Algoritmos de Ordenação

Introdução

Algoritmos de ordenação são algoritmos que direcionam para a ordenação, ou reordenação, de valores apresentados em uma dada sequência, para que os dados possam ser acessados posteriormente de forma mais eficiente. Uma das principais finalidades desse tipo de algoritmo é a ordenação de vetores, uma vez que, em uma única variável pode-se ter inúmeras posições, a depender do tamanho do vetor declarado. Por exemplo, a organização de uma lista de presença escolar para que a relação fique organizada em ordem alfabética.

Critérios de Classificação

- Localização dos Dados
    - Ordenação Interna Todas os dados estão em memória principal (RAM)
    - Ordenação Externa: Memória principal não cabe todos os dados. Dados armazenados em memória secundária (disco)
- Estabilidade
    - Método é estável se a ordem relativa dos registros com a mesma chave não se altera após a ordenação.
- Adaptabilidade
    - Um método é adaptável quando a sequencia de operações realizadas depende da entrada
    - Um método que sempre realiza as mesmas operações, independente da entrada, é não adaptável.
- Uso da Memória
    - In place: ordena sem usar memória adicional ou usando uma quantidade constante de memória adicional. Alguns métodos precisam duplicar os dados.

- Movimentação dos Dados
  - Direta: estrutura toda é movida.

```
struct item a;
struct item b;
struct item aux = a;
a = b;
b = aux;
```

  - Indireta: apenas as chaves são acessadas e ponteiros para as estruturas são rearranjados

```
struct item *a;
struct item *b;
struct item *aux = *a;
a = b;
b = aux;
```

Critérios de Avaliação

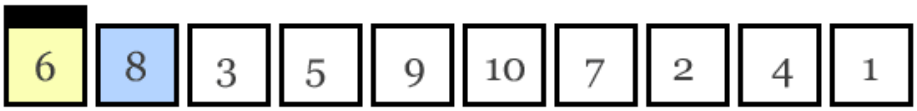
Seja *n* o número de registros em um vetor, considera-se duas medidas de complexidade:

- Número de comparações C(n) entre as chaves.
- Número de trocas ou movimentações M(n) de itens.

```
#define troca(A, B) {struct item c = A; A = B; B = c;}

void ordena(struct item *v, int n) {
  int i = 0, j = 0;
  for(i = 0; i < n-1; i++) {
    for(j = n-1; j > i; j--) {
      if(v[j-1].chave > v[j].chave) /* comparações */
        troca(v[j-1], v[j]); /* trocas */
    }
  }
}
```

Ordenação por Seleção (Selection sort)



Yellow is smallest number found
Blue is current item
Green is sorted list

Um dos mais simples e utilizados, o Selection Sort tem como principal finalidade passar o menor valor para a primeira posição, o segundo menor para a segunda posição, e assim sucessivamente, para os n valores nas n posições, onde o valor à esquerda é sempre menor que o valor à direita (valoresquerda < valordireita).

1. Procura o n-ésimo menor elemento do vetor.
2. Troca do n-ésimo menor elemento com o elemento na n-ésima posição.
3. Repete até ter colocado todos os elementos em suas posições
4. Elementos são movimentados apenas uma vez

```
#include <stdio.h>

void ordSelecao(int v[], int tam){
    int minimo = 0, i = 0, j = 0, menor = 0;

    for (i=0;i<tam-1;i++){
        minimo = i;
        //Encontra o menor elemento do vetor
        for (j=i+1;j<tam;j++){
            if (v[j]<v[minimo])
                minimo = j;
            }
        //Realiza a troca dos valores
        menor = v[minimo];
        v[minimo] = v[i];
        v[i] = menor;
    }
    //Note que o número de trocas é O(n)
    return;
}
```

