Ordenação de dados eficiente

Profª. Rose Yuri Shimizu

Roteiro

1. Algoritmos de Ordenação Eficientes
   - Heap Sort
   - Intro Sort
   - Counting Sort
   - Radix Sort

Roteiro

1. Algoritmos de Ordenação Eficientes
   - Heap Sort
   - Intro Sort
   - Counting Sort
   - Radix Sort

Algoritmos de Ordenação Eficientes - Heap Sort

- Usar as filas de prioridades para ordenar elementos
- Construir e destruir a heap da fila de prioridades

Algoritmos de Ordenação Eficientes - Heap Sort

- Usar as filas de prioridades para ordenar elementos
- Construir e destruir a heap da fila de prioridades
- Fase 1: construção/conserto da heap-ordenada (fila prioridades)
  - Topo é o de maior prioridade
  - Não há garantia de ordenação de todos os itens
  - Somente que o pai é maior que seus filhos
- Fase 2: ordenação por remoção

Algoritmos de Ordenação Eficientes - Heap Sort

- Conceito: fila de prioridades para ordenação
- Exemplo usando somente a interface:
  - Cria-se uma fila de prioridades
    - ★ Utiliza-se espaço extra
  - Fase 1: construção/conserto da heap
    - ★ Construção da heap por inserção
    - ★ `fixUp` para posicionar na heap
  - Fase 2: ordenação (decrescente)
    - ★ Ordenação por remoção (maior prioridade)
    - ★ Reorganização da fila de prioridades
    - ★ Cada item removido volta para o vetor original
  - Custo proporcional a $2N \log N$ - $O(n \log n)$

Algoritmos de Ordenação Eficientes - Heap Sort

```
1  //Ideia de utilização da fila de prioridades
2  void PQsort(Item *v, int l, int r) {
3      PQinit(r-l+1);
4      for(int k=l; k<=r; k++)
5      {
6          PQinsert(v[k]);
7      }
8      for(int k=r; k>=l; k--)
9      {
10         v[k] = PQdelmax();
11     }
12 }
```

Algoritmos de Ordenação Eficientes - Heap Sort

- Versão utilizada/implementada
- Usar diretamente as funções internas: fixUp(swim), fixDown(sink)
- Não há a necessidade de espaços extras
  - Vetor original é utilizado para construir a heap
  - Vetor original como uma heap desordenada

Algoritmos de Ordenação Eficientes - Heap Sort

- Versão utilizada/implementada
- Usar diretamente as funções internas: fixUp(swim), fixDown(sink)
- Não há a necessidade de espaços extras
  - Vetor original é utilizado para construir a heap
  - Vetor original como uma heap desordenada
- Fase 1: conserto da heap desordenada
  - Consertar debaixo para cima
  - Consertar subárvores $\rightarrow$ sub-heaps ordenadas
  - Assim, cada nó superior é reorganizado em uma sub-heap já ordenada
  - Lembrando que cada nó é uma raiz de uma subárvore

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 1: construção/conserto da heap
- Debaixo p/ cima: k inicial?

```
1 int v[10] = {16, 94, 43, 10, 22, 18, 40, 12, 68};
2 int *pq = v-1; //pq aponta &v[0]-1
3                //pq[1] = v[0]
```

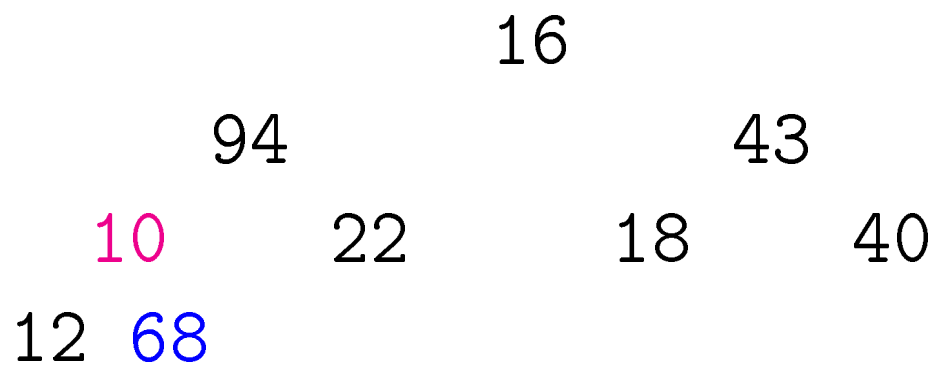
```
 v       0   1   2   3   4   5   6   7   8
pq  0   1   2   3   4   5   6   7   8   9
    [  16  94  43  10  22  18  40  12  68]
```

```
                16
          94            43
      10      22    18      40
   12  68
```

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 1: construção/conserto da heap
- k = pai da última folha

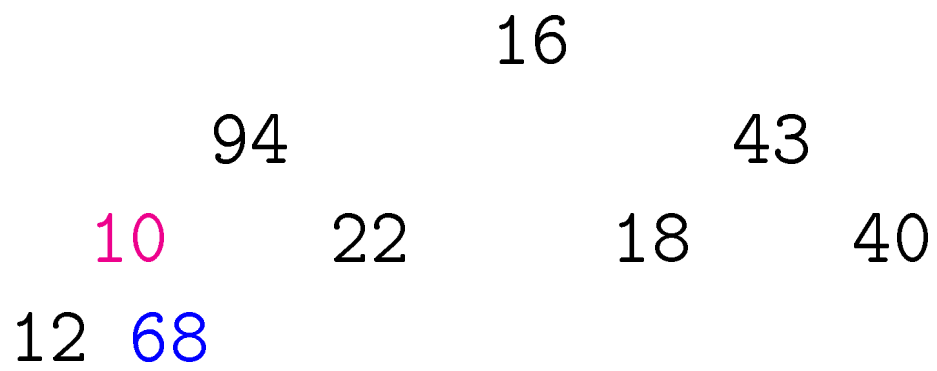
```
 v        0   1   2   3   4   5   6   7   8
pq  0   1   2   3   4   5   6   7   8   9
    [  16  94  43  10  22  18  40  12  68]
```



Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 1: construção/conserto da heap
- $k = N/2 \rightarrow$ `fixDown(k)`

v		0	1	2	3	4	5	6	7	8
pq	0	1	2	3	4	5	6	7	8	9
	[	16	94	43	10	22	18	40	12	68]



Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 1: construção/conserto da heap
- $k = N/2 \rightarrow$ `fixDown(k)`
- `fixDown(i)`: pai (i) $<$ filhos ($2i$ e $2i+1$)?

v		0	1	2	3	4	5	6	7	8
pq	0	1	2	3	4	5	6	7	8	9
	[	16	94	43	10	22	18	40	12	68]

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 1: construção/conserto da heap
- $k = N/2 \rightarrow$ `fixDown(k)`
- `fixDown(i)`: swap e $(i = 2i + 1) < N$?

v		0	1	2	3	4	5	6	7	8
pq	0	1	2	3	4	5	6	7	8	9
	[	16	94	43	68	22	18	40	12	10]

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 1: construção/conserto da heap
- $k = N/2 \rightarrow$ `fixDown(k)`
- `fixDown(i)`: swap e $(i = 2i) < N$? Não: sub-heap ordenada.

v		0	1	2	3	4	5	6	7	8
pq	0	1	2	3	4	5	6	7	8	9
	[	16	94	43	68	22	18	40	12	10]

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 1: construção/conserto da heap
- Próxima subárvore: $pq[x]$, sendo $x > k$, possuem filhos?

v		0	1	2	3	4	5	6	7	8
pq	0	1	2	3	4	5	6	7	8	9
	[	16	94	43	68	22	18	40	12	10]

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 1: construção/conserto da heap
- Próxima subárvore: `(--k >= 1) ? fixDown(k)`

v		0	1	2	3	4	5	6	7	8
pq	0	1	2	3	4	5	6	7	8	9
	[	16	94	43	68	22	18	40	12	10]

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 1: construção/conserto da heap
- `(--k >= 1) ? fixDown(k)`
- `fixDown(i)`: pai (i) $<$ filhos ($2i$ e $2i+1$)?

v		0	1	2	3	4	5	6	7	8
pq	0	1	2	3	4	5	6	7	8	9
	[	16	94	43	68	22	18	40	12	10]

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 1: construção/conserto da heap
- `(--k >= 1) ? fixDown(k)`

v		0	1	2	3	4	5	6	7	8
pq	0	1	2	3	4	5	6	7	8	9
	[	16	94	43	68	22	18	40	12	10]

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 1: construção/conserto da heap
- `(--k >= 1) ? fixDown(k)`
- `fixDown(i)`: pai (i) $<$ filhos ($2i$ e $2i+1$)?

v		0	1	2	3	4	5	6	7	8
pq	0	1	2	3	4	5	6	7	8	9
	[	16	94	43	68	22	18	40	12	10]

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 1: construção/conserto da heap
- `(--k >= 1) ? fixDown(k)`

v		0	1	2	3	4	5	6	7	8
pq	0	1	2	3	4	5	6	7	8	9
	[	16	94	43	68	22	18	40	12	10]

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 1: construção/conserto da heap
- `(--k >= 1) ? fixDown(k)`
- `fixDown(i)`: pai (i) $<$ filhos ($2i$ e $2i+1$)?

v		0	1	2	3	4	5	6	7	8
pq	0	1	2	3	4	5	6	7	8	9
	[	16	94	43	68	22	18	40	12	10]

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 1: construção/conserto da heap
- `(--k >= 1) ? fixDown(k)`
- `fixDown(i)`: swap e $i = 2i$

v		0	1	2	3	4	5	6	7	8
pq	0	1	2	3	4	5	6	7	8	9
	[	94	16	43	68	22	18	40	12	10]

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 1: construção/conserto da heap
- `(--k >= 1) ? fixDown(k)`
- `fixDown(i)`: pai (i) $<$ filhos ($2i$ e $2i+1$)?

v		0	1	2	3	4	5	6	7	8
pq	0	1	2	3	4	5	6	7	8	9
	[	94	16	43	68	22	18	40	12	10]

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 1: construção/conserto da heap
- `(--k >= 1) ? fixDown(k)`
- `fixDown(i)`: swap e $i = 2i$

v		0	1	2	3	4	5	6	7	8
pq	0	1	2	3	4	5	6	7	8	9
	[	94	68	43	16	22	18	40	12	10]

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 1: construção/conserto da heap
- `(--k >= 1) ? fixDown(k)`
- `fixDown(i)`: pai (i) $<$ filhos ($2i$ e $2i+1$)?

v		0	1	2	3	4	5	6	7	8
pq	0	1	2	3	4	5	6	7	8	9
	[	94	68	43	16	22	18	40	12	10]

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 1: construção/conserto da heap
- `(--k >= 1)` ? heap ordenada.

v		0	1	2	3	4	5	6	7	8
pq	0	1	2	3	4	5	6	7	8	9
	[	94	68	43	16	22	18	40	12	10]

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 2: ordenação (decrescente)
  - Remover o máximo repetidamente
  - Raiz para o fim da fila
  - Diminui-se o tamanho da fila
  - fixDown da raiz

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 2: ordenação (decrescente) - enquanto `N > 1`
- `exch(pq[1], pq[N])`

v		0	1	2	3	4	5	6	7	8
pq	0	1	2	3	4	5	6	7	8	9
	[	94	68	43	16	22	18	40	12	10]

