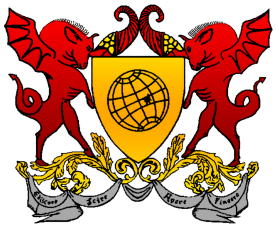
Universidade Federal de Viçosa
Campus Florestal
Ciência da Computação



Documentação do 3º e 4º Trabalho Prático

Algoritmos e Estrutura de Dados

Grupo Victor (02658) e Adriano (02640)
Professor Thais Regina de Moura Braga Silva

Belo Horizonte, 2 de dezembro de 2016

Sumário

1 Organização e Planejamento **1**

2 Organização de pastas do projeto **1**

2.1 build . . . 1
2.2 cmake . . . 1
2.3 docs . . . 2
2.4 src . . . 2
2.4.1 quicksorts . . . 2
2.4.2 heapsorts . . . 2
2.4.3 main . . . 2

3 Licença **3**

4 Referência **4**

4.1 pilha . . . 4
4.1.1 header . . . 4
4.1.2 source . . . 4
4.1.3 Descrição . . . 5
4.2 inssort . . . 6
4.2.1 header . . . 6
4.2.2 source . . . 7
4.2.3 Descrição . . . 7
4.3 lista . . . 8
4.3.1 header . . . 8
4.3.2 source . . . 9
4.3.3 Descrição . . . 9
4.4 insquick . . . 10
4.4.1 header . . . 11
4.4.2 source . . . 11
4.4.3 Descrição . . . 12
4.5 iteintquick . . . 13
4.5.1 header . . . 13
4.5.2 source . . . 13
4.5.3 Descrição . . . 14
4.6 itequick . . . 15
4.6.1 header . . . 15

4.6.2 source . . . 15
4.6.3 Descrição . . . 16
4.7 medquick . . . 16
4.7.1 header . . . 16
4.7.2 source . . . 17
4.7.3 Descrição . . . 18
4.8 recquick . . . 19
4.8.1 header . . . 19
4.8.2 source . . . 19
4.8.3 Descrição . . . 20
4.9 recintquick . . . 21
4.9.1 header . . . 21
4.9.2 source . . . 21
4.9.3 Descrição . . . 22
4.10 heapsort . . . 22
4.10.1 header . . . 22
4.10.2 source . . . 23
4.10.3 Descrição . . . 24

5 Conclusão **24**

6 Agradecimentos **26**

Resumo

Com este trabalho temos como objetivo principal estudar e demonstrar a diferença no comportamento assintótico da complexidade de dois dos algoritmos de ordenação mais eficientes, HeapSort e QuickSort, sendo que o segundo foi implementado de 6 diferentes formas.

1 Organização e Planejamento

O desenvolvimento se deu no ambiente de controle de versão GitHub, para que pudéssemos programar sem a necessidade de reunirmos pessoalmente e de maneira mais organizada.

O projeto foi dividido em três etapas de produção:

1. Na primeira prepararíamos a documentação e os TADS necessários(TItem, TLista e TPilha).

2. Em seguida fizemos os arquivos de *E/S*(Entrada e Saída), e a função principal. Permitindo assim testarmos o código com decorrer de sua evolução.

3. Depois implementamos cada diferente "versão"do QuickSort.

4. Por fim implementamos o HeapSort.

2 Organização de pastas do projeto

2.1 build

A pasta build é responsável pelo armazenamento dos arquivos resultantes da pré-compilação feita pelo cmake e da compilação feita logo em seguida.

2.2 cmake

A pasta cmake é onde encontrariam-se os arquivos de configuração do cmake (config.h.in ou FindLib.cmake), contudo não necessitamos utilizá-los.

2.3 docs

A pasta docs possui os pdfs de documentação, proposta do TP e o slide da aula referente a quicksort.

2.4 src

A pasta src é composta apenas por duas outras pastas (main e quicksorts):

2.4.1 quicksorts

A pasta quicksorts possui os arquivos de cabeçalho e de código-fonte de todas as implementações de quicksort propostas. Ela também é responsável pela criação da biblioteca estática (libQuickSorts.a), que é composta pelos código encontrado nessa.

2.4.2 heapsorts

A pasta heapsorts possui os arquivos de cabeçalhos e de código-fonte da implementação de heapsort. Esta, também, é responsável pela criação da biblioteca estática (libHeapSorts.a), que é composta pelos códigos que nela se encontram.

2.4.3 main

A pasta main é onde se encontra o arquivo main.c que possui a função principal do projeto. E é a pasta responsável pela compilação do executável (RunQuickSorts) utilizando-se a linkagem com a biblioteca criada na pasta vizinha.

