Ordenação

Ordenação

```
void bubblesort(int v[], int n){
  int i, k, t;
  do{
    k = 0;
    for(i=0; i < (n-1); i++)
      if(v[i] > v[i+1]) {
        t=v[i]; v[i]=v[i+1]; v[i+1]= t;
        k++;
      }
  } while (k > 0);
}
```

Limite inferior

- Para n valores distintos podemos ter **n!** sequências diferentes.
- Uma comparação na melhor das hipóteses divide o número de possibilidades pela metade.
- O melhor algoritmo de ordenação portanto irá fazer $\mathbf{log_2(n!)}$ comparações **~** $\mathbf{n\ log_2(n)}$

Ordenação por seleção

para todo i desde 1 até n-1 {

determine j tal que v[j] é o menor elemento no intervalo [i+1, n];

troque v[i] c/ v[j];

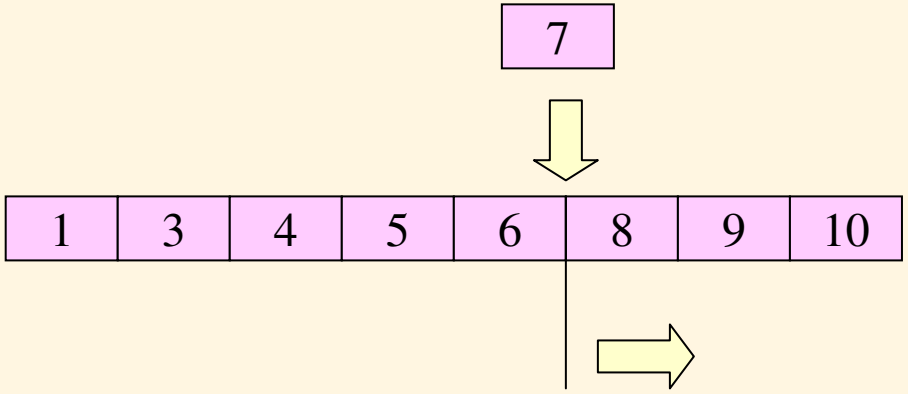
}

Ordenação por seleção

```
#define key(A) (A)
#define less(A, B) (key(A) < key(B))
#define exch(A, B) { Item t = A; A = B; B = t; }
typedef int Item

void selectionsort(Item a[],int n) {
  int i, j;
  for(i = 0; i < (n-1); i++){
    int min = i;
    for(j = i+1; j < n; j++)
      if (less(a[j], a[min])) min = j;
      exch(a[i], a[min]);
  }
}
```

Ordenação por inserção



1. Procurar o ponto de inserção
2. Deslocar todos os elementos posteriores
3. Inserir o novo elemento

Ordenação por inserção

```
#define compexch(A, B) if (less(B, A)) exch(A, B)

void insertionsort(Item a[], int n){
  int i;
  for(i = 1; i < n; i++) compexch(a[0], a[i]);
  for(i = 2; i < n; i++){
    int j = i; Item v = a[i];
    while(less(v, a[j-1])) { a[j] = a[j-1]; j--;}
    a[j] = v;
  }
}
```