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 2: ordenação (decrescente) - enquanto `N > 1`
- `fixDown(1, --N)`

v		0	1	2	3	4	5	6	7	8
pq	0	1	2	3	4	5	6	7	8	
	[	10	68	43	16	22	18	40	12	**94]**

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 2: ordenação (decrescente) - enquanto `N > 1`
- `fixDown(1, --N)`

v		0	1	2	3	4	5	6	7	8
pq	0	1	2	3	4	5	6	7	8	
	[	68	10	43	16	22	18	40	12	**94]**

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 2: ordenação (decrescente) - enquanto `N > 1`
- `fixDown(1, --N)`

v		0	1	2	3	4	5	6	7	8
pq	0	1	2	3	4	5	6	7	8	
	[	68	10	43	16	22	18	40	12	**94]**

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 2: ordenação (decrescente) - enquanto `N > 1`
- `fixDown(1, --N)`

v		0	1	2	3	4	5	6	7	8
pq	0	1	2	3	4	5	6	7	8	
	[	68	22	43	16	10	18	40	12	**94]**

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 2: ordenação (decrescente) - enquanto `N > 1`
- `exch(pq[1], pq[N])`

v		0	1	2	3	4	5	6	7	8
pq	0	1	2	3	4	5	6	7	8	
	[	68	22	43	16	10	18	40	12	**94]**

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 2: ordenação (decrescente) - enquanto `N > 1`
- `exch(pq[1], pq[N])`

v		0	1	2	3	4	5	6	7	8
pq	0	1	2	3	4	5	6	7	8	
	[	12	22	43	16	10	18	40	68	**94]**

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 2: ordenação (decrescente) - enquanto `N > 1`
- `fixDown(1, --N)`

v		0	1	2	3	4	5	6	7	8
pq	0	1	2	3	4	5	6	7		
	[	12	22	43	16	10	18	40	**68**	**94]**

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 2: ordenação (decrescente) - enquanto `N > 1`
- `fixDown(1, --N)`

v		0	1	2	3	4	5	6	7	8
pq	0	1	2	3	4	5	6	7		
	[	43	22	12	16	10	18	40	**68**	**94]**

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 2: ordenação (decrescente) - enquanto `N > 1`
- `fixDown(1, --N)`

v		0	1	2	3	4	5	6	7	8
pq	0	1	2	3	4	5	6	7		
	[	43	22	12	16	10	18	40	**68**	**94]**

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 2: ordenação (decrescente) - enquanto `N > 1`
- `fixDown(1, --N)`

v		0	1	2	3	4	5	6	7	8
pq	0	1	2	3	4	5	6	7		
	[	43	22	40	16	10	18	12	**68**	**94]**

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 2: ordenação (decrescente) - enquanto `N > 1`
- `exch(pq[1], pq[N])`

v		0	1	2	3	4	5	6	7	8
pq	0	1	2	3	4	5	6	7		
	[	43	22	40	16	10	18	12	**68**	**94]**

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 2: ordenação (decrescente) - enquanto `N > 1`
- `exch(pq[1], pq[N])`

v		0	1	2	3	4	5	6	7	8
pq	0	1	2	3	4	5	6	7		
	[	12	22	40	16	10	18	43	**68**	**94]**

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 2: ordenação (decrescente) - enquanto `N > 1`
- `fixDown(1, --N)`

v		0	1	2	3	4	5	6	7	8
pq	0	1	2	3	4	5	6			
	[	12	22	40	16	10	18	**43**	**68**	**94]**

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 2: ordenação (decrescente) - enquanto `N > 1`
- `fixDown(1, --N)`

v		0	1	2	3	4	5	6	7	8
pq	0	1	2	3	4	5	6			
	[	40	22	12	16	10	18	**43**	**68**	**94]**

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 2: ordenação (decrescente) - enquanto `N > 1`
- `fixDown(1, --N)`

v		0	1	2	3	4	5	6	7	8
pq	0	1	2	3	4	5	6			
	[	40	22	12	16	10	18	**43**	**68**	**94]**

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 2: ordenação (decrescente) - enquanto `N > 1`
- `fixDown(1, --N)`

v		0	1	2	3	4	5	6	7	8
pq	0	1	2	3	4	5	6			
	[	40	22	18	16	10	12	**43**	**68**	**94]**

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 2: ordenação (decrescente) - enquanto `N > 1`
- `exch(pq[1], pq[N])`

v		0	1	2	3	4	5	6	7	8
pq	0	1	2	3	4	5	6			
	[	40	22	18	16	10	12	**43**	**68**	**94]**

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 2: ordenação (decrescente) - enquanto `N > 1`
- `exch(pq[1], pq[N])`

v		0	1	2	3	4	5	6	7	8
pq	0	1	2	3	4	5	6			
	[	12	22	18	16	10	40	**43**	**68**	**94]**

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 2: ordenação (decrescente) - enquanto `N > 1`
- `fixDown(1, --N)`

v		0	1	2	3	4	5	6	7	8
pq	0	1	2	3	4	5				
	[	12	22	18	16	10	**40**	**43**	**68**	**94]**

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 2: ordenação (decrescente) - enquanto `N > 1`
- `fixDown(1, --N)`

v		0	1	2	3	4	5	6	7	8
pq	0	1	2	3	4	5				
	[	22	12	18	16	10	**40**	**43**	**68**	**94]**

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 2: ordenação (decrescente) - enquanto `N > 1`
- `fixDown(1, --N)`

v		0	1	2	3	4	5	6	7	8
pq	0	1	2	3	4	5				
	[	22	12	18	16	10	**40**	**43**	**68**	**94]**

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 2: ordenação (decrescente) - enquanto `N > 1`
- `fixDown(1, --N)`

v		0	1	2	3	4	5	6	7	8
pq	0	1	2	3	4	5				
	[	22	16	18	12	10	**40**	**43**	**68**	**94]**

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 2: ordenação (decrescente) - enquanto `N > 1`
- `exch(pq[1], pq[N])`

v		0	1	2	3	4	5	6	7	8
pq	0	1	2	3	4	5				
	[	22	16	18	12	10	**40**	**43**	**68**	**94]**

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 2: ordenação (decrescente) - enquanto `N > 1`
- `exch(pq[1], pq[N])`

v		0	1	2	3	4	5	6	7	8
pq	0	1	2	3	4	5				
	[	10	16	18	12	22	**40**	**43**	**68**	**94]**

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 2: ordenação (decrescente) - enquanto `N > 1`
- `fixDown(1, --N)`

v		0	1	2	3	4	5	6	7	8
pq	0	1	2	3	4					
	[	10	16	18	12	**22**	**40**	**43**	**68**	**94]**

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 2: ordenação (decrescente) - enquanto `N > 1`
- `fixDown(1, --N)`

v		0	1	2	3	4	5	6	7	8
pq	0	1	2	3	4					
	[	18	16	10	12	**22**	**40**	**43**	**68**	**94]**

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 2: ordenação (decrescente) - enquanto `N > 1`
- `exch(pq[1], pq[N])`

v		0	1	2	3	4	5	6	7	8
pq	0	1	2	3	4					
	[	18	16	10	12	**22**	**40**	**43**	**68**	**94]**

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 2: ordenação (decrescente) - enquanto `N > 1`
- `exch(pq[1], pq[N])`

v		0	1	2	3	4	5	6	7	8
pq	0	1	2	3	4					
	[	12	16	10	18	**22**	**40**	**43**	**68**	**94]**

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 2: ordenação (decrescente) - enquanto `N > 1`
- `fixDown(1, --N)`

```
 v       0    1    2    3    4    5    6    7    8
pq   0   1    2    3
     [  12   16   10   18   22   40   43   68   94]
```

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 2: ordenação (decrescente) - enquanto `N > 1`
- `fixDown(1, --N)`

```
 v      0   1   2   3   4   5   6   7   8
pq  0   1   2   3
    [  16  12  10  18  22  40  43  68  94]
```

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 2: ordenação (decrescente) - enquanto `N > 1`
- `exch(pq[1], pq[N])`

```
 v       0   1   2   3   4   5   6   7   8
pq   0   1   2   3
    [  16  12  10  18  22  40  43  68  94]
```

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 2: ordenação (decrescente) - enquanto `N > 1`
- `exch(pq[1], pq[N])`

```
 v       0   1   2   3   4   5   6   7   8
pq   0   1   2   3
    [  10  12  16  18  22  40  43  68  94]
```

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 2: ordenação (decrescente) - enquanto `N > 1`
- `fixDown(1, --N)`

```
 v      0   1   2   3   4   5   6   7   8
pq  0   1   2
    [  10  12  16  18  22  40  43  68  94]
```

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 2: ordenação (decrescente) - enquanto `N > 1`
- `fixDown(1, --N)`

```
 v       0    1    2    3    4    5    6    7    8
pq   0   1    2
     [  12   10   16   18   22   40   43   68   94]
```

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 2: ordenação (decrescente) - enquanto `N > 1`
- `exch(pq[1], pq[N])`

v		0	1	2	3	4	5	6	7	8
pq	0	1	2							
	[	12	10	**16**	**18**	**22**	**40**	**43**	**68**	**94]**

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 2: ordenação (decrescente) - enquanto `N > 1`
- `exch(pq[1], pq[N])`

v		0	1	2	3	4	5	6	7	8
pq	0	1	2							
	[	10	12	**16**	**18**	**22**	**40**	**43**	**68**	**94]**

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 2: ordenação (decrescente) - enquanto `N > 1`
- `fixDown(1, --N)`: tem filhos?

```
 v      0    1    2    3    4    5    6    7    8
pq   0   1
     [  10   12   16   18   22   40   43   68   94]
```

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 2: ordenação (decrescente)
- N>1?

```
 v       0    1    2    3    4    5    6    7    8
pq   0   1
     [  10   12   16   18   22   40   43   68   94]
```

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 2: ordenação (decrescente)
- Vetor ordenado.

v	0	1	2	3	4	5	6	7	8
	10	**12**	**16**	**18**	**22**	**40**	**43**	**68**	**94**

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 1: construção/conserto da heap
  - Inicializa da metade do vetor
  - $N/2$: pai dos nós folhas
    - ★ Pular sub-heaps de tamanho 1
  - Termina na posição 1

```
1 for(int k=N/2; k>=1; k--)
2 {
3     fixDown(k, N);
4 }
```

Algoritmos de Ordenação Eficientes - Heap Sort

- Fase 2: ordenação (decrescente)
  - Remover o máximo repetidamente
  - Raiz para o fim da fila
  - Diminui-se o tamanho da fila
  - fixDown da raiz

```
1 while(N>1)
2 {
3     exch(pq[1], pq[N]);
4     fixDown(1, --N);
5 }
```

Algoritmos de Ordenação Eficientes - Heap Sort

```
1  void heap_sort(Item *v, int l, int r) {
2      pq = v+l-1; //fila de prioridades construída em v
3                  //uma posição anterior a v[l]
4                  //se l=0 -> pq[1] = v[0]
5      N = r-l+1;
6
7      for(int k=N/2; k>=1; k--)
8      {
9          fixDown(k, N);
10     }
11
12     while(N>1)
13     {
14         exch(pq[1], pq[N]);
15         fixDown(1, --N);
16     }
17 }
```

Algoritmos de Ordenação Eficientes - Heap Sort

- Complexidade proporcional a $2N \log N$: $O(n \log n)$
- In-place: sim
- Estabilidade: não é estável
- Adaptatividade:
  - Complexidade não é alterada, pois mesmo que na criação da heap há a diminuição do custo, na ordenação continuamos com custo linearítmico
  - Portanto, não é adaptativo

Roteiro

1. Algoritmos de Ordenação Eficientes
   - Heap Sort
   - Intro Sort
   - Counting Sort
   - Radix Sort

Algoritmos de Ordenação Eficientes - Intro Sort

- É uma importante combinação de algoritmos de ordenação interna, utilizado no C++, C#
  - Java é quicksort three-way
    - ★ Particionar o vetor em três partes (three-way): menores, iguais e maiores
- Híbrido:
  - quick + merge(mais espaço) + insertion
  - quick + heap(maior constante) + insertion
- Solução para utilizar as eficiências e evitar as deficiências de cada método
  - insertion: adaptativo
  - quick: bom desempenho na maioria dos casos e inplace
  - merge e heap: desempenho previsível para todos os casos
  - quando a profundidade da recursividade atinge um máximo estipulado, alterna-se para outro método de ordenação
- Complexidade no pior caso: $O(n \log n)$
- In-place
  - Merge: espaço extra, proporcional a N
  - Heap e Quick: sim
- Não estável
- Não adaptativo

Algoritmos de Ordenação Eficientes - Intro Sort

```
1  void intro(int *v, int l, int r, int maxdepth) {
2    if(r-l<=15){
3      return;
4
5    } else if(maxdepth == 0) {
6      //merge_sort(v, l, r);
7      heap_sort(v, l, r);
8
9    } else {
10     compexch(v[l], v[(l+r)/2]);
11     compexch(v[l], v[r]);
12     compexch(v[r], v[(l+r)/2]);
13
14     int p = partition(v, l, r);
15     intro(v, l, p-1, maxdepth-1);
16     intro(v, p+1, r, maxdepth-1);
17   }
18 }
```

Algoritmos de Ordenação Eficientes - Intro Sort

```
1 void intro_sort(int *v,  int l, int r)
2 {
3     //proporcional à altura de uma árvore balanceada
4     int maxdepth = 2*((int)log2((double)(r-l+1)));
5
6     intro(v, l, r, maxdepth);
7     insertion_sort(v, l, r);
8 }
```

Intro Sort - Execução de um exemplo reduzido

```
1 v[] = 11 10 9 8 7 6 1 4 13 12 5
2 introsort(v, 0, 10)
```

Intro Sort - Execução de um exemplo reduzido

```
1 v[] = 11 10 9 8 7 6 1 4 13 12 5
2 introsort(v, 0, 10)
3 |   maxdepth = 2
4 |   intro(v, 0, 10, 2) //quick sem mediana
```

Intro Sort - Execução de um exemplo reduzido

```
1 v[] = 11 10 9 8 7 6 1 4 13 12 5
2 introsort(v, 0, 10)
3 |   maxdepth = 2
4 |   intro(v, 0, 10, 2) //quick sem mediana
5 |   |   partition(v, 0, 10) : 11 10 9 8 7 6 1 4 13 12 5
```

Intro Sort - Execução de um exemplo reduzido

```
1 v[] = 11 10 9 8 7 6 1 4 13 12 5
2 introsort(v, 0, 10)
3 |   maxdepth = 2
4 |   intro(v, 0, 10, 2) //quick sem mediana
5 |   |   partition(v, 0, 10) : 11 10 9 8 7 6 1 4 13 12 5
6 |   |   |                       4 10 9 8 7 6 1 11 13 12 5
```