3 Licença

4 Referência

O programa foi organizado para que a compilação da biblioteca de QuickSort's e da parte executável do programa fossem separadas.

A biblioteca é composta pela união de vários arquivos, sendo que cada um possui o código fonte de um dos "tipos"de QuickSort's ou de um dos TADS utilizados na implementação desses:

4.1 pilha

TAD amplamente utilizado nas implementações de QuickSort's iterativos da biblioteca. Responsável por armazenar valores a serem reposicionados na lista.

4.1.1 header

```
1  #ifndef PILHA_H_INCLUDED
2  #define PILHA_H_INCLUDED
3
4  typedef struct{
5  unsigned long int *vecI;
6  int top;
7  int tam;
8  }TPilha;
9
10 void iniPilha(TPilha* p, int tam);
11 int empilha(TPilha* p, int a);
12 int desempilha(TPilha* p);
13 int destroyTPilha(TPilha* p);
14
15 #endif // PILHA_H_INCLUDED
```

4.1.2 source

```
1  #include "pilha.h"
2
3  void iniPilha(TPilha* p, int tam){
4  p->top = 0;
```

```
5     p->tam = tam;
6     p->vecI = (int*)malloc((tam*4)*sizeof(int));
7     }
8     int empilha(TPilha* p, int a){
9     if(p->top == p->tam)
10    return 0;
11    p->vecI[p->top++] = a;
12    return 1;
13    }
14    int desempilha(TPilha* p){
15    if(p->top == 0)
16    return 0;
17    return p->vecI[--p->top];
18    }
19
20    int destroyTPilha(TPilha* p){
21    free(p->vecI);
22    return 1;
23    }
```

4.1.3 Descrição

- **TPilha**

  1. *unsigned long int **vecI*
     Arranjo de itens da pilha.
  2. *int top*
     Próxima posição do arranjo a ser empilhado um elemento.
  3. *int tam*
     Responsável por armazenar o tamanho da pilha.

- **void iniPilha(TPilha* p, int tam)**

  Função que inicializa uma pilha, alocando dinamicamente seu valor, de acordo com o tamanho passado.

  1. *TPilha* p*
     Ponteiro para a pilha a ser inicializada.
  2. *int tam*
     Parâmetro que armazena o tamanho da pilha a ser inicializada.

- **void empilha(TPilha* p, int a)**

  Também conhecida como *push*, essa função é responsável por posicionar um único elemento na lista.

  1. *TPilha *p*
     Ponteiro para a pilha a ser empilhada.
  2. *int a*
     Parâmetro que indica o valor a ser empilhado.

- **void desempilha(TPilha* p)**

  Também conhecida como *pop*, essa função é responsável por retirar um único elemento na lista.

  1. *TPilha *p*
     Parâmetro que armazena o próximo elemento a ser empilhado.

- **void destroyPilha(TPilha* p)**

  Função que destrói uma pilha, desalocando dinamicamente seu valor.

  1. *TPilha *p*
     Ponteiro para a pilha a ser destruída.

4.2 inssort

Implementação do algoritmo de ordenação de inserção. Utilizado juntamente com uma implementação de quicksort com o intuito de reduzir o tempo de execução.

4.2.1 header

```
1  #ifndef INSSORT_H_INCLUDED
2  #define INSSORT_H_INCLUDED
3  #include "insquick.h"
4  #include "lista.h"
5  void insercao(TLista *lista, int start, int end);
6  void insSort(TLista *lista,int *pos,int npos);
7  #endif
```

4.2.2 source

```
1  #include "inssort.h"
2  void insSort(TLista *lista,int *pos,int npos){
3  int i, j, aux;
4  for (i = 1; i < npos; i++) {
5  aux = pos[i];
6  j = i - 1;
7  while(( j >= 0) && (lista->itens[aux].chave < lista->itens[↩
       pos[j]].chave)){
8  pos[j + 1] = pos[j];
9  j--;
10 }
11 pos[j+1] = aux;
12 }
13 }
14
15 void insercao(TLista *lista, int start, int end){
16 int i;
17 TItem aux;
18 for(; start < end; start++){
19 aux = lista->itens[start];
20 i = start - 1;
21 while ((i>=0) && (aux.chave < lista->itens[i].chave)) {
22 lista->itens[i+1] = lista->itens[i];
23 i--;
24 }
25 lista->itens[i+1] = aux;
26 }
27 }
```

4.2.3 Descrição

- **void insercao(TLista *lista, int start, int end)**

  Implementação do algoritmo de ordenação por inserção adaptado para ordenar um intervalo especifico de posições do vetor.