Bublesort

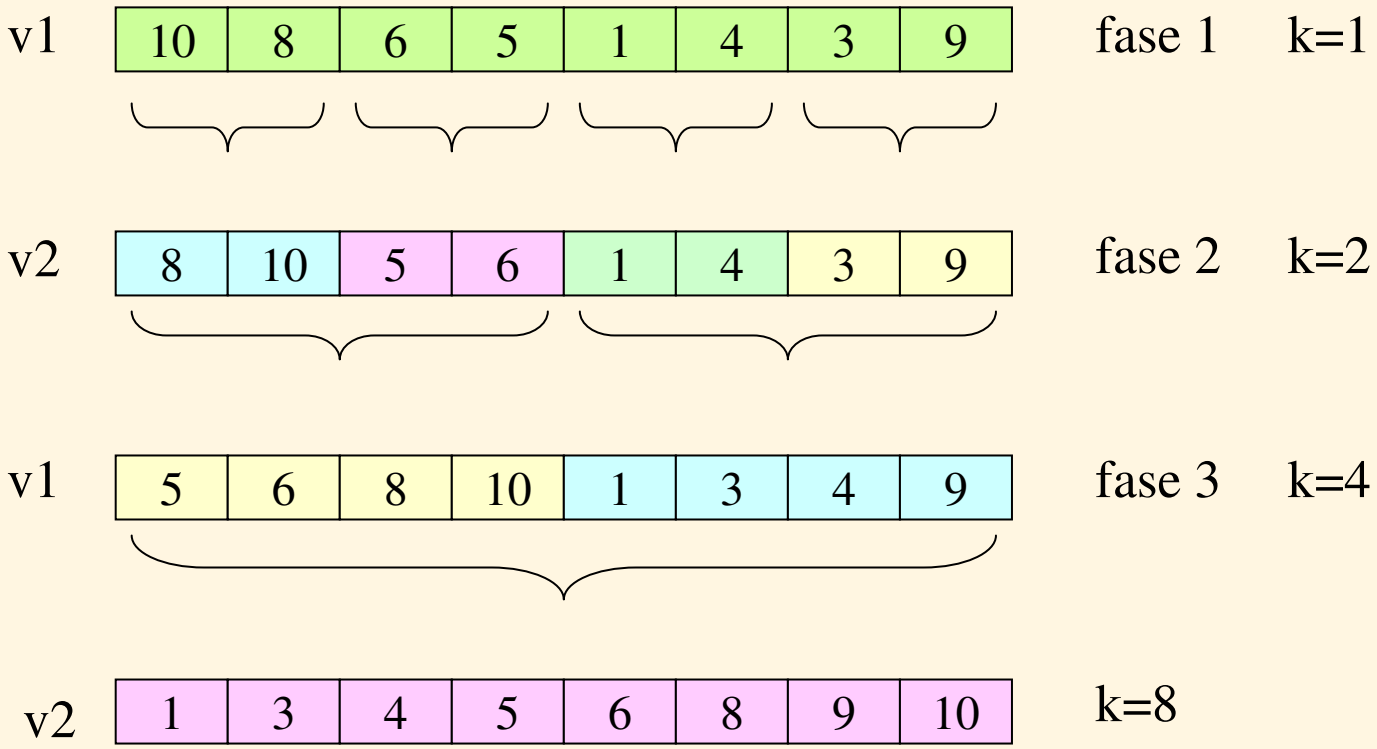
- baseado na troca de elementos adjacentes

```
void bubblesort(Item a[], int n){
  int i, j;
  for(i = 1; i < n; i++)
    for(j = i; j > 0; j--)
      compexch(a[j-1], a[j]);
}
```

Shellsort

```
void shellsort(Item a[],int n){
  int i, j, h;
  for(h = 1; h <= (n-2)/3; h = 3*h+1) ;
  for( ; h > 0; h /= 3)
    for(i = h; i < n; i++){
      int j = i; Item v = a[i];
      while (j >= h && less(v, a[j-h])){
        a[j] = a[j-h]; j -= h;
      }
      a[j] = v;
    }
}
```

Mergesort: ordenação por intercalação



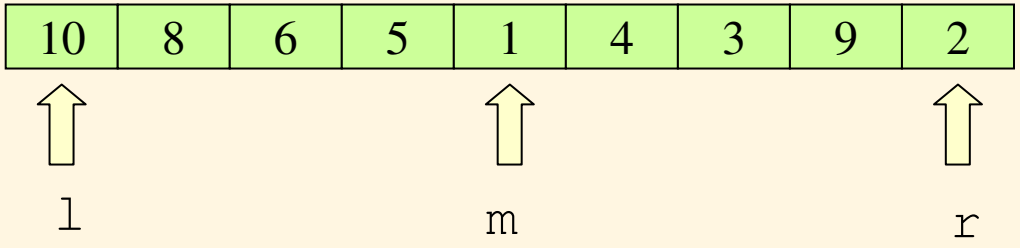
Intercalação

```
void mergeAB( Item a[],Item b[],Item c[],
              int N,int M)
{
  int i, j, k;
  for(i=0, j=0, k=0; k < N+M; k++){
    if (i == N) { c[k] = b[j++]; continue; }
    if (j == M) { c[k] = a[i++]; continue; }
    c[k] = (less(a[i], b[j])) ? a[i++] : b[j++];
  }
}
```