Intro Sort - Execução de um exemplo reduzido

```
1 v[] = 11 10 9 8 7 6 1 4 13 12 5
2 introsort(v, 0, 10)
3 |   maxdepth = 2
4 |   intro(v, 0, 10, 2) //quick sem mediana
5 |   |   partition(v, 0, 10) : 11 10 9 8 7 6 1 4 13 12 5
6 |   |   |                       4 10 9 8 7 6 1 11 13 12 5
7 |   |   |                       4 1 9 8 7 6 10 11 13 12 5
```

Intro Sort - Execução de um exemplo reduzido

```
1 v[] = 11 10 9 8 7 6 1 4 13 12 5
2 introsort(v, 0, 10)
3 |   maxdepth = 2
4 |   intro(v, 0, 10, 2) //quick sem mediana
5 |   |   partition(v, 0, 10) : 11 10 9 8 7 6 1 4 13 12 5
6 |   |   |                     4 10 9 8 7 6 1 11 13 12 5
7 |   |   |                     4 1 9 8 7 6 10 11 13 12 5
8 |   |   |                     4 1 5 8 7 6 10 11 13 12 9
```

Intro Sort - Execução de um exemplo reduzido

```
1  v[] = 11 10 9 8 7 6 1 4 13 12 5
2  introsort(v, 0, 10)
3  |   maxdepth = 2
4  |   intro(v, 0, 10, 2) //quick sem mediana
5  |   |   partition(v, 0, 10) : 11 10 9 8 7 6 1 4 13 12 5
6  |   |   |                     4 10 9 8 7 6 1 11 13 12 5
7  |   |   |                     4 1 9 8 7 6 10 11 13 12 5
8  |   |   |                     4 1 5 8 7 6 10 11 13 12 9
9  |   |   |
10 |   |   intro(v, 0, 1, 1) //quick esquerda
11 |   |   |   //r-l <= 2 : 4 1 5 8 7 6 10 11 13 12 9
12 |   |   |   return
```

Intro Sort - Execução de um exemplo reduzido

```
1  v[] = 11 10 9 8 7 6 1 4 13 12 5
2  introsort(v, 0, 10)
3  |   maxdepth = 2
4  |   intro(v, 0, 10, 2) //quick sem mediana
5  |   |   partition(v, 0, 10) : 11 10 9 8 7 6 1 4 13 12 5
6  |   |   |                     4 10 9 8 7 6 1 11 13 12 5
7  |   |   |                     4 1 9 8 7 6 10 11 13 12 5
8  |   |   |                     4 1 5 8 7 6 10 11 13 12 9
9  |   |   |
10 |   |   intro(v, 0, 1, 1) //quick esquerda
11 |   |   |   //r-l <= 2 : 4 1 5 8 7 6 10 11 13 12 9
12 |   |   |   return
13 |   |   |
14 |   |   intro(v, 2, 10, 1) //quick direita
```

Intro Sort - Execução de um exemplo reduzido

```
1  v[] = 11 10 9 8 7 6 1 4 13 12 5
2  introsort(v, 0, 10)
3  |   maxdepth = 2
4  |   intro(v, 0, 10, 2) //quick sem mediana
5  |   |   partition(v, 0, 10) : 11 10 9 8 7 6 1 4 13 12 5
6  |   |   |                     4 10 9 8 7 6 1 11 13 12 5
7  |   |   |                     4 1 9 8 7 6 10 11 13 12 5
8  |   |   |                     4 1 5 8 7 6 10 11 13 12 9
9  |   |   |
10 |   |   intro(v, 0, 1, 1) //quick esquerda
11 |   |   |   //r-l <= 2 : 4 1 5 8 7 6 10 11 13 12 9
12 |   |   |   return
13 |   |   |
14 |   |   intro(v, 2, 10, 1) //quick direita
15 |   |   |   partition(v, 0, 10) : 4 1 5 8 7 6 10 11 13 12 9
```

Intro Sort - Execução de um exemplo reduzido

```
1  v[] = 11 10 9 8 7 6 1 4 13 12 5
2  introsort(v, 0, 10)
3  |   maxdepth = 2
4  |   intro(v, 0, 10, 2) //quick sem mediana
5  |   |   partition(v, 0, 10) : 11 10 9 8 7 6 1 4 13 12 5
6  |   |   |                     4 10 9 8 7 6 1 11 13 12 5
7  |   |   |                     4 1 9 8 7 6 10 11 13 12 5
8  |   |   |                     4 1 5 8 7 6 10 11 13 12 9
9  |   |   |
10 |   |   intro(v, 0, 1, 1) //quick esquerda
11 |   |   |   //r-l <= 2 : 4 1 5 8 7 6 10 11 13 12 9
12 |   |   |   return
13 |   |   |
14 |   |   intro(v, 2, 10, 1) //quick direita
15 |   |   |   partition(v, 0, 10) : 4 1 5 8 7 6 10 11 13 12 9
16 |   |   |   |                       4 1 5 8 7 6 10 11 13 12 9
```

Intro Sort - Execução de um exemplo reduzido

```
1  v[] = 11 10 9 8 7 6 1 4 13 12 5
2  introsort(v, 0, 10)
3  |   maxdepth = 2
4  |   intro(v, 0, 10, 2) //quick sem mediana
5  |   |   partition(v, 0, 10) : 11 10 9 8 7 6 1 4 13 12 5
6  |   |   |                     4 10 9 8 7 6 1 11 13 12 5
7  |   |   |                     4 1 9 8 7 6 10 11 13 12 5
8  |   |   |                     4 1 5 8 7 6 10 11 13 12 9
9  |   |   |
10 |   |   intro(v, 0, 1, 1) //quick esquerda
11 |   |   |   //r-l <= 2 : 4 1 5 8 7 6 10 11 13 12 9
12 |   |   |   return
13 |   |   |
14 |   |   intro(v, 2, 10, 1) //quick direita
15 |   |   |   partition(v, 0, 10) : 4 1 5 8 7 6 10 11 13 12 9
16 |   |   |   |                     4 1 5 8 7 6 10 11 13 12 9
17 |   |   |   |                     4 1 5 8 7 6 9 11 13 12 10
```

Intro Sort - Execução de um exemplo reduzido

```
1  v[] = 11 10 9 8 7 6 1 4 13 12 5
2  introsort(v, 0, 10)
3  |   maxdepth = 2
4  |   intro(v, 0, 10, 2) //quick sem mediana
5  |   |   partition(v, 0, 10) : 11 10 9 8 7 6 1 4 13 12 5
6  |   |   |                       4 10 9 8 7 6 1 11 13 12 5
7  |   |   |                       4 1 9 8 7 6 10 11 13 12 5
8  |   |   |                       4 1 5 8 7 6 10 11 13 12 9
9  |   |   |
10 |   |   intro(v, 0, 1, 1) //quick esquerda
11 |   |   |   //r-l <= 2 : 4 1 5 8 7 6 10 11 13 12 9
12 |   |   |   return
13 |   |   |
14 |   |   intro(v, 2, 10, 1) //quick direita
15 |   |   |   partition(v, 0, 10) : 4 1 5 8 7 6 10 11 13 12 9
16 |   |   |   |                       4 1 5 8 7 6 10 11 13 12 9
17 |   |   |   |                       4 1 5 8 7 6 9 11 13 12 10
18 |   |   |   intro(v, 2, 6, 0) //quick esquerda
19 |   |   |   |   //maxdepth==0
```