  1. *TLista *lista*

     Parâmetro que armazena endereço da lista a ser inicializada.

  2. *int start*

     Parâmetro que indica o inicio da lista.

3. *int end*

   Parâmetro que indica o término da lista.

- **void insSort(TLista *lista, int *pos, int npos)**

  Implementação do algoritmo de ordenação por inserção que se baseia nas chaves dos itens da lista para ordenar o vetor *pos*.

  1. *TLista *lista*

     Parâmetro que armazena endereço da lista a ser ordenada.

  2. *int *pos*

     Parâmetro que contém as posições a serem ordenadas pela função *insSort*.

  3. *int npos*

     Parâmetro que indica o tamanho do vetor *pos*.

4.3 lista

TAD utilizado em todas implementações de QuickSort's feitas nesta biblioteca. Responsável por armazenar um vetor de itens a serem ordenados e os iteradores utilizados para isso.

4.3.1 header

```
1  #ifndef LISTA_H_INCLUDED
2  #define LISTA_H_INCLUDED
3  #include <stdlib.h>
4  typedef struct{
5  int chave;
6  } TItem;
7  typedef struct{
8  int esq, dir, sz;
9  TItem *itens;
10 } TLista;
11 void initTLista(TLista *lista,TItem *itens,int n);
12 void destroyTLista(TLista *lista);
13 void reinitTLista(TLista *lista,TItem *itens,int n);
14 #endif
```

4.3.2 source

```
#include "lista.h"
void initTLista(TLista *lista,TItem *itens,int n){
int i;
lista->sz = n;
lista->itens = (TItem*)malloc(n*sizeof(TItem));
for (i = 0; i < n; i++) {
lista->itens[i] = itens[i];
}
lista->esq = 0;
lista->dir = n-1;
}
void reinitTLista(TLista *lista,TItem *itens,int n){
int i;
lista->sz = n;
lista->itens = (TItem*)realloc(lista->itens,n*sizeof(TItem));
for (i = 0; i < n; i++) {
lista->itens[i] = itens[i];
}
lista->esq = 0;
lista->dir = n-1;
}
void destroyTLista(TLista *lista){
free(lista->itens);
}
```

4.3.3 Descrição

- **TItem**

  1. *int chave*
     Responsável por armazenar a informação contida no item.

- **TLista**

  1. *int esq, dir*
     Iteradores responsáveis por delimitar partição a ser ordenada.
  2. *int sz*
     Variável que armazena tamanho atual no Arranjo de itens.

3. *TItem *itens*
   Arranjo que armazena os itens contidos na lista.

- **void initTLista(TLista *lista,TItem *itens,int n)**

  Função que inicializa uma lista.

  1. *TLista *lista*
     Parâmetro que armazena endereço da lista a ser inicializada.
  2. *TItem *itens*
     Parâmetro que armazena itens iniciais a serem inseridos à lista.
  3. *int n*
     Parâmetro que contém o tamanho do arranjo de itens.

- **void destroyTLista(TLista *lista)**

  Função responsável pela destruição da lista. Importante para evitar vazamento de memória (memory leak).

  1. *TLista *lista*
     Parâmetro que armazena endereço da lista a ser destruída.

- **void reinitTLista(TLista *lista,TItem *itens,int n)**

  Função responsável pela reinicialização da lista. Mais eficaz do que chamar destroyTLista e initTLista, além de ser mais prático.

  1. *TLista *lista*
     Parâmetro que armazena endereço da lista a ser destruída e reinicializada.
  2. *TItem *itens*
     Parâmetro que armazena itens iniciais a serem inseridos à lista.
  3. *int n*
     Parâmetro que contém o tamanho do arranjo de itens.

4.4 insquick

Implementação do quicksort que utiliza o algoritmo insertSort para tratar a ordenação de partições menores que um valor pré-definido (em tempo de compilação) M.

4.4.1 header

```
#ifndef INSQUICK_H_INCLUDED
#define INSQUICK_H_INCLUDED
#include "inssort.h"
#include "lista.h"
void ins_particao(TLista *lista, int *i, int *j); //↩
    iteradores i e j ãso criados dentro de ordena!!
void ins_ordena(TLista *lista);
#endif
```