Comparações: $C(n) = O(n^2)$

Movimentações: $M(n) = 3(n-1)$

Trocas: $T(n) = O(n)$

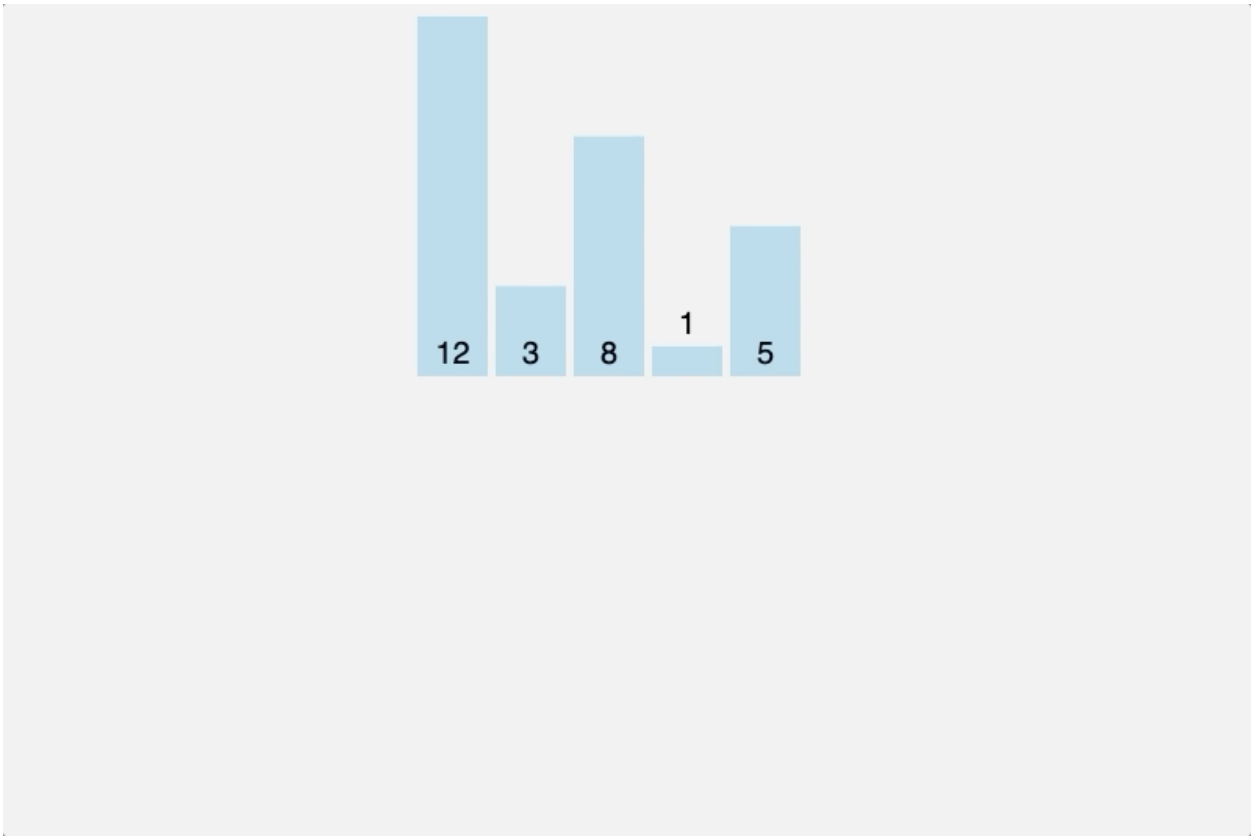
Vantagens

- Custo linear no tamanho da entrada para o número de movimentos de registros – a ser utilizado quando há registros muito grandes.
- **Preferido numa situação em que a operação de troca seja muito cara.**

Desvantagens

- Não adaptável. Não importa se o arquivo está parcialmente ordenado
- Algoritmo não é estável.

Ordenação por inserção (Inserction Sort)



O Insertion Sort é de fácil implementação, similar ao Bubble Sort (SZWARCFITER e MARKENZON, 2015). Seu funcionamento se dá por comparação e inserção direta. A medida que o algoritmo varre a lista de elementos, o mesmo os organiza, um a um, em sua posição mais correta, onde o elemento a ser alocado (k) terá, a sua esquerda um valor menor (k-1), e, de maneira similar, à sua direita um valor maior (k+1).

1. Mantemos os elementos entre zero e i-1 ordenados.
2. Achamos a posição do i-ésimo elemento e inserimos ele entre os i-1 que já estavam ordenados.
3. O programa repete esse passo até ordenar todos os elementos.

```
#include <stdio.h>

void ordInsercao(int *v, int n) {
    int i = 0, j = 0;
    int aux;
    for(i = 1; i < n; i++) {
        //Seleciona um valor na posição i
        aux = v[i];
        //j é um contador
        j = i - 1;
        //Enquanto j <=0 e aux for menor que v[j] movemos os valores
        while((j >= 0) && (aux < v[j])) {
            v[j+1] = v[j];
            j--;
        }
        //Adicionamos aux na posição i
        v[j + 1] = aux;
    }
    //Note que ele verifica se o vetor está organizado na primeira passagem
    return;
}
```

Comparações: $C(n) = O(n^2)$

Trocas: $T(n) = O(n^2)$

Adaptativo: $O(n)$ em tempo quando o conjunto de dados está parcialmente ordenado.

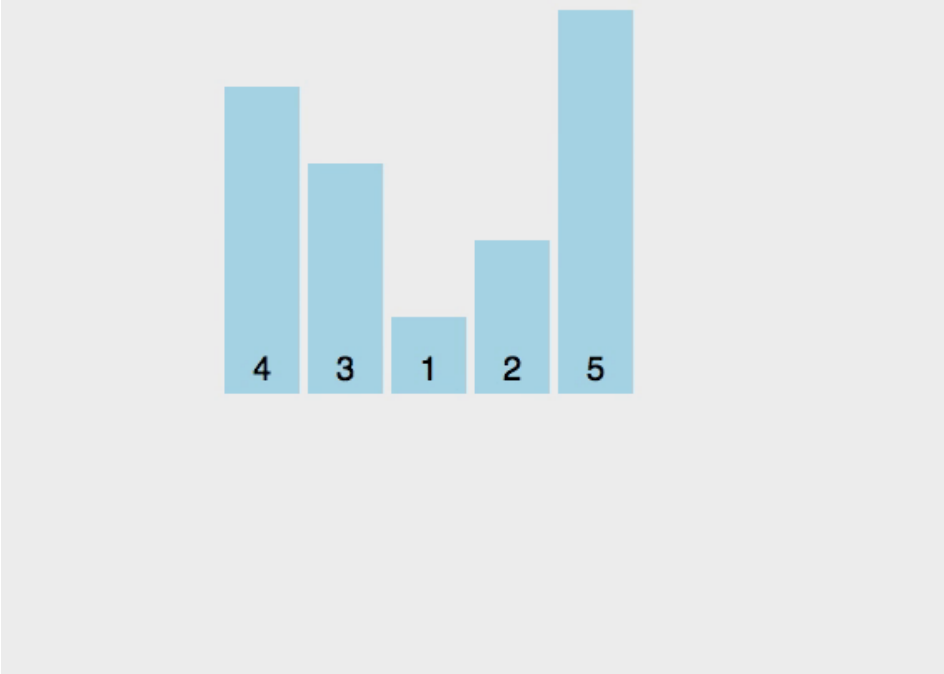
Vantagens

- Laço interno é eficiente, inserção é adequado para ordenar vetores pequenos.
- **É o método a ser utilizado quando o arquivo está “quase” ordenado.**
- É um bom método quando se deseja adicionar poucos itens a um arquivo ordenado, pois o custo é linear.
- O algoritmo de ordenação por inserção é estável.

Desvantagens

- Número de comparações tem crescimento quadrático.
- Alto custo de movimentação de elementos no vetor.