Intro Sort - Execução de um exemplo reduzido

```
1  v[] = 11 10 9 8 7 6 1 4 13 12 5
2  introsort(v, 0, 10)
3  |   maxdepth = 2
4  |   intro(v, 0, 10, 2) //quick sem mediana
5  |   |   partition(v, 0, 10) : 11 10 9 8 7 6 1 4 13 12 5
6  |   |   |                     4 10 9 8 7 6 1 11 13 12 5
7  |   |   |                     4 1 9 8 7 6 10 11 13 12 5
8  |   |   |                     4 1 5 8 7 6 10 11 13 12 9
9  |   |   |
10 |   |   intro(v, 0, 1, 1) //quick esquerda
11 |   |   |   //r-l <= 2 : 4 1 5 8 7 6 10 11 13 12 9
12 |   |   |   return
13 |   |   |
14 |   |   intro(v, 2, 10, 1) //quick direita
15 |   |   |   partition(v, 0, 10) : 4 1 5 8 7 6 10 11 13 12 9
16 |   |   |   |                     4 1 5 8 7 6 10 11 13 12 9
17 |   |   |   |                     4 1 5 8 7 6 9 11 13 12 10
18 |   |   |   intro(v, 2, 6, 0) //quick esquerda
19 |   |   |   |   //maxdepth==0
20 |   |   |   |   heap(v, 2, 6)
```

```
v[] = 11 10 9 8 7 6 1 4 13 12 5
introsort(v, 0, 10)
|   maxdepth = 2
|   intro(v, 0, 10, 2) //quick sem mediana
|   |   partition(v, 0, 10) : 4 1 5 8 7 6 10 11 13 12 9
|   |   intro(v, 0, 1, 1) //quick esquerda
|   |   |   //r-l <= 2 : 4 1 5 8 7 6 10 11 13 12 9
|   |   |   return
|   |   intro(v, 2, 10, 1) //quick direita
|   |   |   partition(v, 0, 10) : 4 1 5 8 7 6 9 11 13 12 10
|   |   |   intro(v, 2, 6, 0) //quick esquerda
|   |   |   |   //maxdepth==0
|   |   |   |   heap(v, 2, 6) : 4 1 5 8 7 6 9 11 13 12 10
```

```
v[] = 11 10 9 8 7 6 1 4 13 12 5
introsort(v, 0, 10)
|   maxdepth = 2
|   intro(v, 0, 10, 2) //quick sem mediana
|   |   partition(v, 0, 10) : 4 1 5 8 7 6 10 11 13 12 9
|   |   intro(v, 0, 1, 1) //quick esquerda
|   |   |   //r-l <= 2 : 4 1 5 8 7 6 10 11 13 12 9
|   |   |   return
|   |   intro(v, 2, 10, 1) //quick direita
|   |   |   partition(v, 0, 10) : 4 1 5 8 7 6 9 11 13 12 10
|   |   |   intro(v, 2, 6, 0) //quick esquerda
|   |   |   |   //maxdepth==0
|   |   |   |   heap(v, 2, 6) : 4 1 5 8 7 6 9 11 13 12 10
|   |   |   |   |                   4 1 5 8 7 6 9 11 13 12 10
```

```
v[] = 11 10 9 8 7 6 1 4 13 12 5
introsort(v, 0, 10)
|   maxdepth = 2
|   intro(v, 0, 10, 2) //quick sem mediana
|   |   partition(v, 0, 10) : 4 1 5 8 7 6 10 11 13 12 9
|   |   intro(v, 0, 1, 1) //quick esquerda
|   |   |   //r-l <= 2 : 4 1 5 8 7 6 10 11 13 12 9
|   |   |   return
|   |   intro(v, 2, 10, 1) //quick direita
|   |   |   partition(v, 0, 10) : 4 1 5 8 7 6 9 11 13 12 10
|   |   |   intro(v, 2, 6, 0) //quick esquerda
|   |   |   |   //maxdepth==0
|   |   |   |   heap(v, 2, 6) : 4 1 5 8 7 6 9 11 13 12 10
|   |   |   |   |                 4 1 5 8 7 6 9 11 13 12 10
|   |   |   |   |                 4 1 5 8 7 6 9 11 13 12 10
```

```
v[] = 11 10 9 8 7 6 1 4 13 12 5
introsort(v, 0, 10)
|   maxdepth = 2
|   intro(v, 0, 10, 2) //quick sem mediana
|   |   partition(v, 0, 10) : 4 1 5 8 7 6 10 11 13 12 9
|   |   intro(v, 0, 1, 1) //quick esquerda
|   |   |   //r-l <= 2 : 4 1 5 8 7 6 10 11 13 12 9
|   |   |   return
|   |   intro(v, 2, 10, 1) //quick direita
|   |   |   partition(v, 0, 10) : 4 1 5 8 7 6 9 11 13 12 10
|   |   |   intro(v, 2, 6, 0) //quick esquerda
|   |   |   |   //maxdepth==0
|   |   |   |   heap(v, 2, 6) : 4 1 5 8 7 6 9 11 13 12 10
|   |   |   |   |                   4 1 5 8 7 6 9 11 13 12 10
|   |   |   |   |                   4 1 5 8 7 6 9 11 13 12 10
|   |   |   |   |                   4 1 5 6 7 8 9 11 13 12 10
```

```
v[] = 11 10 9 8 7 6 1 4 13 12 5
introsort(v, 0, 10)
|   maxdepth = 2
|   intro(v, 0, 10, 2) //quick sem mediana
|   |   partition(v, 0, 10) : 4 1 5 8 7 6 10 11 13 12 9
|   |   intro(v, 0, 1, 1) //quick esquerda
|   |   |   //r-l <= 2 : 4 1 5 8 7 6 10 11 13 12 9
|   |   |   return
|   |   intro(v, 2, 10, 1) //quick direita
|   |   |   partition(v, 0, 10) : 4 1 5 8 7 6 9 11 13 12 10
|   |   |   intro(v, 2, 6, 0) //quick esquerda
|   |   |   |   //maxdepth==0
|   |   |   |   heap(v, 2, 6) : 4 1 5 8 7 6 9 11 13 12 10
|   |   |   |   |                   4 1 5 8 7 6 9 11 13 12 10
|   |   |   |   |                   4 1 5 8 7 6 9 11 13 12 10
|   |   |   |   |                   4 1 5 6 7 8 9 11 13 12 10
|   |   |   |   |                   4 1 5 6 7 8 9 11 13 12 10
```

```
v[] = 11 10 9 8 7 6 1 4 13 12 5
introsort(v, 0, 10)
|   maxdepth = 2
|   intro(v, 0, 10, 2) //quick sem mediana
|   |   partition(v, 0, 10) : 4 1 5 8 7 6 10 11 13 12 9
|   |   intro(v, 0, 1, 1) //quick esquerda
|   |   |   //r-l <= 2 : 4 1 5 8 7 6 10 11 13 12 9
|   |   |   return
|   |   intro(v, 2, 10, 1) //quick direita
|   |   |   partition(v, 0, 10) : 4 1 5 8 7 6 9 11 13 12 10
|   |   |   intro(v, 2, 6, 0) //quick esquerda
|   |   |   |   //maxdepth==0
|   |   |   |   heap(v, 2, 6) : 4 1 5 8 7 6 9 11 13 12 10
|   |   |   |   |                   4 1 5 8 7 6 9 11 13 12 10
|   |   |   |   |                   4 1 5 8 7 6 9 11 13 12 10
|   |   |   |   |                   4 1 5 6 7 8 9 11 13 12 10
|   |   |   |   |                   4 1 5 6 7 8 9 11 13 12 10
|   |   |   |   |                   4 1 5 7 6 8 9 11 13 12 10
```

```
v[] = 11 10 9 8 7 6 1 4 13 12 5
introsort(v, 0, 10)
|   maxdepth = 2
|   intro(v, 0, 10, 2) //quick sem mediana
|   |   partition(v, 0, 10) : 4 1 5 8 7 6 10 11 13 12 9
|   |   intro(v, 0, 1, 1) //quick esquerda
|   |   |   //r-l <= 2 : 4 1 5 8 7 6 10 11 13 12 9
|   |   |   return
|   |   intro(v, 2, 10, 1) //quick direita
|   |   |   partition(v, 0, 10) : 4 1 5 8 7 6 9 11 13 12 10
|   |   |   intro(v, 2, 6, 0) //quick esquerda
|   |   |   |   //maxdepth==0
|   |   |   |   heap(v, 2, 6) : 4 1 5 8 7 6 9 11 13 12 10
|   |   |   |   |                   4 1 5 8 7 6 9 11 13 12 10
|   |   |   |   |                   4 1 5 8 7 6 9 11 13 12 10
|   |   |   |   |                   4 1 5 6 7 8 9 11 13 12 10
|   |   |   |   |                   4 1 5 6 7 8 9 11 13 12 10
|   |   |   |   |                   4 1 5 7 6 8 9 11 13 12 10
|   |   |   |   |                   4 1 5 7 6 8 9 11 13 12 10
```

```
v[] = 11 10 9 8 7 6 1 4 13 12 5
introsort(v, 0, 10)
|   maxdepth = 2
|   intro(v, 0, 10, 2) //quick sem mediana
|   |   partition(v, 0, 10) : 4 1 5 8 7 6 10 11 13 12 9
|   |   intro(v, 0, 1, 1) //quick esquerda
|   |   |   //r-l <= 2 : 4 1 5 8 7 6 10 11 13 12 9
|   |   |   return
|   |   intro(v, 2, 10, 1) //quick direita
|   |   |   partition(v, 0, 10) : 4 1 5 8 7 6 9 11 13 12 10
|   |   |   intro(v, 2, 6, 0) //quick esquerda
|   |   |   |   //maxdepth==0
|   |   |   |   heap(v, 2, 6) : 4 1 5 8 7 6 9 11 13 12 10
|   |   |   |   |                   4 1 5 8 7 6 9 11 13 12 10
|   |   |   |   |                   4 1 5 8 7 6 9 11 13 12 10
|   |   |   |   |                   4 1 5 6 7 8 9 11 13 12 10
|   |   |   |   |                   4 1 5 6 7 8 9 11 13 12 10
|   |   |   |   |                   4 1 5 7 6 8 9 11 13 12 10
|   |   |   |   |                   4 1 5 7 6 8 9 11 13 12 10
|   |   |   |   |                   4 1 5 6 7 8 9 11 13 12 10
```

```
v[] = 11 10 9 8 7 6 1 4 13 12 5
introsort(v, 0, 10)
|   maxdepth = 2
|   intro(v, 0, 10, 2) //quick sem mediana
|   |   partition(v, 0, 10) : 4 1 5 8 7 6 10 11 13 12 9
|   |   intro(v, 0, 1, 1) //quick esquerda
|   |   |   //r-l <= 2 : 4 1 5 8 7 6 10 11 13 12 9
|   |   |   return
|   |   intro(v, 2, 10, 1) //quick direita
|   |   |   partition(v, 0, 10) : 4 1 5 8 7 6 9 11 13 12 10
|   |   |   intro(v, 2, 6, 0) //quick esquerda
|   |   |   |   //maxdepth==0
|   |   |   |   heap(v, 2, 6) : 4 1 5 8 7 6 9 11 13 12 10
|   |   |   |   |                   4 1 5 8 7 6 9 11 13 12 10
|   |   |   |   |                   4 1 5 8 7 6 9 11 13 12 10
|   |   |   |   |                   4 1 5 6 7 8 9 11 13 12 10
|   |   |   |   |                   4 1 5 6 7 8 9 11 13 12 10
|   |   |   |   |                   4 1 5 7 6 8 9 11 13 12 10
|   |   |   |   |                   4 1 5 7 6 8 9 11 13 12 10
|   |   |   |   |                   4 1 5 6 7 8 9 11 13 12 10
|   |   |   |   |                   4 1 5 6 7 8 9 11 13 12 10
```

```
v[] = 11 10 9 8 7 6 1 4 13 12 5
introsort(v, 0, 10)
|   maxdepth = 2
|   intro(v, 0, 10, 2) //quick sem mediana
|   |   partition(v, 0, 10) : 4 1 5 8 7 6 10 11 13 12 9
|   |   intro(v, 0, 1, 1) //quick esquerda
|   |   |   //r-l <= 2 : 4 1 5 8 7 6 10 11 13 12 9
|   |   |   return
|   |   intro(v, 2, 10, 1) //quick direita
|   |   |   partition(v, 0, 10) : 4 1 5 8 7 6 9 11 13 12 10
|   |   |   intro(v, 2, 6, 0) //quick esquerda
|   |   |   |   //maxdepth==0
|   |   |   |   heap(v, 2, 6) : 4 1 5 8 7 6 9 11 13 12 10
|   |   |   |   |                   4 1 5 8 7 6 9 11 13 12 10
|   |   |   |   |                   4 1 5 8 7 6 9 11 13 12 10
|   |   |   |   |                   4 1 5 6 7 8 9 11 13 12 10
|   |   |   |   |                   4 1 5 6 7 8 9 11 13 12 10
|   |   |   |   |                   4 1 5 7 6 8 9 11 13 12 10
|   |   |   |   |                   4 1 5 7 6 8 9 11 13 12 10
|   |   |   |   |                   4 1 5 6 7 8 9 11 13 12 10
|   |   |   |   |                   4 1 5 6 7 8 9 11 13 12 10
|   |   |   |   |                   4 1 5 6 7 8 9 11 13 12 10
```

```
v[] = 11 10 9 8 7 6 1 4 13 12 5
introsort(v, 0, 10)
|   maxdepth = 2
|   intro(v, 0, 10, 2) //quick sem mediana
|   |   partition(v, 0, 10) : 4 1 5 8 7 6 10 11 13 12 9
|   |   intro(v, 0, 1, 1) //quick esquerda
|   |   |   //r-l <= 2 : 4 1 5 8 7 6 10 11 13 12 9
|   |   |   return
|   |   intro(v, 2, 10, 1) //quick direita
|   |   |   partition(v, 0, 10) : 4 1 5 8 7 6 9 11 13 12 10
|   |   |   intro(v, 2, 6, 0) //quick esquerda
|   |   |   |   //maxdepth==0
|   |   |   |   heap(v, 2, 6) : 4 1 5 8 7 6 9 11 13 12 10
|   |   |   |   |                   4 1 5 8 7 6 9 11 13 12 10
|   |   |   |   |                   4 1 5 8 7 6 9 11 13 12 10
|   |   |   |   |                   4 1 5 6 7 8 9 11 13 12 10
|   |   |   |   |                   4 1 5 6 7 8 9 11 13 12 10
|   |   |   |   |                   4 1 5 7 6 8 9 11 13 12 10
|   |   |   |   |                   4 1 5 7 6 8 9 11 13 12 10
|   |   |   |   |                   4 1 5 6 7 8 9 11 13 12 10
|   |   |   |   |                   4 1 5 6 7 8 9 11 13 12 10
|   |   |   |   |                   4 1 5 6 7 8 9 11 13 12 10
|   |   |   intro(v, 7, 10, 0) //quick direita
|   |   |   |   //maxdepth==0
|   |   |   |   heap(v, 7, 10) : 4 1 5 6 7 8 9 10 11 12 13
|   |   |   |   ...
```

```
v[] = 11 10 9 8 7 6 1 4 13 12 5
introsort(v, 0, 10)
|   maxdepth = 2
|   intro(v, 0, 10, 2) //quick sem mediana
|   |   partition(v, 0, 10) : 4 1 5 8 7 6 10 11 13 12 9
|   |   intro(v, 0, 1, 1) //quick esquerda
|   |   |   //r-l <= 2 : 4 1 5 8 7 6 10 11 13 12 9
|   |   |   return
|   |   intro(v, 2, 10, 1) //quick direita
|   |   |   partition(v, 0, 10) : 4 1 5 8 7 6 9 11 13 12 10
|   |   |   intro(v, 2, 6, 0) //quick esquerda
|   |   |   |   //maxdepth==0
|   |   |   |   heap(v, 2, 10) : 4 1 5 6 7 8 9 11 13 12 10
|   |   |   intro(v, 7, 10, 0) //quick direita
|   |   |   |   //maxdepth==0
|   |   |   |   heap(v, 7, 10) : 4 1 5 6 7 8 9 10 11 12 13
|   insertion(v, 0, 10) : 4 1 5 6 7 8 9 10 11 12 13
```

```
v[] = 11 10 9 8 7 6 1 4 13 12 5
introsort(v, 0, 10)
|   maxdepth = 2
|   intro(v, 0, 10, 2) //quick sem mediana
|   |   partition(v, 0, 10) : 4 1 5 8 7 6 10 11 13 12 9
|   |   intro(v, 0, 1, 1) //quick esquerda
|   |   |   //r-l <= 2 : 4 1 5 8 7 6 10 11 13 12 9
|   |   |   return
|   |   intro(v, 2, 10, 1) //quick direita
|   |   |   partition(v, 0, 10) : 4 1 5 8 7 6 9 11 13 12 10
|   |   |   intro(v, 2, 6, 0) //quick esquerda
|   |   |   |   //maxdepth==0
|   |   |   |   heap(v, 2, 10) : 4 1 5 6 7 8 9 11 13 12 10
|   |   |   intro(v, 7, 10, 0) //quick direita
|   |   |   |   //maxdepth==0
|   |   |   |   heap(v, 7, 10) : 4 1 5 6 7 8 9 10 11 12 13
|   insertion(v, 0, 10) : 4 1 5 6 7 8 9 10 11 12 13
|   |                       1 4 5 6 7 8 9 10 11 12 13
```

```
v[] = 11 10 9 8 7 6 1 4 13 12 5
introsort(v, 0, 10)
|   maxdepth = 2
|   intro(v, 0, 10, 2) //quick sem mediana
|   |   partition(v, 0, 10) : 4 1 5 8 7 6 10 11 13 12 9
|   |   intro(v, 0, 1, 1) //quick esquerda
|   |   |   //r-l <= 2 : 4 1 5 8 7 6 10 11 13 12 9
|   |   |   return
|   |   intro(v, 2, 10, 1) //quick direita
|   |   |   partition(v, 0, 10) : 4 1 5 8 7 6 9 11 13 12 10
|   |   |   intro(v, 2, 6, 0) //quick esquerda
|   |   |   |   //maxdepth==0
|   |   |   |   heap(v, 2, 10) : 4 1 5 6 7 8 9 11 13 12 10
|   |   |   intro(v, 7, 10, 0) //quick direita
|   |   |   |   //maxdepth==0
|   |   |   |   heap(v, 7, 10) : 4 1 5 6 7 8 9 10 11 12 13
|   insertion(v, 0, 10) : 4 1 5 6 7 8 9 10 11 12 13
|   |                       1 4 5 6 7 8 9 10 11 12 13
|   |                       1 4 5 6 7 8 9 10 11 12 13
```

```
v[] = 11 10 9 8 7 6 1 4 13 12 5
introsort(v, 0, 10)
|   maxdepth = 2
|   intro(v, 0, 10, 2) //quick sem mediana
|   |   partition(v, 0, 10) : 4 1 5 8 7 6 10 11 13 12 9
|   |   intro(v, 0, 1, 1) //quick esquerda
|   |   |   //r-l <= 2 : 4 1 5 8 7 6 10 11 13 12 9
|   |   |   return
|   |   intro(v, 2, 10, 1) //quick direita
|   |   |   partition(v, 0, 10) : 4 1 5 8 7 6 9 11 13 12 10
|   |   |   intro(v, 2, 6, 0) //quick esquerda
|   |   |   |   //maxdepth==0
|   |   |   |   heap(v, 2, 10) : 4 1 5 6 7 8 9 11 13 12 10
|   |   |   intro(v, 7, 10, 0) //quick direita
|   |   |   |   //maxdepth==0
|   |   |   |   heap(v, 7, 10) : 4 1 5 6 7 8 9 10 11 12 13
|   insertion(v, 0, 10) : 4 1 5 6 7 8 9 10 11 12 13
|   |                       1 4 5 6 7 8 9 10 11 12 13
|   |                       1 4 5 6 7 8 9 10 11 12 13
|   |                       1 4 5 6 7 8 9 10 11 12 13
|   |                       ...
```

```
v[] = 11 10 9 8 7 6 1 4 13 12 5
introsort(v, 0, 10)
|   maxdepth = 2
|   intro(v, 0, 10, 2) //quick sem mediana
|   |   partition(v, 0, 10) : 4 1 5 8 7 6 10 11 13 12 9
|   |   intro(v, 0, 1, 1) //quick esquerda
|   |   |   //r-l <= 2 : 4 1 5 8 7 6 10 11 13 12 9
|   |   |   return
|   |   intro(v, 2, 10, 1) //quick direita
|   |   |   partition(v, 0, 10) : 4 1 5 8 7 6 9 11 13 12 10
|   |   |   intro(v, 2, 6, 0) //quick esquerda
|   |   |   |   //maxdepth==0
|   |   |   |   heap(v, 2, 10) : 4 1 5 6 7 8 9 11 13 12 10
|   |   |   intro(v, 7, 10, 0) //quick direita
|   |   |   |   //maxdepth==0
|   |   |   |   heap(v, 7, 10) : 4 1 5 6 7 8 9 10 11 12 13
|   insertion(v, 0, 10) : 4 1 5 6 7 8 9 10 11 12 13
|   |                       1 4 5 6 7 8 9 10 11 12 13
|   |                       1 4 5 6 7 8 9 10 11 12 13
|   |                       1 4 5 6 7 8 9 10 11 12 13
|   |                       ...
|   |                       1 4 5 6 7 8 9 10 11 12 13
```

```
v[] = 11 10 9 8 7 6 1 4 13 12 5
introsort(v, 0, 10)
|   maxdepth = 2
|   intro(v, 0, 10, 2) //quick sem mediana
|   |   partition(v, 0, 10) : 4 1 5 8 7 6 10 11 13 12 9
|   |   intro(v, 0, 1, 1) //quick esquerda
|   |   |   //r-l <= 2 : 4 1 5 8 7 6 10 11 13 12 9
|   |   |   return
|   |   intro(v, 2, 10, 1) //quick direita
|   |   |   partition(v, 0, 10) : 4 1 5 8 7 6 9 11 13 12 10
|   |   |   intro(v, 2, 6, 0) //quick esquerda
|   |   |   |   //maxdepth==0
|   |   |   |   heap(v, 2, 10) : 4 1 5 6 7 8 9 11 13 12 10
|   |   |   intro(v, 7, 10, 0) //quick direita
|   |   |   |   //maxdepth==0
|   |   |   |   heap(v, 7, 10) : 4 1 5 6 7 8 9 10 11 12 13
|   insertion(v, 0, 10) : 4 1 5 6 7 8 9 10 11 12 13
|   |                       1 4 5 6 7 8 9 10 11 12 13
|   |                       1 4 5 6 7 8 9 10 11 12 13
|   |                       1 4 5 6 7 8 9 10 11 12 13
|   |                       ...
|   |                       1 4 5 6 7 8 9 10 11 12 13
|   |                       1 4 5 6 7 8 9 10 11 12 13
```