Mergesort

```
int *aux;

void merge(Item a[], int l, int m, int r){
  int i, j, k;
  for(i = m+1; i > l; i--) aux[i-1] = a[i-1];
  for(j = m; j < r; j++) aux[r+m-j] = a[j+1];
  for(k = l; k <= r; k++)
    if(less(aux[i], aux[j]))
      a[k] = aux[i++]; else a[k] = aux[j--];
}
```



Mergesort

```
void mergesort(Item a[], int l, int r){
  int m = (r+l)/2;
  if (r <= l) return;
  mergesort(a, l, m);
  mergesort(a, m+1, r);
  merge(a, l, m, r);
}
```

Quicksort

- Idéia geral

quickSort(seqüência S){

escolha ao acaso um elemento x em S;

S1 = seqüência dos elementos de S menores que x;

S2 = seqüência dos elementos de S maiores que x;

retorne S1 | x | S2

}

Quicksort

- função de partição

```
int partition(Item a[], int l, int r) {
  int i = l-1, j = r; Item v = a[r];
  for(;;){
    while (less(a[++i], v)) ;
    while (less(v, a[--j])) if (j == l) break;
    if (i >= j) break;
    exch(a[i], a[j]);
  }
  exch(a[i], a[r]);
  return i;
}
```

Quicksort

- Ordenação

```
void quicksort(Item a[], int l, int r){
  int i;
  if (r <= l) return;
  i = partition(a, l, r);
  quicksort(a, l, i-1);
  quicksort(a, i+1, r);
}
```

Ordenação lexicográfica
(bucket sort, radix sort, bin sort)

- idéia geral
  - ordenar pela coluna i:

    1 - para todo nome n na lista

    inserir n na fila correspondente ao

    caracter n[i];

    2 - concatenar as filas numa lista

Ordenação lexicográfica
(bucket sort, radix sort, bin sort)

- Ordenação de uma lista de nomes, todos com um mesmo tamanho n:

para todo i desde n até 1

ordenar a lista pela coluna i

Implementação

```
#define N 'z'-'a'+1
void bucketsort(apItem *lista, int size){
  apItem v[N];
  int i,j; apItem p;
  for(i = 0; i < N; i++) v[i] = NULL;
  for(i = size-1; i >= 0; i--){
    while((p = removeFirst(lista)) != NULL)
      insert(p,&v[p->info[i]-'a']);
    for(j=0; j < N; j++) concat(lista,&v[j]);
  }
}
```

Funções auxiliares

- Remover o primeiro elemento de uma lista circular:

```
apItem removeFirst(apItem *lista);
```

- Concatenar duas listas circulares:

```
void concat(apItem *lista1, apItem *lista2);
```

- Inserir um item numa lista circular:

```
void insert(apItem p, apItem *lista)
```

Tempo de execução

- Número de nomes : n
- Tamanho de cada nome: k
- Tempo para as funções auxiliares: constante
- Tempo de ordenação: O(nk) (cada caracter é examinado uma única vez).
- Comparado com heapsort ou mergesort:
  - número de comparações: O(n log n), mas cada comparação examina k caracteres.
  - tempo: O(k.n log n)

Nomes com tamanhos diferentes

- Duas abordagens:
  1. completar os nomes menores com espaços
  2. separar os nomes em listas organizadas pelo tamanho e ir introduzindo os nomes no processo de ordenação à medida que eles contenham a coluna sendo considerada.

A segunda abordagem é mais interessante porque é mais rápida e usa menos memória.