Ordenação pelo método da Bolha (Bubble Sort)



Este algoritmo é um do mais simples (SZWARCFITER e MARKENZON, 2015) e seu funcionamento ocorre por meio da comparação entre dois elementos e sua permuta, de modo que o elemento de maior valor fique à direita do outro (PEREIRA, 2010). “Após a primeira passagem completa pelo vetor (...) podemos garantir que o maior item terá sido deslocado para a última posição do vetor” [Pereira 2010, p.95]. Essa movimentação repete-se até que o vetor fique totalmente ordenado.

1. Compara dois elementos adjacentes e troca de posição se estiverem fora de ordem.
2. Quando o maior elemento do vetor for encontrado, ele será trocado até ocupar a última posição.
3. Na segunda passada, o segundo maior será movido para a penúltima posição do vetor, e assim sucessivamente.

```
#include<stdio.h>
#include<stdbool.h>

void ordBolha (int v[], int n) {
    int i = 0, j = 0;
    int temp = 0;
    bool troca = true;

    //Percorremos o vetor enquanto i >= 1 e troca for verdadeiro
    for(i = n-1; (i >= 1) && troca; i--){
        troca = false;
        for(j = 0; j < i ;j++){
            //Se o elemento seguinte a i é menor do que ele, fazemos a troca
            if(v[j] > v[j+1]){
                temp = v[j];
                v[j] = v[j+1];
                v[j+1] = temp;
                //Houve uma troca
                troca = true;
```

```
            }
        }
    }
    //Note que o algoritmo é estável
    return;
}
```

Comparações: $C(n) = O(n^2)$

Trocas: $T(n) = O(n^2)$

Adaptativo: $O(n)$ em tempo quando o conjunto de dados está parcialmente ordenado.

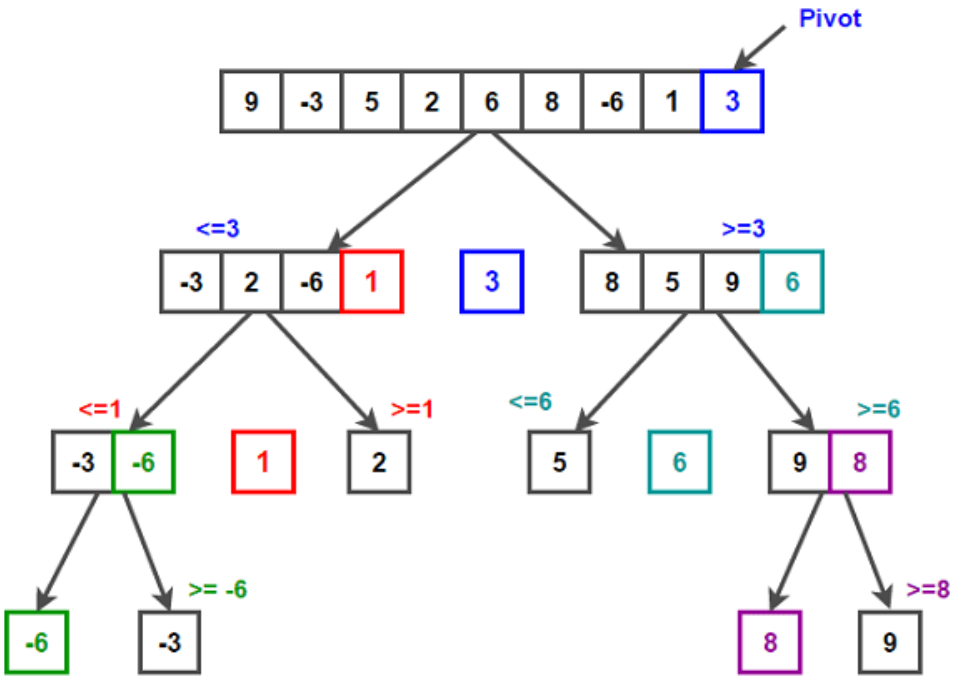
Vantagens

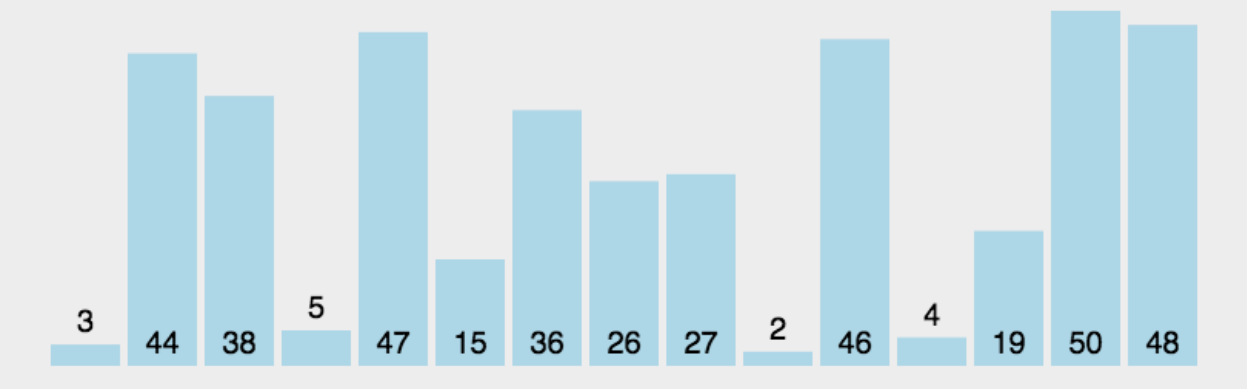
- Algoritmo simples.
- Algoritmo estável.

Desvantagens

- Não adaptável.
- Muitas trocas de itens.