Algoritmos de Ordenação Eficientes - Intro Sort

```
1  //Complexidade: ≈ quick(n) + heap(m) + insertion(k)
2  //              ≈ O(n*logn) + O(m*logm) + O(k*n) = O(n*logn)
3  void intro(int *v,  int l, int r, int maxdepth) {
4    if(r-l<=15) return;
5    if(maxdepth == 0) heap_sort(v, l, r);
6    else {
7      compexch(v[l], v[(l+r)/2]);
8      compexch(v[l], v[r]);
9      compexch(v[r], v[(l+r)/2]);
10
11     int p = partition(v, l, r);
12     intro(v, l, p-1, maxdepth-1);
13     intro(v, p+1, r, maxdepth-1);
14   }
15 }
16 void intro_sort(int *v,  int l, int r) {
17     int maxdepth = ((int)log2((double)(r-l+1)));
18
19     intro(v, l, r, maxdepth);
20     insertion_sort(v, l, r);
21 }
```

Roteiro

1. Algoritmos de Ordenação Eficientes
   - Heap Sort
   - Intro Sort
   - Counting Sort
   - Radix Sort

Algoritmos de Ordenação Eficientes

- Linearítmicos
  - $O(n \log n)$
  - Ordenação pela comparação do valor das chaves
- Lineares
  - $O(n)$
  - Ordenação pela comparação das partes que formam as chaves

Algoritmos de Ordenação Eficientes - Counting Sort

- Ordenação por contagem da frequência das chaves
- Etapas:
  1. Contar as frequências de cada chave
  2. Calcular as posições através das frequências
  3. Distribuir as chaves

Counting Sort - Etapa 1 - Contar as frequências

- Para chaves com valores de 0 até $R-1$
- Utiliza-se: `count[R+1]`
- Para cada chave i
  - `count[i+1]` = frequência da chave i
- Para cada chave
  - `count[0]` = posição inicial
  - `count[1]` = frequência da chave 0
  - `count[2]` = frequência da chave 1
  - `count[R]` = frequência da chave $R-1$

Counting Sort - Etapa 1 - Contar as frequências

- R = 5 (intervalo 0 - 4)

```
              0   1   2   3   4
    v[5] [ 2   3   3   4   1 ]

              0   1   2   3   4   5
count[R+1] [ 0   0   0   0   0   0 ]
```

Counting Sort - Etapa 1 - Contar as frequências

- R = 5 (intervalo 0 - 4)

```
                 0   1   2   3   4
    v[0]     [ 2   3   3   4   1 ]

                 0   1   2   3   4   5
count[v[0]+1] [ 0   0   0   0+1 0   0 ]
```

Counting Sort - Etapa 1 - Contar as frequências

- R = 5 (intervalo 0 - 4)

```
                 0   1   2   3   4
      v[1]    [ 2   3   3   4   1 ]

                 0   1   2   3   4   5
count[v[1]+1] [ 0   0   0   1   0+1 0 ]
```

Counting Sort - Etapa 1 - Contar as frequências

- R = 5 (intervalo 0 - 4)

```
                0   1   2   3   4
     v[2]    [ 2   3   3   4   1 ]

                0   1   2   3   4   5
count[v[2]+1] [ 0   0   0   1   1+1 0 ]
```

Counting Sort - Etapa 1 - Contar as frequências

- R = 5 (intervalo 0 - 4)

```
                 0   1   2   3   4
    v[3]     [ 2   3   3   4   1 ]

                 0   1   2   3   4   5
count[v[3]+1] [ 0   0   0   1   2   0+1 ]
```

Counting Sort - Etapa 1 - Contar as frequências

- R = 5 (intervalo 0 - 4)

```
                 0   1   2   3   4
     v[4]    [ 2   3   3   4   1 ]

                 0   1   2   3   4   5
count[v[4]+1] [ 0   0   0+1 1   2   1 ]
```

Counting Sort - Etapa 1 - Contar as frequências

- R = 5 (intervalo 0 - 4)

```
              0   1   2   3   4
    v[5] [ 2   3   3   4   1 ]

              0   1   2   3   4   5
count[5+1] [ 0   0   1   1   2   1 ]
```

Counting Sort - Etapa 1 - Contar as frequências

```
1   //memset(count, 0, sizeof(int)*(R+1))
2   for(i=0; i<=R; i++)
3   {
4      count[i] = 0;
5   }
6
7   for(i=l; i<=r; i++)
8   {
9      count[v[i]+1] = count[v[i]+1]+1;
10     //count[v[i]+1]++;
11  }
```

Counting Sort - Etapa 2 - Calcular as posições pela frequência

- Vetor de frequências:
  - `count[i]` = quantidade de chaves `i-1`
- Calcular a posição de `i`
  - Quantidade de chaves menores que `i`
  - `count[i] = count[i] + count[i-1] + ... + count[0]`
  - `count[i]` = distância de 0 até a chave `i` (posição de `i`)

Counting Sort - Etapa 2 - Calcular as posições

- R = 5 (intervalo 0 - 4)

```
1       count[k] = (<= k-1)
```

Counting Sort - Etapa 2 - Calcular as posições

- R = 5 (intervalo 0 - 4)

```
1       count[k] = (<= k-1)
2       count[k] = (< k-1) + (= k-1)
```

Counting Sort - Etapa 2 - Calcular as posições

- R = 5 (intervalo 0 - 4)

```
1       count[k] = (<= k-1)
2       count[k] = (< k-1) + (= k-1)
3       count[k] = count[k-1] + count[k]
```

Counting Sort - Etapa 2 - Calcular as posições

- R = 5 (intervalo 0 - 4)

```
1       count[k] = (<= k-1)
2       count[k] = (< k-1) + (= k-1)
3       count[k] = count[k-1] + count[k]
```

```
Posição inicial da primeira chave 0

chave          0   1   2   3   4   5
               i  =0  =1  =2  =3  =4
count       [ 0   0   1   1   2   1 ] posição inicial
```

Counting Sort - Etapa 2 - Calcular as posições

- R = 5 (intervalo 0 - 4)

```
1       count[k] = (<= k-1)
2       count[k] = (< k-1) + (= k-1)
3       count[k] = count[k-1] + count[k]
```

```
Posição inicial da primeira chave 1

chave        0   1   2   3   4   5
             i  =0  =1  =2  =3  =4
count     [ 0   0   1   1   2   1 ] chaves <=0
             |_+_|
```

Counting Sort - Etapa 2 - Calcular as posições

- R = 5 (intervalo 0 - 4)

```
1       count[k] = (<= k-1)
2       count[k] = (< k-1) + (= k-1)
3       count[k] = count[k-1] + count[k]
```

```
Posição inicial da primeira chave 1

chave        0   1   2   3   4   5
             i <=0  =1  =2  =3  =4
count     [ 0   0   1   1   2   1 ] chaves <=0
              |
```

Counting Sort - Etapa 2 - Calcular as posições

- R = 5 (intervalo 0 - 4)

```
1       count[k] = (<= k-1)
2       count[k] = (< k-1) + (= k-1)
3       count[k] = count[k-1] + count[k]
```

```
Posição inicial da primeira chave 2

chave        0   1   2   3   4   5
             i <=0  =1  =2  =3  =4
count     [ 0   0   1   1   2   1 ] chaves <=1
                |_+_|
```

Counting Sort - Etapa 2 - Calcular as posições

- R = 5 (intervalo 0 - 4)

```
1       count[k] = (<= k-1)
2       count[k] = (< k-1) + (= k-1)
3       count[k] = count[k-1] + count[k]
```

```
Posição inicial da primeira chave 2

chave          0   1   2   3   4   5
               i <=0 <=1  =2  =3  =4
count       [ 0   0   1   1   2   1 ] chaves <=1
                      |
```

Counting Sort - Etapa 2 - Calcular as posições

- R = 5 (intervalo 0 - 4)

```
1       count[k] = (<= k-1)
2       count[k] = (< k-1) + (= k-1)
3       count[k] = count[k-1] + count[k]
```

```
Posição inicial da primeira chave 3

chave        0   1   2   3   4   5
             i <=0 <=1  =2  =3  =4
count     [ 0   0   1   1   2   1 ] chaves <=2
                    |_+_|
```