4.4.2 source

```
#include "insquick.h"
#define M 10
void ins_particao(TLista *lista, int *i, int *j){
TItem pivo, aux;
*i = lista->esq;
*j = lista->dir;
pivo = lista->itens[(*i+*j)/2];
do {
while (pivo.chave > lista->itens[*i].chave) {
(*i)++;
}
while (pivo.chave < lista->itens[*j].chave) {
(*j)--;
}
if (*i <= *j) {
aux = lista->itens[*i];
lista->itens[*i] = lista->itens[*j];
lista->itens[*j] = aux;
(*i)++;
(*j)--;
}
} while (*i <= *j);
}
void ins_ordena(TLista *lista){
int i,j, sEsq, sDir;
if(lista->dir - lista->esq > M){
ins_particao(lista,&i,&j);
if(lista->esq < j){
```

```
29      sDir = lista->dir;
30      lista->dir = j;
31      ins_ordena(lista);
32      lista->dir = sDir;
33      }
34      if(i < lista->dir){
35      sEsq = lista->esq;
36      lista->esq = i;
37      ins_ordena(lista);
38      lista->esq = sEsq;
39      }
40      }
41      else{
42      insercao(lista,lista->dir,lista->esq);
43      }
44      }
```

4.4.3 Descrição

- **void ins_particao(TLista *lista, int *i, int *j)**

  Função responsável por dividir a lista em itens menores que a esquerda do *pivô* e maiores a direita do mesmo.

  1. *TLista *lista*

     Parâmetro que armazena endereço da lista a ser ordenada.

  2. *int *i*

     Ponteiro para endereço do iterador, que percorre cada elemento a partir do inicio da lista.

  3. *int *j*

     Ponteiro para endereço do iterador, que percorre cada elemento a partir do final da lista.

- **void ins_ordena(TLista *lista)**

  Função responsável pela conexão das partições formadas por *ins_particao*.

  1. *TLista *lista*

     Parâmetro que armazena o endereço da lista particionada e que será redirecionado para a função *ins_particao*, executando a parte recursiva do *insquick* de particionamento.

4.5 iteintquick

Implementação iterativa do quicksort que ordena as menores partições primeiro.

4.5.1 header

```
1  #ifndef ITEINTQUICK_H_INCLUDED
2  #define ITEINTQUICK_H_INCLUDED
3  #include "lista.h"
4  #include "recquick.h"
5  #include "pilha.h"
6  void it_int_ordena(TLista *lista);
7  #endif
```

4.5.2 source

```
1  #include "iteintquick.h"
2
3  void it_int_ordena(TLista *lista){
4  int i, j;
5  TPilha p;
6  iniPilha(&p, lista->dir);
7
8  particao(lista, &i,&j);
9  if((j-lista->esq) > (lista->dir-i)){//Verifica se a çãpartio↩
       da esquerda é a maior, ou seja é empilhada primeiro
10 empilha(&p, lista->esq);
11 empilha(&p, j);
12 empilha(&p, i);
13 empilha(&p, lista->dir);
14 }else{// se ãno a da direira é empilhada primeiro
15 empilha(&p, i);
16 empilha(&p, lista->dir);
17 empilha(&p, lista->esq);
18 empilha(&p, j);
19 }
20
21 do{ // repte o processo enquanto houverem elementos a serem ↩
     desempilhados
```

```
22      lista->dir = desempilha(&p);
23      lista->esq = desempilha(&p);
24      particao(lista,&i,&j);
25      if((j-lista->esq) > (lista->dir-i)){
26      if(i<lista->dir && j>lista->esq){
27      empilha(&p, lista->esq);
28      empilha(&p, j);
29      empilha(&p, i);
30      empilha(&p, lista->dir);
31      }else if(i<lista->dir){
32      empilha(&p, i);
33      empilha(&p, lista->dir);
34      }else if(j>lista->esq){
35      empilha(&p, lista->esq);
36      empilha(&p, j);
37      }
38      }else{
39      if(i<lista->dir && j>lista->esq){
40      empilha(&p, i);
41      empilha(&p, lista->dir);
42      empilha(&p, lista->esq);
43      empilha(&p, j);
44      }else if(i<lista->dir){
45      empilha(&p, i);
46      empilha(&p, lista->dir);
47      }else if(j>lista->esq){
48      empilha(&p, lista->esq);
49      empilha(&p, j);
50      }
51      }
52
53      }while(p.top > 0);
54      destroyTPilha(&p);// desaloca o vetor de indeces da pilha.
55
56
57      }
```

4.5.3 Descrição

- **void it_int_ordena(TLista *lista)**

  Função responsável pela conexão das partições formadas por *particao*.

  1. *TLista *lista*

Parâmetro que armazena endereço da lista a ser ordenada.

4.6 itequick

Implementação iterativa do quicksort.