Quick Sort





- É o algoritmo de ordenação interna mais rápido que se conhece para uma ampla variedade de situações.
- A ordenação baseia-se em **dividir para conquistar**. Um conjunto com n itens deve dividir-se em dois conjuntos menores que devem ser ordenados de forma independente.
- A ordenação acontece no próprio vetor e o método não garante estabilidade de ordem para valores iguais.
- **Estratégia para ordenação**
  - Definição de um pivô.
    - Primeiro elemento do vetor
    - Posicionar o pivô ao final dos menores.
  - Colocam-se os elementos menores que o pivô à esquerda.
  - Os elementos maiores que o pivô são acomodados à direita.
  - O pivô é colocado na sua posição correta (ordenada).
  - Os lados esquerdo e o direito são em seguida ordenados independentemente.

```
#include<stdio.h>

//Protótipo das funções
void quicksort(int v[], int esq, int dir);
int particao(int v[],int esq,int dir);
void troca(int v[], int i, int j);

void troca(int v[], int i,int j){
  int temp;
  temp=v[i];
  v[i]=v[j];
  v[j]=temp;
}

/*  Parte o vetor em duas metadas, de preferência equilibradas e
*   volta a posição onde o pivô se encontra.*/
int particao(int v[],int esq,int dir){
    int i = 0, fim = 0;
```

```
    fim = esq;
    for(i=esq+1;i<=dir;i++){
        if(v[i]<v[esq]) {
            fim=fim+1;
            troca(v,fim,i);
        }
    }
    troca(v,esq,fim);
    return fim;
}

void quicksort(int v[], int esq, int dir){
    int i = 0;

    //Já está ordenado
    if(esq>=dir) return;

    //Devolve a posição onde cai o pivô
    i = particao(v,esq,dir);

    //Lista dos menores
    quicksort(v,esq,i-1);

    //Lista dos maiores
    quicksort(v,i+1,dir);
}
```

A função partição executa em tempo $O(n)$

Espaço: $E(n) = O(n^2)$, mas se modificado pode ser $O(n * log(n))$

Tempo: $T(n) = O(n^2)$, mas tipicamente $O(n * log(n))$

Médio Caso

- O tempo médio (esperado) do quicksort é $O(nlogn)$ quando o pivô particiona o vetor de forma equilibrada.

Piores Casos

- Ocorrem quando a sequência está ordenada ou em ordem inversa, pois a chave particionadora será o menor elemento (ou o maior).
- Uma entrada de dados ordenada representa o pior caso para o algoritmo quicksort quando a escolha do pivô se dá pelo primeiro elemento.O tempo de execução do quicksort é quadrático neste caso.

Vantagens

- Eficiente para ordenar arquivos.
- Requer $O(nlogn)$ comparações em média (caso médio) para ordenar n itens.

Desvantagens

- Tem um pior caso com $O(n^2)$ comparações.
- O método não é estável.

Modificações no QuickSort

```
/*Funções do quickbásico*/

void troca(int v[], int i,int j){
  int temp;
  temp=v[i];
  v[i]=v[j];
  v[j]=temp;
}

int particao(int v[],int esq,int dir){
  int i, fim;

  fim=esq;
  for(i=esq+1;i<=dir;i++)
    if(v[i]<v[esq]) {
       fim=fim+1;
       troca(v,fim,i);
    }
  troca(v,esq,fim);
  return fim;
}

void quicksort(int v[], int esq, int dir){
  int i;
  if(esq>=dir) return;
  i=particao(v,esq,dir);
  quicksort(v,esq,i-1);
  quicksort(v,i+1,dir);
}
```

```
/*quickbásico.h*/

void quicksort(int v[], int esq, int dir);
int particao(int v[],int esq,int dir);
void troca(int v[], int i,int j);
```

O pior caso pode ser evitado através da realização de pequenas modificações no algoritmo. Algumas opções são:

- **Escolha aleatória do pivô**
  - O tempo de execução depende dos pivôs sorteados.
  - O tempo médio é $O(nlogn)$ – podendo ser rápido ou lento, mas não depende de como os elementos estão organizados no vetor.

```
#include <time.h>
#include <stdlib.h>
#include "quickbasico.h"


int pivo_aleatorio(int esq, int dir) {
   double r;
   r = (double) rand()/RAND_MAX;  // valor entre 0.01 e 0.99
```

```
   return (int)(esq + r*(dir-esq));
}
//A função rand() gera um número pseudo-aleatório entre 0 e a constante RAND_MAX. A constante RAND_MAX é 32762.

void quicksort_aleatorizado(int *v, int esq, int dir) {
   int i;
   if(dir <= esq) return;
   troca(v, pivo_aleatorio(esq,dir), esq);
   i = particao(v, esq, dir);
   quicksort_aleatorizado(v, esq, i-1);
   quicksort_aleatorizado(v, i+1, dir);
}
```

- **Mediana de três**
  - Em lugar de fixar como pivô o elemento da esquerda, pode-se escolher o elemento médio de uma amostra de três elementos. Por exemplo: o da esquerda, o do meio, e o da direita.
  - Implementação:
    1. Recuperar os elementos primeiro, do meio e último
    2. Trocar o elemento do meio com o segundo elemento
    3. Escolher como pivô a mediana entre os elementos: primeiro, segundo e último
    4. O pivô deve ser colocado na segunda posição, o menor na primeira o e maior deles ao final.
    5. Esta estratégia permite entrar na partição sem considerar os elementos maior e menor que o pivô.
  - Esta estratégia diminui a probabilidade de que o pivô seja o maior ou o menor elemento da sequência.

```
#include "quickbasico.h"

void quicksort_mediana_tres(int v[], int esq, int dir) {
   int i;
   if(dir <= esq) return;
   troca(v, (esq+dir)/2, esq+1);
   if(v[esq] > v[esq+1])
      troca(v, esq, esq+1);
   if(v[esq] > v[dir])
      troca(v, esq, dir);
   if(v[esq+1] > v[dir])
      troca(v, esq+1, dir);
   i = particao(v, esq+1, dir-1);
   quicksort_mediana_tres(v, esq, i-1);
   quicksort_mediana_tres(v, i+1, dir);
 }
```

- Antes de iniciar a ordenação, aplicar um algoritmo de embaralhamento, como o de Ficher-Yates $(O(n))$