Counting Sort - Etapa 2 - Calcular as posições

- R = 5 (intervalo 0 - 4)

```
1       count[k] = (<= k-1)
2       count[k] = (< k-1) + (= k-1)
3       count[k] = count[k-1] + count[k]
```

```
Posição inicial da primeira chave 3

chave        0   1   2   3   4   5
             i <=0 <=1 <=2  =3  =4
count     [ 0   0   1   2   2   1 ] chaves <=2
                      |
```

Counting Sort - Etapa 2 - Calcular as posições

- R = 5 (intervalo 0 - 4)

```
1       count[k] = (<= k-1)
2       count[k] = (< k-1) + (= k-1)
3       count[k] = count[k-1] + count[k]
```

```
Posição inicial da primeira chave 4

chave        0   1   2   3   4   5
             i <=0 <=1 <=2  =3  =4
count     [ 0   0   1   2   2   1 ] chaves <=3
                      |_+_|
```

Counting Sort - Etapa 2 - Calcular as posições

- R = 5 (intervalo 0 - 4)

```
1       count[k] = (<= k-1)
2       count[k] = (< k-1) + (= k-1)
3       count[k] = count[k-1] + count[k]
```

```
Posição inicial da primeira chave 4

chave        0   1   2   3   4   5
             i <=0 <=1 <=2 <=3  =4
count     [ 0   0   1   2   4   1 ] chaves <=3
                                |
```

Counting Sort - Etapa 2 - Calcular as posições

- R = 5 (intervalo 0 - 4)

```
1       count[k] = (<= k-1)
2       count[k] = (< k-1) + (= k-1)
3       count[k] = count[k-1] + count[k]
```

```
Posição inicial da primeira chave 5

chave        0   1   2   3   4   5
             i <=0 <=1 <=2 <=3  =4
count     [ 0   0   1   2   4   1 ] chaves <=4
                                |_+_|
```

Counting Sort - Etapa 2 - Calcular as posições

- R = 5 (intervalo 0 - 4)

```
1       count[k] = (<= k-1)
2       count[k] = (< k-1) + (= k-1)
3       count[k] = count[k-1] + count[k]
```

```
Posição inicial da primeira chave 5

chave        0   1   2   3   4   5
             i <=0 <=1 <=2 <=3 <=4
count      [ 0   0   1   2   4   5 ] chaves <=4
                                 |
```

Counting Sort - Etapa 2 - Calcular as posições

```
1  //memset(count, 0, sizeof(int)*(R+1))
2  for(i=0; i<=R; i++) count[i] = 0;
3
4  for(i=l; i<=r; i++) count[v[i]+1]++;
5
6  for(i=1; i<R; i++)
7  {
8     count[i] = count[i] + count[i-1];
9  }
```

Counting Sort - Etapa 3 - Distribuir os elementos

- `count[R+1]`: tabela de índices/posições
- `v[]`: vetor de chaves
- `aux[]`: auxiliar para ordenação
  - `v[i]`: chave
  - `count[ v[i] ]`: posição da chave
  - `aux[ count[v[i]] ] = v[i]`: ordena
  - `count[ v[i] ]++`: próxima posição (chave igual)

Counting Sort - Etapa 3 - Distribuir os elementos

- R = 5 (intervalo 0 - 4)
- posição: `count[v[i]]`
- ordena: `aux[count[v[i]]] = v[i]`
- próxima: `count[v[i]]++`

```
          i     0   1   2   3   4
       v[0]   [ 2   3   3   4   1 ]

              0   1   2   3   4   5
count[v[0]]   [ 0   0   1   2   4   5 ]

              0   1   2   3   4
aux[count[v[0]]]  [                 ]
```

Counting Sort - Etapa 3 - Distribuir os elementos

- R = 5 (intervalo 0 - 4)
- posição: `count[v[i]]`
- ordena: `aux[count[v[i]]] = v[i]`
- próxima: `count[v[i]]++`

```
              i     0   1   2   3   4
         v[0]   [ 2   3   3   4   1 ]

                    0   1   2   3   4   5
  count[v[0]]   [ 0   0   1   2   4   5 ]

                    0   1   2   3   4
aux[count[v[0]]]  [     2               ]
```

Counting Sort - Etapa 3 - Distribuir os elementos

- R = 5 (intervalo 0 - 4)
- posição: `count[v[i]]`
- ordena: `aux[count[v[i]]] = v[i]`
- próxima: `count[v[i]]++`

```
              i     0   1   2   3   4
          v[0]   [ 2   3   3   4   1 ]

                    0   1   2   3   4   5
    count[v[0]]   [ 0   0   1+1 2   4   5 ]

                    0   1   2   3   4
aux[count[v[0]]]  [     2                 ]
```

Counting Sort - Etapa 3 - Distribuir os elementos

- R = 5 (intervalo 0 - 4)
- posição: `count[v[i]]`
- ordena: `aux[count[v[i]]] = v[i]`
- próxima: `count[v[i]]++`

```
              i     0   1   2   3   4
          v[1]  [ 2   3   3   4   1 ]

                    0   1   2   3   4   5
    count[v[1]]   [ 0   0   2   2+1 4   5 ]

                    0   1   2   3   4
aux[count[v[1]]]  [     2   3           ]
```

Counting Sort - Etapa 3 - Distribuir os elementos

- R = 5 (intervalo 0 - 4)
- posição: `count[v[i]]`
- ordena: `aux[count[v[i]]] = v[i]`
- próxima: `count[v[i]]++`

```
                i     0   1   2   3   4
             v[2]   [ 2   3   3   4   1 ]

                     0   1   2   3   4   5
    count[v[2]]   [  0   0   2   3+1 4   5 ]

                     0   1   2   3   4
aux[count[v[2]]]  [      2   3   3       ]
```

Counting Sort - Etapa 3 - Distribuir os elementos

- R = 5 (intervalo 0 - 4)
- posição: `count[v[i]]`
- ordena: `aux[count[v[i]]] = v[i]`
- próxima: `count[v[i]]++`

```
          i      0   1   2   3   4
       v[3]   [ 2   3   3   4   1 ]

                 0   1   2   3   4   5
 count[v[3]]   [ 0   0   2   4   4+1 5 ]

                 0   1   2   3   4
aux[count[v[3]]] [     2   3   3   4 ]
```

Counting Sort - Etapa 3 - Distribuir os elementos

- R = 5 (intervalo 0 - 4)
- posição: `count[v[i]]`
- ordena: `aux[count[v[i]]] = v[i]`
- próxima: `count[v[i]]++`

i	0	1	2	3	4	
v[4]	2	3	3	4	1	

	0	1	2	3	4	5
count[v[4]]	0	0+1	2	4	5	5

	0	1	2	3	4
aux[count[v[4]]]	1	2	3	3	4

Counting Sort - Etapa 3 - Distribuir os elementos

- R = 5 (intervalo 0 - 4)
- posição: `count[v[i]]`
- ordena: `aux[count[v[i]]] = v[i]`
- próxima: `count[v[i]]++`

```
    i     0   1   2   3   4
    v  [ 2   3   3   4   1 ]

          0   1   2   3   4   5
count  [ 0   1   2   4   5   5 ]

          0   1   2   3   4
  aux  [ 1   2   3   3   4 ]
```

Counting Sort - Etapa 3 - Distribuir os elementos

- R = 5 (intervalo 0 - 4)
- posição: `count[v[i]]`
- ordena: `aux[count[v[i]]] = v[i]`
- próxima: `count[v[i]]++`

```
          0   1   2   3   4
aux   [ 1   2   3   3   4 ]
v     [ 1   2   3   3   4 ]
```

```
1 #define MAX 10000
2 #define R 5
3
4 int aux[MAX];
5 void countingsort(int *v, int l, int r) {
6   int i, count[R+1];
```

```c
#define MAX 10000
#define R 5

int aux[MAX];
void countingsort(int *v, int l, int r) {
  int i, count[R+1];

  //zerando
  for(i = 0; i <= R; i++) count[i] = 0;
```

```
1  #define MAX 10000
2  #define R 5
3
4  int aux[MAX];
5  void countingsort(int *v, int l, int r) {
6    int i, count[R+1];
7
8    //zerando
9    for(i = 0; i <= R; i++) count[i] = 0;
10
11   //frequencias
12   for(i = l; i <= r; i++) count[v[i] + 1]++;
```

```c
#define MAX 10000
#define R 5

int aux[MAX];
void countingsort(int *v, int l, int r) {
  int i, count[R+1];

  //zerando
  for(i = 0; i <= R; i++) count[i] = 0;

  //frequencias
  for(i = l; i <= r; i++) count[v[i] + 1]++;

  //posições
  for(i = 1; i <= R; i++) count[i] += count[i-1];
```

```c
#define MAX 10000
#define R 5

int aux[MAX];
void countingsort(int *v, int l, int r) {
  int i, count[R+1];

  //zerando
  for(i = 0; i <= R; i++) count[i] = 0;

  //frequencias
  for(i = l; i <= r; i++) count[v[i] + 1]++;

  //posições
  for(i = 1; i <= R; i++) count[i] += count[i-1];

  //distribuição
  for(i = l; i <= r; i++) aux[count[v[i]]++] = v[i];
```

```c
#define MAX 10000
#define R 5

int aux[MAX];
void countingsort(int *v, int l, int r) {
  int i, count[R+1];

  //zerando
  for(i = 0; i <= R; i++) count[i] = 0;

  //frequencias
  for(i = l; i <= r; i++) count[v[i] + 1]++;

  //posições
  for(i = 1; i <= R; i++) count[i] += count[i-1];

  //distribuição
  for(i = l; i <= r; i++) aux[count[v[i]]++] = v[i];

  //cópia : a partir de aux[0]; ex.: i=l=3, i-l=0
  for (i = l; i <= r; i++) v[i] = aux[i-l];
}
```

```c
#define MAX 10000
#define R 5

typedef struct{ int chave; char nome[20]; } Item;
Item aux[MAX];

void countingsort(Item *v, int l, int r) {
  int i; count[R+1];

  for(i = 0; i <= R; i++) count[i] = 0;
  for(i = l; i <= r; i++) count[v[i].chave + 1]++;
  for(i = 1; i <= R; i++) count[i] += count[i-1];

  //distribuição dos itens
  for(i = l; i <= r; i++)
    aux[count[v[i].chave]++] = v[i];

  for (i = l; i <= r; i++) v[i] = aux[i-l];
}
```

Algoritmos de Ordenação Eficientes - Counting Sort

- Complexidade no pior caso: $O(N + R)$
  - N: número de chaves
  - R: intervalo das chaves
- In-place: não
- Estável
- Adaptatividade:
  - Contar: custo linear em N
  - Posição: custo linear em R
  - Ordenar: custo linear em N
  - Não é adaptativo

Roteiro

1. Algoritmos de Ordenação Eficientes
   - Heap Sort
   - Intro Sort
   - Counting Sort
   - Radix Sort

Algoritmos de Ordenação Eficientes - Radix Sort

- Ordenar pela a raiz(radix) da representação dos dados
- Extraindo o i-ésimo digito da chave

Algoritmos de Ordenação Eficientes - Radix Sort

Ordenação: compara-se as chaves/dados

- Comparar a estrutura das chaves
- Decompondo a chave em subestruturas que a compõe:
  - Números: unidades, dezenas, centenas...
  - Palavras: letras
- A cada iteração/recursão,
  - Comparar somente parte da chave
  - Ordenando parcialmente
  - Usa-se *counting sort (key-indexed counting)* byte a byte

Algoritmos de Ordenação Eficientes - Radix Sort

17**0** 04**5** 07**5** 09**0** 80**2** 02**4** 00**2** 06**6**

Algoritmos de Ordenação Eficientes - Radix Sort

17**0** 04**5** 07**5** 09**0** 80**2** 02**4** 00**2** 06**6**

17**0** 09**0** 80**2** 00**2** 02**4** 04**5** 07**5** 06**6**

Algoritmos de Ordenação Eficientes - Radix Sort

1**7**0 0**9**0 8**0**2 0**0**2 0**2**4 0**4**5 0**7**5 0**6**6

Algoritmos de Ordenação Eficientes - Radix Sort

1**7**0 0**9**0 8**0**2 0**0**2 0**2**4 0**4**5 0**7**5 0**6**6

8**0**2 0**0**2 0**2**4 0**4**5 0**6**6 1**7**0 0**7**5 0**9**0

Algoritmos de Ordenação Eficientes - Radix Sort

802 **0**02 **0**24 **0**45 **0**66 **1**70 **0**75 **0**90

Algoritmos de Ordenação Eficientes - Radix Sort

802 002 024 045 066 170 075 090

002 024 045 066 075 090 170 802

Algoritmos de Ordenação Eficientes - Radix Sort

170 045 075 090 802 024 002 066

$\Downarrow$

002 024 045 066 075 090 170 802

- Ordena pela unidade, dezena, centena, etc.
- Problema?