4.6.1 header

```
1  #ifndef ITEQUICK_H_INCLUDED
2  #define ITEQUICK_H_INCLUDED
3
4  #include "lista.h"
5  #include "recquick.h"
6  #include "pilha.h"
7  void it_ordena(TLista *lista);
8  #endif
```

4.6.2 source

```
1  #include "itequick.h"
2  void it_ordena(TLista *lista){
3  int i, j;
4
5  TPilha p;
6  iniPilha(&p, lista->dir);
7
8  particao(lista, &i,&j);
9  empilha(&p, lista->esq); //empilha primeiro a çãpartio ↩
       esqueda
10 empilha(&p, j);
11 empilha(&p, i); // logo a da direita é processada primeiro ↩
       nesse caso.
12 empilha(&p, lista->dir);
13 do{
14 lista->dir = desempilha(&p); // desempilha a çãpartio ↩
       limitado por esqueda e direita
15 lista->esq = desempilha(&p);
16 particao(lista,&i,&j);
17 if(i<lista->dir && j>lista->esq){ // verifica se ãno chegou ↩
       ao final de nenhum lado da lista
```

```
18    empilha(&p, lista->esq); // logo empilha a çãpartio da ↩
          direita e esqueda
19    empilha(&p, j);
20    empilha(&p, i);
21    empilha(&p, lista->dir);
22    }else if(i<lista->dir){ // verifica se chegou ao final da ↩
          lista na esqueda
23    empilha(&p, i); // logo empilha apenas a çãpartio da direita↩
          áj que a esquerda chegou ao final
24    empilha(&p, lista->dir);
25    }else if(j>lista->esq){ // verifica se chegou ao final da ↩
          lista na direita
26    empilha(&p, lista->esq); // logo empilha apenas a çãpartio ↩
          da esqueda, áj que a direita chegou ao final
27    empilha(&p, j);
28    }// caso ácontrrio, direita e esqueda tiver chegado ao final↩
          , ãno fz nada
29    }while(p.top > 0); // enquanto houverem çõparties continua ↩
          particionando ou desempilhando.
30    destroyTPilha(&p);
31    }
```

4.6.3 Descrição

- **void it_ordena(TLista *lista)**

  Função responsável pela conexão das partições formadas por *particao*.

  1. *TLista *lista*

     Parâmetro que armazena endereço da lista a ser ordenada.

4.7 medquick

Implementação do quicksort que seleciona o pivô a partir da mediana feita entre K elementos aleatórios da lista.

4.7.1 header

```
1    #ifndef MEDQUICK_H_INCLUDED
2    #define MEDQUICK_H_INCLUDED
3    #include "lista.h"
```

```
4   #include "inssort.h"
5   void med_particao(TLista *lista, int *i, int *j, int k); //↩
        iteradores i e j ãso criados dentro de med_ordena!!
6   void med_ordena(TLista *lista);  //çãFuno ordena q chama ↩
        çãmed_partio
7
8
9   #endif
```

4.7.2 source

```
1   #include "medquick.h"
2   #define K 3
3   void med_particao(TLista *lista, int *i, int *j, int k){
4   int *ipivo, c;
5   TItem pivo, aux;
6   *i = lista->esq;
7   *j = lista->dir;
8   ipivo = (int*)malloc(k*sizeof(int));
9   for (c = 0; c < k; c++) {
10  ipivo[c] = rand()%(*j - *i +1) + *i;
11  }
12  //ordena ipivo pelas chaves em lista
13  insSort(lista,ipivo,k);
14  //Pega o do meio
15  pivo = lista->itens[ipivo[(k/2)+1]];
16  free(ipivo);
17  do {
18  while (pivo.chave > lista->itens[*i].chave) {
19  (*i)++;
20  }
21  while (pivo.chave < lista->itens[*j].chave) {
22  (*j)--;
23  }
24  if (*i <= *j) {
25  aux = lista->itens[*i];
26  lista->itens[*i] = lista->itens[*j];
27  lista->itens[*j] = aux;
28  (*i)++;
29  (*j)--;
30  }
31  } while (*i <= *j);
32
```

```
33    }
34    void med_ordena(TLista *lista){
35    int i,j, sEsq, sDir;
36    med_particao(lista,&i,&j,K);
37    if(lista->esq < j){
38    sDir = lista->dir;
39    lista->dir = j;
40    med_ordena(lista);
41    lista->dir = sDir;
42    }
43    if(i < lista->dir){
44    sEsq = lista->esq;
45    lista->esq = i;
46    med_ordena(lista);
47    lista->esq = sEsq;
48    }
49    }
50    \end{lstlistinag}
51    \subsubsection{çãDescrio}
52
53    \subsection{recintquick}
54    \subsubsection{header}
55    \begin{lstlisting}
```

4.7.3 Descrição

- **void med_particao(TLista *lista, int *i, int *j, int k)**

  Função responsável por calcular o pivô a partir da mediana de k itens e particionar o vetor em problemas menores.