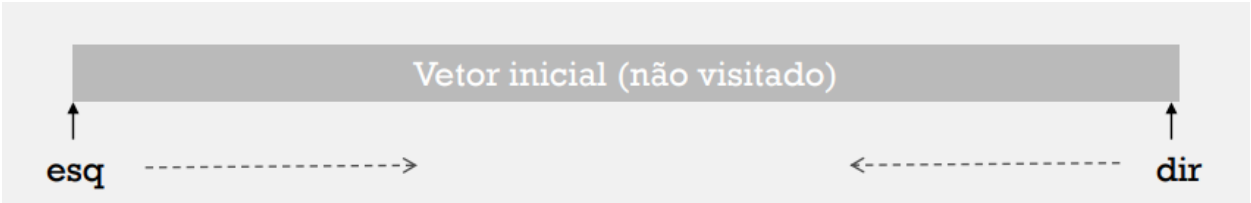
Pivô: Elemento Fundamental

- A escolha de um pivô adequado é uma atividade crítica para o bom funcionamento do quicksort. Se pudermos garantir que o pivô está próximo à mediana dos valores do vetor, então o quicksort é muito eficiente.
- Uma técnica que pode ser utilizada para aumentar a chance de encontrar bons pivôs é escolher aleatoriamente três elementos do vetor e usar a mediana desses três valores como pivô para a partição
- Em sequências com muitos elementos repetidos, ainda é grande a chance de não encontrarmos bons pivôs aleatoriamente ou com a ajuda da mediana de três.

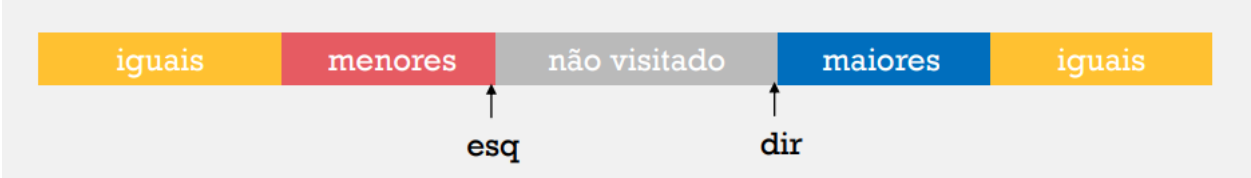
Sequências com muitas repetições

- Jon Bentley e Douglas McIlroy co-produziram (em 1993) uma versão otimizada do quicksort para tratar entradas com muitos elementos repetidos utilizando um **particionamento em três vias**:

- O objetivo é que em uma varredura, tenhamos um valor de particionamento (pivô) e que tanto o pivô quanto os elementos iguais a ele sejam corretamente posicionados (ao meio) enquanto os menores ficam à esquerda e os maiores à direita.
- A construção do particionamento se inicia com dois ponteiros: um em cada extremidade, e o caminhamento é feito em direção ao meio buscando separar os menores, maiores e iguais ao elemento pivô.



- Nos dois extremos estão as regiões que mantêm os elementos iguais ao pivô.
- A região ainda não visitada fica no meio e vai diminuindo a cada iteração.
- À esquerda dos não visitados estão os menores que o pivô.
- Ao lado direito dos não visitados está a região que mantém os elementos maiores que o pivô.



- Ao final da primeira visita completa, tem-se:

- Em seguida, todos os elementos iguais, acomodados nas extremidades, são movidos para o centro (apontado por esq/dir):

- Ao encerrar a visita e organizar as três vias (ou partições), a mesma abordagem será utilizada para particionar as regiões dos menores e maiores.
- **Implementação**

```
#include "quickbasico.h"

// Quicksort com três vias (partições)
// Exemplo:
// 3 5 12 6 11 7 14 9    (9 pivô)
//      |      |
// 3 5  7 6 11 12 14 9   (1a troca)
// encerra for
// 3 5 7 6 9 12 14 11    (colocação do pivô na posição certa)
//         |
// continuação esquerda e direita


void quicksort_tres_partes(int a[], int esq, int dir){
   int k, i = esq-1, j = dir, p = esq-1, q = dir;
   int v = a[dir];
   if (dir <= esq) return;
   for (;;)
   {
      while (a[++i] < v) ;
      while (v < a[--j])
           if (j == esq) break;
      if (i >= j) break;
      troca(a,i,j);
      if (a[i] == v) { p++; troca(a,p,i); }
      if (v == a[j]) { q--; troca(a,j,q); }
   }
  troca(a,i,dir);
  j = i-1;
  i = i+1;
  for (k = esq; k <= p; k++, j--) troca(a,k,j);
  for (k = dir-1; k >= q; k--, i++) troca(a,i,k);
  quicksort_tres_partes(a, esq, j);
  quicksort_tres_partes(a, i, dir);
}
```

  - Definir pivô: o pivô será o elemento mais à direita.
  - Apontar para a esquerda e caminhar até achar um elemento que não seja menor que o pivô

- Apontar para a direita-1 (não considerar o pivô) e caminhar até achar um elemento que não seja maior que o pivô.
- Parar se os ponteiros se cruzarem.
- Trocar os elementos nas posições de parada.
- Se os elementos trocados forem iguais ao pivô, enviá-los para as extremidades correspondentes.

Mergesort

Base do Método

1. Divida o vetor em duas metades.
2. Recursivamente ordene cada metade.

```
#include <stdio.h>
#include <stdlib.h>

//Protótipo de funções
void intercala(int v[], int e, int m, int d);
void mergesort (int v[], int e, int d);

void intercala(int v[], int e, int m, int d){
    int *r;  //vetor auxiliar dinâmico para armazenar a mistura
    int i = 0,j = 0,k = 0;

    r = (int *) malloc (((d+1)-e)*sizeof(int));

    i=e;
    j=m+1;
```

```
    k=0;

    /*  intercala enquanto nenhuma das partes do vetor
        foi consumida totalmente
    */
    while ((i<=m)&&(j<=d)){
    if(v[i]<=v[j]){
        r[k]=v[i];
        i++;
    }
    else{
        r[k]=v[j];
        j++;
    }
    k++;
    }

    while (i<=m) {r[k]=v[i]; i++; k++;} //termina de intercalar se sobrou à esquerda
    while (j<=d) {r[k]=v[j]; j++; k++;} //termina de intercalar se sobrou à direita

    //Copia do vetor auxiliar (contendo os dados ordenados) sobre o vetor original
    j=0;
    for(i=e;i<=d;i++){
        v[i]=r[j];
        j++;
    }
    free(r);

    return;
}

void mergesort (int v[], int e, int d){
    int m = 0;
    if (e<d){
        m = (e+d)/2;
        mergesort(v,e,m);
        mergesort(v,m+1,d);
        intercala(v,e,m,d);
    }//Note que o algoritmo é estavel
    return;
}
```