Algoritmos de Ordenação Eficientes - Radix Sort

- Envolve operações custosas

Algoritmos de Ordenação Eficientes - Radix Sort

- Envolve operações custosas
  - 802: 802%10, (802/10)%10, (802/100)%10

Algoritmos de Ordenação Eficientes - Radix Sort

- Envolve operações custosas
  - 802: 802%10, (802/10)%10, (802/100)%10
- Quantidade de unidades impacta no custo

Algoritmos de Ordenação Eficientes - Radix Sort

- Envolve operações custosas
  - 802: 802%10, (802/10)%10, (802/100)%10
- Quantidade de unidades impacta no custo
- Alternativa: decomposição por bytes

Algoritmos de Ordenação Eficientes - Radix Sort

- Envolve operações custosas
  - 802: 802%10, (802/10)%10, (802/100)%10
- Quantidade de unidades impacta no custo
- Alternativa: decomposição por bytes
- Exemplo com inteiros: partes de 1 byte

Algoritmos de Ordenação Eficientes - Radix Sort

- Envolve operações custosas
  - 802: 802%10, (802/10)%10, (802/100)%10
- Quantidade de unidades impacta no custo
- Alternativa: decomposição por bytes
- Exemplo com inteiros: partes de 1 byte
  - 510 = 00000000 00000000 00000001 11111110

Algoritmos de Ordenação Eficientes - Radix Sort

- Envolve operações custosas
  - 802: 802%10, (802/10)%10, (802/100)%10
- Quantidade de unidades impacta no custo
- Alternativa: decomposição por bytes
- Exemplo com inteiros: partes de 1 byte
  - $510 = 00000000\ 00000000\ 00000001\ 11111110$
  - $510 = 1 * 2^8 + 1 * 2^7 + 1 * 2^6 + 1 * 2^5 + 1 * 2^4 + 1 * 2^3 + 1 * 2^2 + 1 * 2^1 + 0 * 2^0$

Algoritmos de Ordenação Eficientes - Radix Sort

- Decomposição dos números
  - 510 = 256+254
  - 256 = 00000000 00000000 0000000**1** 00000000
  - 254 = 00000000 00000000 00000000 **11111110**
  - 510 = 00000000 00000000 0000000**1** **11111110**
- Comparação por bytes
  - 510 = 00000000 00000000 00000001 11111110
  - 767 = 00000000 00000000 00000010 11111111
  - 258 = 00000000 00000000 00000001 00000010
  - 255 = 00000000 00000000 00000000 11111111

Algoritmos de Ordenação Eficientes - Radix Sort

- Comparação por bytes
  - 1o byte
  - 510 $\rightarrow$ 00000000 00000000 00000001 11111110= 254
  - 767 $\rightarrow$ 00000000 00000000 00000010 11111111= 255
  - 258 $\rightarrow$ 00000000 00000000 00000001 00000010= 2
  - 255 $\rightarrow$ 00000000 00000000 00000000 11111111= 255

Algoritmos de Ordenação Eficientes - Radix Sort

- Comparação por bytes
  - 1o byte
  - 258 $\rightarrow$ 00000000 00000000 00000001 00000010$=$ 2
  - 510 $\rightarrow$ 00000000 00000000 00000001 11111110$=$ 254
  - 767 $\rightarrow$ 00000000 00000000 00000010 11111111$=$ 255
  - 255 $\rightarrow$ 00000000 00000000 00000000 11111111$=$ 255

Algoritmos de Ordenação Eficientes - Radix Sort

- Comparação por bytes
  - 2o byte
  - 258 → 00000000 00000000 00000001 00000010= 1 (256)
  - 510 → 00000000 00000000 00000001 11111110= 1 (256)
  - 767 → 00000000 00000000 00000010 11111111= 2 (512)
  - 255 → 00000000 00000000 00000000 11111111= 0 (0)

Algoritmos de Ordenação Eficientes - Radix Sort

- Comparação por bytes
  - 2o byte
  - 255 → 00000000 00000000 00000000 11111111= 0 (0)
  - 258 → 00000000 00000000 00000001 00000010= 1 (256)
  - 510 → 00000000 00000000 00000001 11111110= 1 (256)
  - 767 → 00000000 00000000 00000010 11111111= 2 (512)

Algoritmos de Ordenação Eficientes - Radix Sort

- Comparação por bytes
  - 3o e 4o byte
  - 255 $\rightarrow$ 00000000 00000000 00000000 11111111 = 0
  - 258 $\rightarrow$ 00000000 00000000 00000001 00000010 = 0
  - 510 $\rightarrow$ 00000000 00000000 00000001 11111110 = 0
  - 767 $\rightarrow$ 00000000 00000000 00000010 11111111 = 0

Algoritmos de Ordenação Eficientes - Radix Sort

- Comparação por bytes
  - Ordenado
  - 255
  - 258
  - 510
  - 767

Algoritmos de Ordenação Eficientes - Radix Sort

- Separação dos bytes: byte 0
- Operação *E* bit a bit
- 510 & 255

```
                                                |
00000000 00000000 00000001 11111110
00000000 00000000 00000000 11111111 & (E bit a bit)
-----------------------------------
```

Algoritmos de Ordenação Eficientes - Radix Sort

- Separação dos bytes: byte 0
- Operação *E* bit a bit
- 510 & 255

```
                                                |
00000000 00000000 00000001 11111110
00000000 00000000 00000000 11111111 & (E bit a bit)
-----------------------------------
                                                0
```

Algoritmos de Ordenação Eficientes - Radix Sort

- Separação dos bytes: byte 0
- Operação *E* bit a bit
- 510 & 255

```
                                                |
00000000 00000000 00000001 11111110
00000000 00000000 00000000 11111111 & (E bit a bit)
-----------------------------------
                                                10
```

Algoritmos de Ordenação Eficientes - Radix Sort

- Separação dos bytes: byte 0
- Operação *E* bit a bit
- 510 & 255

```
                                          |
00000000 00000000 00000001 11111110
00000000 00000000 00000000 11111111 & (E bit a bit)
-----------------------------------
                                   110
```

Algoritmos de Ordenação Eficientes - Radix Sort

- Separação dos bytes: byte 0
- Operação *E* bit a bit
- 510 & 255

```
                                | |||||
00000000 00000000 00000001 11111110
00000000 00000000 00000000 11111111 & (E bit a bit)
-----------------------------------
                                0 11111110
```

Algoritmos de Ordenação Eficientes - Radix Sort

- Separação dos bytes: byte 0
- Operação *E* bit a bit
- 510 & 255

```
00000000 00000000 00000001 11111110
00000000 00000000 00000000 11111111 & (E bit a bit)
-----------------------------------
00000000 00000000 00000000 11111110
```

Algoritmos de Ordenação Eficientes - Radix Sort

- Separação dos bytes: byte 1
- "Remover o byte 0" = deslocar 1 byte para direita
- Deslocar 1 byte para direita = colocar o byte 1 no byte 0
- `510 >> 1*8 & 255`:
  00000000 00000000 00000001 11111110 » 8

Algoritmos de Ordenação Eficientes - Radix Sort

- Separação dos bytes: byte 1
- "Remover o byte 0" = deslocar 1 byte para direita
- Deslocar 1 byte para direita = colocar o byte 1 no byte 0
- `510 >> 1*8 & 255`:
  00000000 00000000 00000000 11111111 » 8

Algoritmos de Ordenação Eficientes - Radix Sort

- Separação dos bytes: byte 1
- "Remover o byte 0" = deslocar 1 byte para direita
- Deslocar 1 byte para direita = colocar o byte 1 no byte 0
- `510 >> 1*8 & 255`:
  00000000 00000000 00000000 01111111 » 8

Algoritmos de Ordenação Eficientes - Radix Sort

- Separação dos bytes: byte 1
- "Remover o byte 0" = deslocar 1 byte para direita
- Deslocar 1 byte para direita = colocar o byte 1 no byte 0
- `510 >> 1*8 & 255`:
  00000000 00000000 00000000 00111111 » 8

Algoritmos de Ordenação Eficientes - Radix Sort

- Separação dos bytes: byte 1
- "Remover o byte 0" = deslocar 1 byte para direita
- Deslocar 1 byte para direita = colocar o byte 1 no byte 0
- `510 >> 1*8 & 255`:
  00000000 00000000 00000000 00011111 » 8

Algoritmos de Ordenação Eficientes - Radix Sort

- Separação dos bytes: byte 1
- "Remover o byte 0" = deslocar 1 byte para direita
- Deslocar 1 byte para direita = colocar o byte 1 no byte 0
- `510 >> 1*8 & 255`:
  00000000 00000000 00000000 00001111 » 8

Algoritmos de Ordenação Eficientes - Radix Sort

- Separação dos bytes: byte 1
- "Remover o byte 0" = deslocar 1 byte para direita
- Deslocar 1 byte para direita = colocar o byte 1 no byte 0
- `510 >> 1*8 & 255`:
  00000000 00000000 00000000 00000111 » 8

Algoritmos de Ordenação Eficientes - Radix Sort

- Separação dos bytes: byte 1
- "Remover o byte 0" = deslocar 1 byte para direita
- Deslocar 1 byte para direita = colocar o byte 1 no byte 0
- `510 >> 1*8 & 255`:
  00000000 00000000 00000000 00000011 » 8

Algoritmos de Ordenação Eficientes - Radix Sort

- Separação dos bytes: byte 1
- "Remover o byte 0" = deslocar 1 byte para direita
- Deslocar 1 byte para direita = colocar o byte 1 no byte 0
- `510 >> 1*8 & 255`:
  00000000 00000000 00000000 00000001 » 8

Algoritmos de Ordenação Eficientes - Radix Sort

- Separação dos bytes: byte 1
- "Remover o byte 0" = deslocar 1 byte para direita
- Deslocar 1 byte para direita = colocar o byte 1 no byte 0
- `510 >> 1*8 & 255`:

```
                                          |
00000000 00000000 00000000 00000001
00000000 00000000 00000000 11111111 & (E bit a bit)
-----------------------------------
```

Algoritmos de Ordenação Eficientes - Radix Sort

- Separação dos bytes: byte 1
- "Remover o byte 0" = deslocar 1 byte para direita
- Deslocar 1 byte para direita = colocar o byte 1 no byte 0
- `510 >> 1*8 & 255`:

```
                                          |
00000000 00000000 00000000 00000001
00000000 00000000 00000000 11111111 & (E bit a bit)
-----------------------------------
00000000 00000000 00000000 00000001
```

Algoritmos de Ordenação Eficientes - Radix Sort

```
1 //256
2 #define R (1 << 8)
3 /*
4   << shift esquerda
5   00000000 00000000 00000000 00000001 << 8
6   00000000 00000000 00000001 00000000 256
7 */
```

Algoritmos de Ordenação Eficientes - Radix Sort

```
1 //256
2 #define R (1 << 8)
3 /*
4   << shift esquerda
5   00000000 00000000 00000000 00000001 << 8
6   00000000 00000000 00000001 00000000 256
7 */
```

```
1 //bits por byte
2 #define bitsbyte 8
3
4 //digito D do número N
```

Algoritmos de Ordenação Eficientes - Radix Sort

```
1 //256
2 #define R (1 << 8)
3 /*
4   << shift esquerda
5   00000000 00000000 00000000 00000001 << 8
6   00000000 00000000 00000001 00000000 256
7 */
```

```
1 //bits por byte
2 #define bitsbyte 8
3
4 //digito D do número N
5 // (D*bitsbyte)                    número de bits a deslocar
```

Algoritmos de Ordenação Eficientes - Radix Sort

```
1 //256
2 #define R (1 << 8)
3 /*
4   << shift esquerda
5   00000000 00000000 00000000 00000001 << 8
6   00000000 00000000 00000001 00000000 256
7 */
```

```
1 //bits por byte
2 #define bitsbyte 8
3
4 //digito D do número N
5 // (D*bitsbyte)                    número de bits a deslocar
6 // (N >> (D*bitsbyte))             colocar o byte D no byte 0
```

Algoritmos de Ordenação Eficientes - Radix Sort

```
1 //256
2 #define R (1 << 8)
3 /*
4   << shift esquerda
5   00000000 00000000 00000000 00000001 << 8
6   00000000 00000000 00000001 00000000 256
7 */
```

```
1 //bits por byte
2 #define bitsbyte 8
3
4 //digito D do número N
5 // (D*bitsbyte)                    número de bits a deslocar
6 // (N >> (D*bitsbyte))             colocar o byte D no byte 0
7 // (N >> (D*bitsbyte)) & (R-1)     separar o byte 0
```

Algoritmos de Ordenação Eficientes - Radix Sort

```
1 //256
2 #define R (1 << 8)
3 /*
4   << shift esquerda
5   00000000 00000000 00000000 00000001 << 8
6   00000000 00000000 00000001 00000000 256
7 */
```

```
1 //bits por byte
2 #define bitsbyte 8
3
4 //digito D do número N
5 // (D*bitsbyte)                    número de bits a deslocar
6 // (N >> (D*bitsbyte))             colocar o byte D no byte 0
7 // (N >> (D*bitsbyte)) & (R-1)     separar o byte 0
8 #define digit(N,D) (((N) >> ((D)*bitsbyte)) & (R-1))
```