  1. *TLista *lista*

     Parâmetro que armazena endereço da lista a ser ordenada.

  2. *int *i*

     Ponteiro para endereço do iterador, que percorre cada elemento a partir do início da lista.

  3. *int *j*

     Ponteiro para endereço do iterador, que percorre cada elemento a partir do final da lista.

  4. *int k*

Número de elementos aleatórios que serão selecionados para a realização do cálculo do pivô.

- **void med_ordena(TLista *lista)**

  Função responsável pela conexão das partições formadas pela função *med_particao*.

  1. *TLista *lista*

     Parâmetro que armazena o endereço da lista a ser ordenada.

4.8 recquick

Implementação recursiva do quicksort.

4.8.1 header

```
1  #ifndef RECQUICK_H_INCLUDED
2  #define RECQUICK_H_INCLUDED
3  #include "lista.h"
4  void particao(TLista *lista, int *i, int *j); //iteradores i↩
       e j ãso criados dentro de ordena!!
5  void ordena(TLista *lista);
6  #endif
```

4.8.2 source

```
1  #include "recquick.h"
2  void particao(TLista *lista, int *i, int *j){
3  TItem pivo, aux;
4  *i = lista->esq;
5  *j = lista->dir;
6  pivo = lista->itens[(*i+*j)/2];
7  do {
8  while (pivo.chave > lista->itens[*i].chave) {
9  (*i)++;
10 }
11 while (pivo.chave < lista->itens[*j].chave) {
12 (*j)--;
```

```
13      }
14      if (*i <= *j) {
15      aux = lista->itens[*i];
16      lista->itens[*i] = lista->itens[*j];
17      lista->itens[*j] = aux;
18      (*i)++;
19      (*j)--;
20      }
21      } while (*i <= *j);
22
23      }
24      void ordena(TLista *lista){
25      int i,j, sEsq, sDir;
26      particao(lista,&i,&j);
27      if(lista->esq < j){
28      sDir = lista->dir;
29      lista->dir = j;
30      ordena(lista);
31      lista->dir = sDir;
32      }
33      if(i < lista->dir){
34      sEsq = lista->esq;
35      lista->esq = i;
36      ordena(lista);
37      lista->esq = sEsq;
38      }
39      }
```

4.8.3 Descrição

- **void particao(TLista *lista, int *i, int *j)**

  Função responsável por dividir a lista em itens menores que a esquerda do *pivô* e maiores a direita do mesmo.

  1. *TLista *lista*

     Parâmetro que armazena endereço da lista a ser ordenada.

  2. *int *i*

     Ponteiro para endereço do iterador, que percorre cada elemento a partir do início da lista.

  3. *int *j*

Ponteiro para endereço do iterador, que percorre cada elemento a partir do final da lista.

- **void ordena(TLista *lista)**

  Função responsável pela conexão das partições formadas pela função *particao*.

  1. *TLista *lista*

     Parâmetro que armazena o endereço da lista a ser ordenada.

4.9 recintquick

Implementação recursiva do quicksort, que ordena as menores partições primeiro.

4.9.1 header

```
1  #ifndef RECINTQUICK_H_INCLUDED
2  #define RECINTQUICK_H_INCLUDED
3  #include "lista.h"
4  #include "recquick.h"
5  void rec_int_ordena(TLista *lista); //çãFuno ordena q chama ↩
       çãmed_partio
6  #endif
```

4.9.2 source

```
1  #include "recintquick.h"
2  void rec_int_ordena(TLista *lista){
3  int i,j, sEsq, sDir;
4  particao(lista,&i,&j);
5  if((j - lista->esq) < (lista->dir - i)){
6  if(lista->esq < j){
7  sDir = lista->dir;
8  lista->dir = j;
9  ordena(lista);
10 lista->dir = sDir;
11 }
```

```
12      if(i < lista->dir){
13      sEsq = lista->esq;
14      lista->esq = i;
15      ordena(lista);
16      lista->esq = sEsq;
17      }
18      }
19      else{
20      if(i < lista->dir){
21      sEsq = lista->esq;
22      lista->esq = i;
23      ordena(lista);
24      lista->esq = sEsq;
25      }
26      if(lista->esq < j){
27      sDir = lista->dir;
28      lista->dir = j;
29      ordena(lista);
30      lista->dir = sDir;
31      }
32      }
33      }
```

4.9.3 Descrição

- **void rec_int_ordena(TLista *lista)**

  Função responsável por implementar o algoritmo de quicksort de maneira recursiva, ordenando sempre as menores partições.