$O(n)$ de espaço extra para vetores

$O(lg(n))$ de espaço extra para listas encadeadas

$O(n \cdot lg(n))$ tempo

Vantagens

- Estável
- Não requer acesso randômico a dados.

Desvantagens

- Não adaptativo

ShellSort

- Proposto por Shell em 1959.
- É uma extensão do InsertSort.
- Problema com o algoritmo de ordenação por inserção:
- Troca itens adjacentes para determinar o ponto de inserção.
- São efetuadas n - 1 comparações e movimentações quando o menor item está na posição mais à direita no vetor.
- O método de Shell contorna este problema permitindo trocas de registros distantes um do outro.
- Os itens separados de h posições são rearranjados.
- Todo h-ésimo item leva a uma seqüência ordenada.
- Tal seqüência é dita estar h-ordenada.

```
#include <stdio.h>

void shellsort(int v[], int n) {
 int i, j, k, h = 1;
 int aux;

 do { h = h * 3 + 1;
    } while (h < n);

 do {
   h /= 3;
   for(i = h ; i < n ; i++) {
        aux = v[i];
        j = i;
        while (v[j - h] > aux) {
            v[j] = v[j - h];
            j -= h;
            if (j < h) break;
        }
        v[j] = aux;
        }
 } while (h != 1);
}
```

- A razão da eficiência do algoritmo ainda não é conhecida

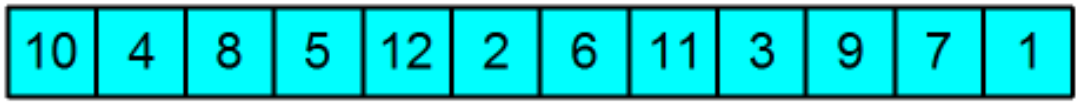
Vantagens

- Shellsort é uma ótima opção para arquivos de tamanho moderado.
- Sua implementação é simples e requer uma quantidade de código pequena.

Desvantagens

- O tempo de execução do algoritmo é sensível à ordem inicial do arquivo.
- O método não é estável

HeapSort



- A estrutura de dados **heap** (binário) é um array que pode ser visto como uma árvore binária praticamente completa.
    - Cada nó da árvore corresponde ao elemento do array que armazena o valor do nó.

    - A árvore está preenchida em todos os níveis, exceto talvez no nível mais baixo, que é preenchido a partir da esquerda.
- Um array A que representa um heap tem dois atributos
    - A.comprimento que é o número de elementos do array
    - A.tamanho-do-heap que é o numero de elementos no heap armazenado em A ($A.tamanho-do-heap \leq A.comprimento$)
- A raiz da árvore é $A[1]$
- Dado o índice i de um nó, os índice de seu pai, do filho a esquerda e do filho a direita podem ser calculados da forma:
    - $parent(i) = [i/2]$
    - $left(i) = 2i$
    - $right(i) = 2i + 1$
- Heap Máximo
    - Em um heap máximo, a propriedade de heap máximo é que, para todo nó i diferente da raiz $A[parent(i)] \geq A[i]$
- Heap Mínimo
    - Em um heap mínimo, a propriedade de heap mínimo é que, para todo nó i diferente da raiz $A[parent(i)] \leq A[i]$
- Visualizando o heap como uma árvore, definimos
    - a altura de um nó como o número de arestas no caminho descendente simples mais longo deste o o nó até uma folha
    - a altura do heap como a altura de sua raiz
    - a altura de um heap é $\Theta(lgn)$
- Algumas operacões sobre heap
    - **max-heapify**, executado no tempo $O(lgn)$, é a chave para manter a propriedade de heap máximo
    - **build-max-heap**, executado em tempo linear, produz um heap a partir de um array de entrada não ordenado
    - heapsort, executado no tempo $O(nlgn)$, ordena um array localmente

```
#include<stdio.h>

//Protótipo de funções
void troca(int *A, int p, int m);
int esq(int pos);
int dir(int pos);
void heap_maximo_pos(int *A, int pos, int tam_heap);
```

```
void construir_heap_max(int *A,  int fim);
void heapsort(int *A, int fim);

void troca(int *A, int p, int m){
    int temp = 0;
    temp = A[p];
    A[p] = A[m];
    A[m] = temp;
}

int esq(int pos){
   return (pos*2)+1;
}

int dir(int pos){
   return (pos*2)+2;
}

/* FUNÇÃO max-heapify
* A função max-heapify deixa que o valor A[i] “flutue para
* baixo”, de maneira que a subárvore com raiz no índice i se
* torne um heap
*/
void heap_maximo_pos(int *A, int pos, int tam_heap){
    int e = 0,d = 0,maior = 0;
    e = esq(pos);
    d = dir(pos);

    if((e <= tam_heap) && (A[e] > A[pos]))
        maior = e;
    else maior = pos;
    if((d <= tam_heap) && (A[d] > A[maior]))
        maior = d;
    if(maior != pos){
        troca(A,pos,maior);
        heap_maximo_pos(A,maior,tam_heap);
    }
    return;
}

/*FUNÇÃO build-max-heap
*O procedimento build-max-heap percorre os nós restantes
*da árvore e executa max-hepify sobre cada um
*/
void construir_heap_max(int *A,  int fim){
    int i;
    for(i=(fim-1)/2; i>=0; i--){
        heap_maximo_pos(A,i,fim);
    }
    return;
}

/* FUNÇÃO heapsort
* 1) Construir um heap, usando a função build-max-heap
* 2) Trocar o elemento A[1] com A[n], e atualiza o tamanho do heap para n − 1
* 3) Corrigir o heap com a função max-heapify e repetir o processo
*/
void heapsort(int *A, int fim){
    int i = 0,j = 0,tam_heap = 0;
    tam_heap = fim;

    construir_heap_max(A,fim);

    for (i = fim; i>=0; i--){
```

