
    while(rgs[esq].ch <= pivo.ch && esq<fim){
      esq++;
      ai->comparacoes++;
    }

    while(pivo.ch < rgs[dir].ch){
      dir--;
      ai->comparacoes++;
    }

    if(esq < dir)
    {
      aux = rgs[esq];
      rgs[esq] = rgs[dir];
      rgs[dir] = aux;
      ai->troca++;
    }

    ai->comparacoes+=comparasionnumber;
  }

  ai->comparacoes++;
  rgs[inicio] = rgs[dir];
  rgs[dir] = pivo;
  return dir;                    //retorna dir, que é o índice que vai dividir o rgs
}

void quick_struct(elem* rgs, int inicio, int fim,algoinfo* ai)
{
  ai->ativacoes++;
  int pivo;
  if(inicio < fim)
  {
    pivo = particiona_struct(rgs,inicio,fim,ai); // encontra um pivo que "divide" o vetor em dois
    quick_struct(rgs, inicio, pivo-1, ai); // realiza a partição para a parte da esquerda
    quick_struct(rgs, pivo+1, fim, ai);  // e realiza a partição para a parte de direita
  }
}
```

2.6 Saida dos resultados

A função a seguir é responsável por imprimir no arquivo de saída os dados de cada algoritmo. Ela recebe uma estrutura do tipo algoinfo que contem informações do algoritmo e ira imprimir todas informações.