  1. *TLista *lista*

     Parâmetro que armazena endereço da lista a ser ordenada.

4.10 heapsort

Implementação iterativa tradicional do heapsort.

4.10.1 header

```
1 #ifndef HEAPSORT_H_INCLUDED
2 #define HEAPSORT_H_INCLUDED
```

```
3 #include "../quicksorts/lista.h"
4 void heapSort(TLista *lista);
5 void refaz(TLista *lista);
6 void constroi(TLista *lista);
7 //Realmente ánecessrios?
8 int retiraMax(TLista *lista, TItem *iMax);
9 int aumentaChave(int i, int nova_chave, TLista *lista);
10 void insere(TLista *lista,TItem item);
11
12 #endif
```

4.10.2 source

```
1 #include "heapsort.h"
2 void heapSort(TLista *lista){
3    TItem aux;
4    constroi(lista);
5    while( lista->dir > 1){
6       aux = lista->itens[1];
7       lista->itens[1] = lista->itens[lista->dir];
8       lista->itens[lista->dir--] = aux;
9       refaz(lista);
10   }
11 }
12 void refaz(TLista *lista){
13    int j, i = lista->esq;
14    TItem aux;
15    j = i * 2;
16    aux = lista->itens[i];
17    while (j <= lista->dir) {
18       if (j < lista->dir && lista->itens[j].chave < lista->↩
          itens[j+1].chave) {
19          j++;
20       }
21       if(aux.chave >= lista->itens[j].chave){
22          break;
23       }
24       lista->itens[i] = lista->itens[j];
25       i = j;
26       j = i * 2;
27    }
28    lista->itens[i] = aux;
29 }
```

```
30 void constroi(TLista *lista){
31    lista->esq = lista->sz/2;
32    while (lista->esq > 0) {
33       refaz(lista);
34       lista->esq--;
35    }
36 }
```

4.10.3 Descrição

- **void heapSort(TLista *lista)** Função que "junta"os elementos já ordenados ao fim do heap até o fim da ordenação.

    1. *TLista *lista* Parâmetro responsável por armazenar os itens do heap e as variáveis sz, esq e dir.

- **void refaz(TLista *lista)** Função que garante a ordenação ao conferir a validade das regras da formação de um heap.

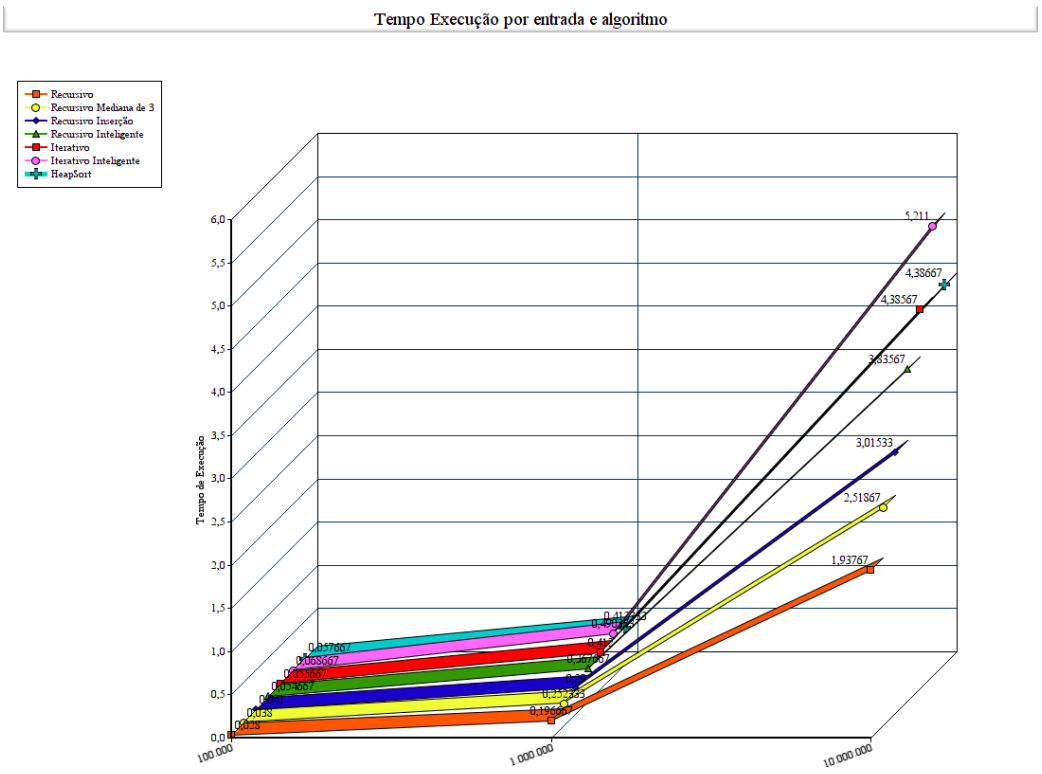
    1. *TLista *lista* Parâmetro responsável por armazenar os itens do heap e as variáveis sz, esq e dir.