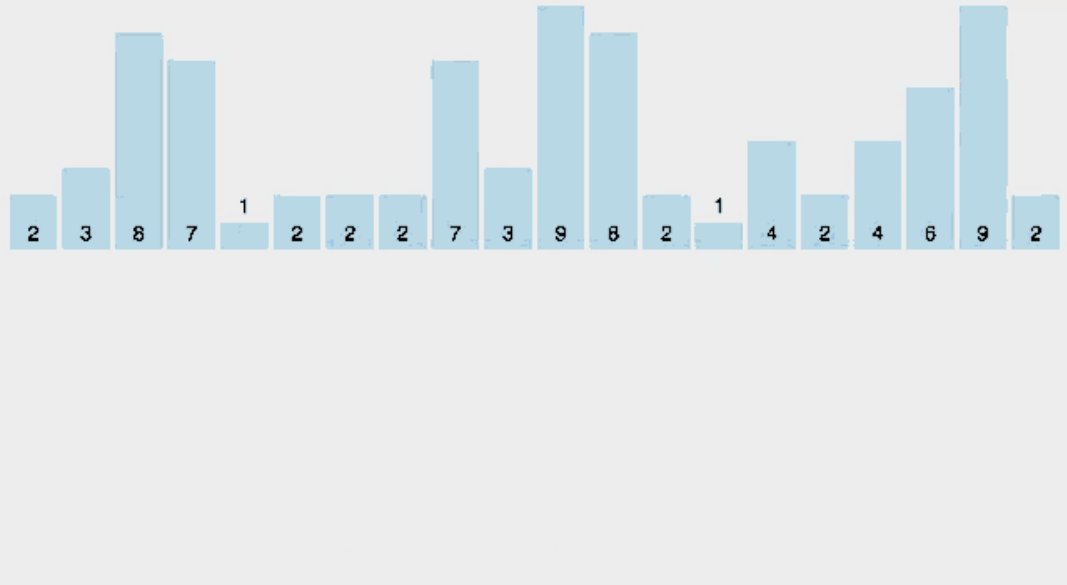
```
        troca(A,0,i);
        tam_heap--;
        heap_maximo_pos(A,0,tam_heap);
    }
    return;
}
```

A chamada a build-max-heap demora $O(n)$

O procedimento max-heapify demora $O(lgn)$ e é chamado $n-1$

Portanto, o tempo de execução do heapsort é $O(nlogn)$.

Counting Sort



- Também conhecido como ordenação por contagem
  - Algoritmo de ordenação para valores inteiros
  - Esse valores devem estar dentro de um determinado intervalo
  - A cada passo ele conta o número de ocorrências de um determinado valor no array
- Funcionamento
  - Usa um array auxiliar de tamanho igual ao maior valor a ser ordenado, K
  - O array auxiliar é usado para contar quantas vezes cada valor ocorre
  - Valor a ser ordenado é tratado como índice
  - Percorre o array auxiliar verificando quais valores existem e os coloca no array ordenado

```
#include<stdio.h>
```

```
#define K 100

void countingSort(int *V, int N){
  int i = 0, j = 0, k = 0;
  int baldes[K];

    for(i = 0; i< K; i++)
    baldes[i] = 0;

    for(i = 0; i < N; i++)
    baldes[V[i]]++;

    for(i = 0, j = 0; j<K; j++){
    for(k = baldes[j]; k>0; k--){
      V[i++] = j;
        }
  }
    return;
}
```

Complexidade linear

Considerando um array com N elementos e o maior valor sendo K, o tempo de execução é sempre de ordem $O(N + K)$.

K é o tamanho do array auxiliar

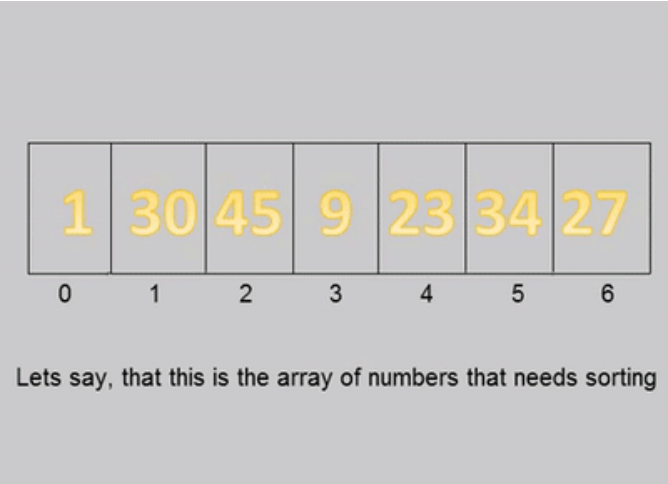
Vantagens

- Estável: não altera a ordem dos dados iguais
- Processamento simples

Desvantagens

- Não recomendado para grandes conjuntos de dados (K muito grande)
- Ordena valores inteiros positivos (pode ser modificado para outros valores)