```
void printalgoinfo(const algoinfo *ai,FILE* f);
```

3 Resultados e Discussões

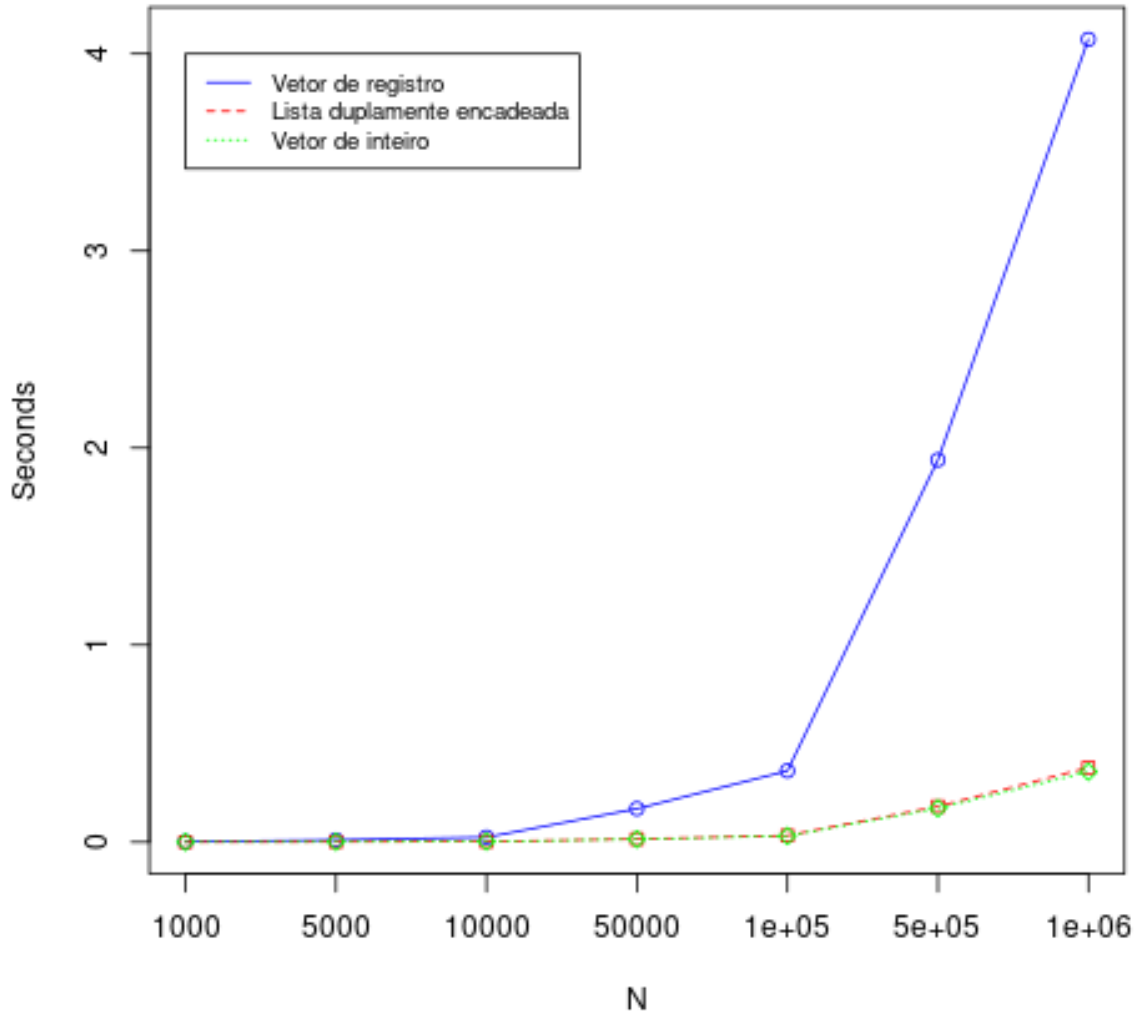
N
1000
5000
10000
50000
100000
500000
1000000

Tabela 1: Entrada teste

Tabela 2: Saida, tempo total em segundos de cada algoritmo em media.

	N						
	1000	5000	10000	50000	100000	500000	1000000
Vetor de inteiro	0.00025	0.00124	0.00249	0.01464	0.03074	0.17130	0.35664
Lista duplamente encadeada	0.00026	0.00130	0.00268	0.01535	0.03294	0.18048	0.37783
Vetor de registro	0.00157	0.01039	0.02445	0.16934	0.36143	1.93895	4.07320

3.0.1 Vetor de registro



É possível ver pelo gráfico que o comportamento da lista duplamente encadeada e bastante similar ao vetor de inteiros, porém o vetor de registros tem um comportamento bastante diferente pois seu tempo de ordenação cresce de maneira descomunal.

É possível entender o porque dessa velocidade através da leitura do tamanho do registro. Fazendo o calculo:

$$\begin{aligned} Seja: \\ sizeof(int) = 4\ bytes \\ sizeof(char) = sizeof(bool) = 1\ bytes \\ sizeof(float) = 4\ bytes \\ sizeof(elem) = sizeof(int) + sizeof(char) * 10 * 200 + sizeof(bool) + sizeof(float) * 4 \\ \therefore sizeof(elem) = 2021\ bytes \end{aligned} \tag{1}$$

O resultado é que o registro utiliza 2021 bytes, logo é possível entender que o custo de troca/copia de dados dessa estrutura é bastante custoso, logo isso é um dos fatores que tornam esse algoritmo mais lento pois a cada troca ele custa muito processamento. A diferença de tamanho do registro a ser trocado para as outras estruturas é significativamente maior, pois nas outras estruturas a troca é feita com variáveis

inteiras que tem tamanho de 4 bytes.

O registro é 505.25 vezes maior que um inteiro ($2021/4 = 505.25 bytes$). Isso fala bastante sobre o custo de troca.

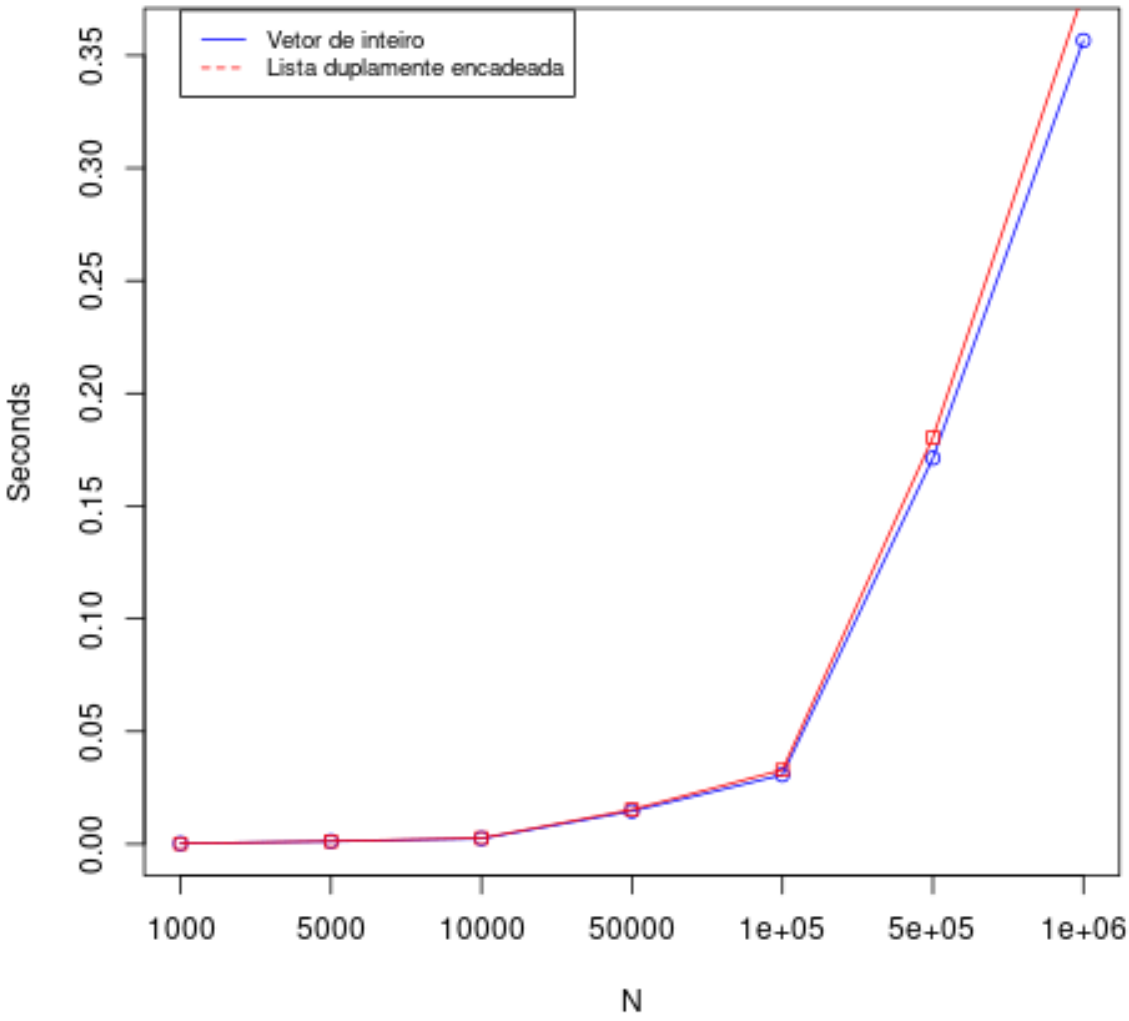
Será feito outro calculo de quanto de memoria é utilizado para guardar 1000000 registros(pior caso de entrada do arquivo utilizado nesse documento).

$$\begin{aligned} N &= 1000000 \\ sizeof(elem) &= 2021\ bytes \\ sizeof(elem) * N &= 20210000000\,\mathrm{B} \\ \therefore sizeof(elem) * N &= 1927.37570\,\mathrm{MiB} \end{aligned} \tag{2}$$

É possível ver que o vetor de registro custa muita memoria com $N = 1000000$, isso também pode atrapalhar o desempenho visto que o processador fica um tempo ocioso esperando os dados chegarem da memoria, isso é também conhecido como gargalo de Von Neumann.

3.0.2 Vetor de inteiros e lista duplamente encadeada

Como na analise anterior o vetor de registro apresenta uma grande disparidade de performance com as duas outras estruturas a analise delas serão feitas nessa subseção para a melhor analise das duas estruturas.



Com esse gráfico um pouco mais próximo é possível observar que a lista duplamente encadeada (lint) mesmo que seja bastante próxima em performance da lista de inteiros (vint) ainda é mais lenta.