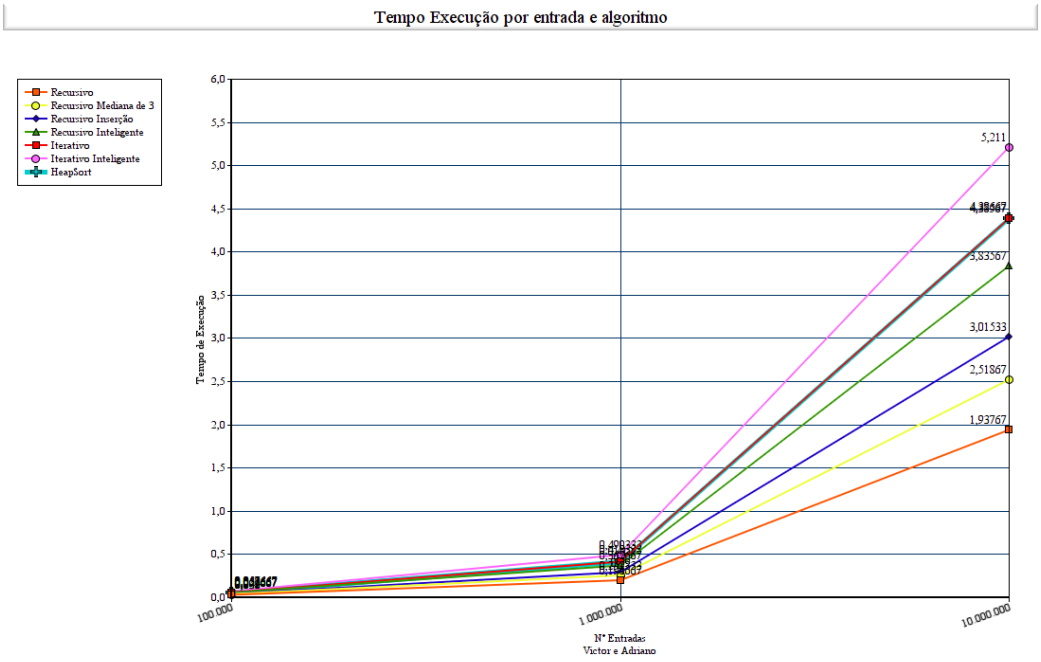
- **void constroi(TLista *lista)** Função responsável por construir o heap, chamando a função refaz para o centro da lista até a primeira posição.

    1. *TLista *lista* Parâmetro responsável por armazenar os itens do heap e as variáveis sz, esq e dir.

5 Conclusão

Após muitos testes em duas máquinas diferentes (i7-3537U 8gb ram e DualCore 2.0GHz 2gb ram), rodando em sistemas operacionais diferentes (ArchLinux e ElementaryOS) conseguimos fazer uma média aritmética entre os valores (tempo por tamanho da lista a ser ordenada) obtidos em ambos os sistemas(ArchLinux e ElementaryOS) de cada computador e representá-la em um gráfico construído e renderizado no site http://www.onlinecharttool.com/


Tempo Execução por entrada e algoritmo
Recursivo
Recursivo Mediana de 3
Recursivo Inserção
Recursivo Inteligente
Iterativo
Iterativo Inteligente
HeapSort
Tempo de Execução
6,0
5,5
5,0
4,5
4,0
3,5
3,0
2,5
2,0
1,5
1,0
0,5
0,0
100.000
1.000.000
10.000.000
5,211
4,38667
4,38567
3,01533
2,51867
1,93767



Como visto nos gráficos o HeapSort obteve um desempenho similar a implementação iterativa do QuickSort, o que é um ótimo resultado para situações que exigem um algoritmo de ordenação que não tenha pior caso $O(n^2)$.

6 Agradecimentos

A Universidade Federal de Viçosa Campus Florestal por nos proporcionar um ambiente de criatividade e inúmeras oportunidades;

A professora Thais Regina de Moura Braga Silva pela imensa preocupação e dedicação ao nosso aprendizado.