Bucket Sort



- Também conhecido como ordenação usando baldes
  - Algoritmo de ordenação para valores inteiros
  - Usa um conjunto de K baldes para separar os dados
  - A ordenação dos valores é feita por balde
- Funcionamento
  - Distribui os valores a serem ordenados em um conjunto de baldes
    - Cada balde é um array auxiliar
    - Cada balde guarda uma faixa de valores
  - Ordena os valores de cada balde.
    - Isso é feito usando outro algoritmo de ordenação ou ele mesmo
  - Percorre os baldes e coloca os valores de cada balde de volta no array ordenado

```
#include<stdio.h>

#define TAM 5

typedef struct{
  int qtd;
  int valores[TAM];
}BALDE;

void bucketSort(int *V, int N){
  int i = 0,j = 0;
    int maior = 0,menor = 0,nroBaldes = 0,pos = 0;
  BALDE *bd;

  //Acha maior e menor valor
  maior = menor = V[0];
  for(i = 1; i < N; i++){
    if(V[i] > maior) maior = V[i];
    if(V[i] < menor) menor = V[i];
  }

    //Inicializa baldes
  nroBaldes = (maior - menor) / TAM + 1;
  bd = (BALDE *)malloc(nroBaldes * sizeof(BALDE));

    for(i = 0; i < nroBaldes; i++)
    bd[i].qtd = 0;

  //Distribui os valores nos baldes
  for(i = 0; i < N; i++){
    pos = (V[i] - menor)/TAM;
    bd[pos].valores[bd[pos].qtd] = V[i];
    bd[pos].qtd++;
  }

  //Ordena baldes e coloca no array
  pos = 0;
  for(i = 0; i < nroBaldes; i++){
    insertionSort(bd[i].valores, bd[i].qtd);
    for(j = 0; j < bd[i].qtd; j++){
```

```
      V[pos] =  bd[i].valores[j];
      pos++;
    }
  }
    free(bd);

    return;
}
```

Considerando um array com N elementos e K baldes, o tempo de execução é

- $O(N + K)$, melhor caso: dados estão uniformemente distribuídos
- $O(N^2)$, pior caso: todos os elementos são colocados no mesmo balde

Vantagens

- Estável: não altera a ordem dos dados iguais
  - Exceto se usar um algoritmo não estável nos baldes
- Processamento simples
- Parecido com o Counting Sort
  - Mas com baldes mais sofisticados

Desvantagens

- Dados devem estar uniformemente distribuídos
- Não recomendado para grandes conjuntos de dados
- Ordena valores inteiros positivos (pode ser modificado para outros valores)

Comparação de Algortitmos

O algoritmo **Bubble Sort**, apesar de ser o de mais fácil implementação, não apresenta resultados satisfatórios, principalmente no número de comparações. A ineficiência desse algoritmo pode ser traduzida como um grande consumo de processamento, o que, para máquinas com poucos (ou limitados) recursos computacionais, resulta em lentidão e longos períodos de espera. Sua aplicação é, na opinião dos autores, indicada somente para fins educacionais, visto que um projeto com o mesmo pode ser considerado ineficiente e/ou fraco (SILVA, 2010).

O **Insertion Sort**, por sua vez, é útil para estruturas lineares pequenas, geralmente entre 8 e 20 elementos, sendo amplamente utilizados em sequências de 10 elementos, tendo ainda, listas ordenadas de forma decrescente como pior caso, listas em ordem crescente como o melhor caso e, as demais ordens como sendo casos medianos. Sua principal vantagem é o pequeno número de comparações, e, o excessivo número de trocas, sua desvantagem. Como exemplo de uso, tem-se a ordenação de cartas de um baralho.

O **Selection Sort** torna-se útil em estruturas lineares similares ao do Insertion Sort, porém, com o número de elementos consideravelmente maior, já que, o número de trocas é

muito inferior ao número de comparações, consumindo, assim, mais tempo de leitura e menos de escrita. A vantagem de seu uso ocorre quando se trabalha com componentes em que, quanto mais se escreve, ou reescreve, mais se desgasta, e, consequentemente, perdem sua eficiência, como é o caso das memórias EEPROM e FLASH.

Ambos os algoritmos (Insertion e Selection), apesar de suas diferentes características, são mais comumente utilizados em associação com outros algoritmos de ordenação, como os Merge Sort, Quick Sort e o Shell Sort, que tendem a subdividir as listas a serem organizadas em listas menores, fazendo com que sejam mais eficientemente utilizados.

O **Merge Sort** apresenta-se, em linhas gerais, como um algoritmo de ordenação mediano. Devido a recursividade ser sua principal ferramenta, seu melhor resultado vem ao lidar com estruturas lineares aleatórias. Entretanto, ao lidar com estrutura pequenas e/ou já pré-ordenada (crescente ou decrescente), a recursividade passa a ser uma desvantagem, consumindo tempo de processamento e realizando trocas desnecessárias. Esse algoritmo é indicado para quando se lida com estruturas lineares em que a divisão em estruturas menores sejam o objetivo, como, por exemplo, em filas para operações bancárias.

O algoritmo **Quick Sort**, ao subdividir o vetor e fazer inserções diretas utilizando um valor de referência (pivô), reduz seu tempo de execução, mas, as quantidades de comparações (leitura) e, principalmente, trocas (escrita) ainda são muito altas. Apesar disso, o Quick Sort se apresenta uma boa opção para situações em que o objetivo é a execução em um menor tempo, mesmo que para isso haja um detrimento em recursos computacionais de processamento.

O **Shell Sort**, baseado nos dados deste trabalho, é o que apresenta os resultados mais satisfatórios, principalmente com estruturas maiores e desorganizadas. Por ser considerado uma melhoria do Selection Sort, o Shell Sort, ao ser utilizado com as mesmas finalidades que seu predecessor – recursos que demandem pouca escrita – irá apresentar um melhor desempenho, e, consequentemente, expandir a vida útil dos recursos.

- Complexidade de espaço: merge > quick > heap

Ordenação de array struct

- A ordenação é baseada em uma chave
  - A chave de ordenação é o campo do item utilizado para comparação
    - Valor armazenado em um array de inteiros
    - Campo de uma struct
  - É por meio dela que sabemos se um determinado elemento está a frente ou não de outros no conjunto
  - Ou seja, devemos modificar o algoritmo para que a comparação das chaves seja feita utilizando um determinado campo da struct.