Como o *vint* é uma estrutura simples sua performance é extremamente alta. É possível ver na tabela de media de tempo (2) no pior caso ($N = 1000000$) a diferença de tempo de execução das duas estruturas é $\Delta T = T_{lint} - T_{vint} = 0.37783 - 0.35664 = 0.02119s$, ou seja, 21.19ms.

Sendo no pior caso a ordenação de *vint* 21.19ms mais rápida que a ordenação em *lint/(mais rápida também para todos outros N's), mesmo que o /vint* seja mais rápido por alguns milissegundos esse tempo em um processo critico pode conferir muitas vantagens.

Um dos fatores que fazem a ordenação da *lint* ser mais lento que o do *vint* é porque as comparações na lista duplamente encadeada são mais custosas pois há mais comparações.(Por exemplo na ordenação de *lint* há a utilização do macro LLTR que trás mais custo ao código)

O principal fator da *lint* ser mais lento que o *vint* é devido a estrutura que é utilizada. A lista duplamente encadeada é um pouco maior que o vetor de inteiros e também mais complexa por isso no final o resultado do vetor de inteiros é melhor, pois é uma estrutura mais simples e não há tanta complexidade de manipulação da estrutura no quicksort.

4 Conclusão

Foi possível observar através desse trabalho as diferenças de performance do algoritmo Quicksort entre estruturas de dados diferentes. O vetor de registro foi a estrutura que apresentou pior performance, as duas outras apresentaram resultados bem semelhantes.

Mas o vetor de inteiros e a lista duplamente encadeada possuem uma distinção, o vetor de inteiros ainda é mais rápido que a lista duplamente encadeada, mesmo que por pouca diferença cada microssegundo é importante em processos críticos.

Visto que o vetor de registro utiliza bastante memoria, como foi mostrado, o gargalo de von Neumann explica a performance baixa da ordenação nessa estrutura.

A estrutura que mais necessitou de esforço de implementação foi a lista duplamente encadeada visto que há diversas formas de fazer o algoritmo e a decisão de qual implementação seria a melhor para esse trabalho foi uma escolha difícil. Poderia ter utilizado diversas abordagens como por exemplo gastar menos poder computacional nas condições e criar uma estrutura de dados que guarda o índice das celulas, foi visto que essa abordagem pode ter alguns benefícios como por exemplo a velocidade no processamento porém é uma troca de memoria por velocidade de processamento e a decisão final foi que o algoritmo mais próximo do original mostrado nas aulas e que gasta menos memoria seria o melhor que se encaixaria nessa situação.

Então o resultado final é que o vetor de inteiros é o mais rápido, a lista duplamente encadeada é segunda mais rápida e o vetor de registro o mais lento.

5 Bibliografia

`https://whatis.techtarget.com/definition/von-Neumann-bottleneck`