- digit(66304, 1)
- 00000000 00000001 00000011 00000000 >> 1*8

Algoritmos de Ordenação Eficientes - Radix Sort

```
1 //256
2 #define R (1 << 8)
3 /*
4   << shift esquerda
5   00000000 00000000 00000000 00000001 << 8
6   00000000 00000000 00000001 00000000 256
7 */
```

```
1 //bits por byte
2 #define bitsbyte 8
3
4 //digito D do número N
5 // (D*bitsbyte)                    número de bits a deslocar
6 // (N >> (D*bitsbyte))             colocar o byte D no byte 0
7 // (N >> (D*bitsbyte)) & (R-1)     separar o byte 0
8 #define digit(N,D) (((N) >> ((D)*bitsbyte)) & (R-1))
```

- digit(66304, 1)
- 00000000 00000001 00000011 00000000 >> 1*8
- 00000000 00000000 00000001 00000011 & 255

Algoritmos de Ordenação Eficientes - Radix Sort

```
1 //256
2 #define R (1 << 8)
3 /*
4   << shift esquerda
5   00000000 00000000 00000000 00000001 << 8
6   00000000 00000000 00000001 00000000 256
7 */
```

```
1 //bits por byte
2 #define bitsbyte 8
3
4 //digito D do número N
5 // (D*bitsbyte)                    número de bits a deslocar
6 // (N >> (D*bitsbyte))             colocar o byte D no byte 0
7 // (N >> (D*bitsbyte)) & (R-1)     separar o byte 0
8 #define digit(N,D) (((N) >> ((D)*bitsbyte)) & (R-1))
```

- digit(66304, 1)
- 00000000 00000001 00000011 00000000 >> 1*8
- 00000000 00000000 00000001 00000011 & 255
- 00000000 00000000 00000000 00000011

Algoritmos de Ordenação Eficientes - Radix Sort

```
1 #define bytesword 4 //int
2 #define bitsbyte 8
3
4 #define R (1 << bitsbyte)
5 #define digit(N,D) (((N) >> ((D)*bitsbyte)) & (R-1))
```

Radix Sort - Método de classificação: LSD

- A partir dígito menos significativo (least significant digit - LSD)
  - Direita para esquerda
  - Ordena estavelmente chaves de comprimento fixo
    - Tamanho da palavra (word) que representa o dado
    - Tipos de tamanho fixo: 4 bytes p/ int, 8 bytes p/ long
    - Strings com C caracteres: $C * 1$ bytes
- Complexidade é proporcional a $W * N$:
  - N chaves
  - chaves de tamanho W

```
1 void radix_sortLSD(int *v,  int l, int r) {
2     int i, w, k;
3     int aux[r-l+1];
4     int count[R+1]; //257: intervalo de 1 byte + 1
5
6
```

```
1 void radix_sortLSD(int *v,  int l, int r) {
2     int i, w, k;
3     int aux[r-l+1];
4     int count[R+1]; //257: intervalo de 1 byte + 1
5
6
7     //byte w
8     for(w=0; w<bytesword; w++){
```

```
1  void radix_sortLSD(int *v,  int l, int r) {
2      int i, w, k;
3      int aux[r-l+1];
4      int count[R+1]; //257: intervalo de 1 byte + 1
5
6
7      //byte w
8      for(w=0; w<bytesword; w++){
9          //for(i=0; i<=R; i++) count[i] = 0;
10         memset(count, 0, sizeof(int)*(R+1));
11
12
13
14
15
16
17
18
19
20
21
22
23         ...
24     }
25 }
```

```
1  void radix_sortLSD(int *v,  int l, int r) {
2      int i, w, k;
3      int aux[r-l+1], count[R+1];
4
5      //byte w
6      for(w=0; w<bytesword; w++){
7          memset(count, 0, sizeof(int)*(R+1));
8
9          //frequências
10         for(i=l; i<=r; i++) {
```

```c
void radix_sortLSD(int *v,  int l, int r) {
    int i, w, k;
    int aux[r-l+1], count[R+1];

    //byte w
    for(w=0; w<bytesword; w++){
        memset(count, 0, sizeof(int)*(R+1));

        //frequências
        for(i=l; i<=r; i++) {
            //byte w da chave v[i]
            k = digit(v[i], w);
```

```
1  void radix_sortLSD(int *v,  int l, int r) {
2      int i, w, k;
3      int aux[r-l+1], count[R+1];
4
5      //byte w
6      for(w=0; w<bytesword; w++){
7          memset(count, 0, sizeof(int)*(R+1));
8
9          //frequências
10         for(i=l; i<=r; i++) {
11             //byte w da chave v[i]
12             k = digit(v[i], w);
13
14             //frequência de k em k+1
15             count[k+1]++;
16         }
17
18
19
20
21
22
23         ...
24     }
25 }
```

```
1  void radix_sortLSD(int *v,  int l, int r) {
2      int i, w, k;
3      int aux[r-l+1], count[R+1];
4
5      //byte w
6      for(w=0; w<bytesword; w++){
7          memset(count, 0, sizeof(int)*(R+1));
8
9          //frequências
10         for(i=l; i<=r; i++) count[ digit(v[i], w)+1 ]++;
11
12
13
14
15
16
17
18
19
20
21
22
23
24     }
25 }
```

```
1  void radix_sortLSD(int *v,  int l, int r) {
2      int i, w, k;
3      int aux[r-l+1], count[R+1];
4
5      //byte w
6      for(w=0; w<bytesword; w++){
7          memset(count, 0, sizeof(int)*(R+1));
8
9          //frequências
10         for(i=l; i<=r; i++) count[ digit(v[i], w)+1 ]++;
11
12         //posição de k: ≤ k-1
13         for(k=1; k<R; k++) count[k] += count[k-1];
14
15
16
17
18
19
20
21
22
23
24     }
25 }
```

```
1  void radix_sortLSD(int *v,  int l, int r) {
2      int i, w, k;
3      int aux[r-l+1], count[R+1];
4
5      //byte w
6      for(w=0; w<bytesword; w++){
7          memset(count, 0, sizeof(int)*(R+1));
8
9          //frequências
10         for(i=l; i<=r; i++) count[ digit(v[i], w)+1 ]++;
11
12         //posições
13         for(k=1; k<R; k++) count[k] += count[k-1];
14
15         //distruibuição
16         for(i=l; i<=r; i++) {
17             //byte w da chave v[i]
18             k = digit(v[i], w);
19
20             //ordenação em aux
21             aux[ count[k]++ ] = v[i];
22         }
23         ...
24     }
25 }
```

```
1  void radix_sortLSD(int *v,  int l, int r) {
2      int i, w, k;
3      int aux[r-l+1], count[R+1];
4
5      //byte w
6      for(w=0; w<bytesword; w++){
7          memset(count, 0, sizeof(int)*(R+1));
8
9          //frequências
10         for(i=l; i<=r; i++) count[ digit(v[i], w)+1 ]++;
11
12         //posições
13         for(k=1; k<R; k++) count[k] += count[k-1];
14
15         //distruibuição
16         for(i=l; i<=r; i++) aux[ count[digit(v[i], w)]++ ] = v[i];
17
18
19
20
21
22
23
24     }
25 }
```

```
1  void radix_sortLSD(int *v,  int l, int r) {
2      int i, w, k;
3      int aux[r-l+1], count[R+1];
4
5      //byte w
6      for(w=0; w<bytesword; w++){
7          memset(count, 0, sizeof(int)*(R+1));
8
9          //frequências
10         for(i=l; i<=r; i++) count[ digit(v[i], w)+1 ]++;
11
12         //posições
13         for(k=1; k<R; k++) count[k] += count[k-1];
14
15         //distruibuição
16         for(i=l; i<=r; i++) aux[ count[digit(v[i], w)]++ ] = v[i];
17
18         //copiando
19         for(i=l; i<=r; i++)
20         {
21             //i-l: iniciar em 0
22             v[i] = aux[i-l];
23         }
24     }
25 }
```

```
1  void radix_sortLSD(int *v,  int l, int r) {
2      int i, w, k;
3      int aux[r-l+1], count[R+1];
4
5      //byte w
6      for(w=0; w<bytesword; w++){
7          memset(count, 0, sizeof(int)*(R+1));
8
9          //frequências
10         for(i=l; i<=r; i++) count[ digit(v[i], w)+1 ]++;
11
12         //posições
13         for(k=1; k<R; k++) count[k] += count[k-1];
14
15         //distruibuição
16         for(i=l; i<=r; i++) aux[ count[digit(v[i], w)]++ ] = v[i];
17
18         //copiando
19         for(i=l; i<=r; i++) v[i] = aux[i-l];
20
21
22
23
24     }
25 }
```

Radix Sort - Método de classificação: LSD

- Chaves de tamanhos fixo: exemplo, inteiros (4 bytes)
- Compara-se todas as chaves em todas as iterações
- E chaves de tamanho variável? Strings de 1 byte, 3 bytes, 200 bytes...
- aab baab ab
  - Qual o índice mais à esquerda?
  - 2 ou 3 ou 1?
- Solução: começar pelo mais significativo

Radix Sort - Método de classificação: MSD

- A partir dígito mais significativo (most significant digit - MSD)
  - Esquerda para direita
  - Ordenação de chaves com tamanhos variáveis
    - Strings: conjunto de palavras de vários tamanhos
  - Ordena-se por **subconjuntos**
    - 1o subconjunto: conjunto total
- Complexidade é proporcional a $W * N$:
  - N chaves
  - chaves de tamanho W

Radix Sort - Método de classificação: MSD

[aab bba aaa baaa]

Radix Sort - Método de classificação: MSD

[aab aaa bba baaa]

Radix Sort - Método de classificação: MSD

[aab aaa] [bba baaa]

Radix Sort - Método de classificação: MSD

[aab aaa] [bba baaa]

Radix Sort - Método de classificação: MSD

[aab aaa] [bba baaa]

Radix Sort - Método de classificação: MSD

[aab aaa] [bba baaa]

Radix Sort - Método de classificação: MSD

[aaa aab] [bba baaa]

Radix Sort - Método de classificação: MSD

[aaa] [aab] [bba baaa]

Radix Sort - Método de classificação: MSD

[aaa] [aab] [bba baaa]

Radix Sort - Método de classificação: MSD

[aaa] [aab] [baaa bba]

Radix Sort - Método de classificação: MSD

[aaa] [aab] [baaa] [bba]

Radix Sort - Método de classificação: MSD

- Strings terminados com '\0'
- Para chaves com valores de 1 até $R - 1$ (R=256)
- Utiliza-se: `count[R+1]`
- Para cada chave *i*
  - Contagem de frequência de `i`: `count[i+1]`
  - Cálculo da posição de `i`: `count[i] = count[i]+count[i-1]`
  - Distribuição de `i`: `count[i]`
  - Subconjunto de `i`: `count[i-1]` até `count[i]-1`
    - `count[i-1]` termina com a primeira posição de `i`
    - `count[i]` termina com a quantidade de `i`'s (e a primeira posição de `i+1`)
    - `count[i]-1` última posição de `i`

Radix Sort - MSD - Etapa 1 - Contar as frequências

- R = 256
- Frequência de `k`: `count[k+1]++`

```
   k           0    1    2    3    4    5
   v[6][5] [ aab  aa   bbaa aabb baa  bc ]

                            'a'
               0   1   2  ... 97 ... 258
count[R+1] [ 0   0   0  ... 0  ... 0   ]
```

Radix Sort - MSD - Etapa 1 - Contar as frequências

- R = 256
- Frequência de k: count[k+1]++

```
    k           0    1    2    3    4    5
    v[0][0] [ aab  aa   bbaa aabb baa  bc ]

                ...  a    b    c    d    e    f    ...
                          =a
count[R+1] [...  0    1    0    0    0    0    ...]
```

Radix Sort - MSD - Etapa 1 - Contar as frequências

- R = 256
- Frequência de k: count[k+1]++

```
    k           0    1    2    3    4    5
    v[1][0] [ aab  aa   bbaa aabb baa  bc ]

                ...  a    b    c    d    e    f   ...
                          =a
count[R+1] [...  0    2    0    0    0    0   ...]
```

Radix Sort - MSD - Etapa 1 - Contar as frequências

- R = 256
- Frequência de k: count[k+1]++

```
    k            0    1    2    3    4    5
    v[2][0] [ aab  aa   bbaa aabb baa  bc ]

                ...  a    b    c    d    e    f   ...
                                =b
count[R+1] [...  0    2    1    0    0    0   ...]
```

Radix Sort - MSD - Etapa 1 - Contar as frequências

- R = 256
- Frequência de k: count[k+1]++

```
    k            0    1    2    3    4    5
    v[3][0] [ aab  aa   bbaa aabb baa  bc ]

                ...  a    b    c    d    e    f   ...
                          =a
count[R+1] [...  0    3    1    0    0    0   ...]
```

Radix Sort - MSD - Etapa 1 - Contar as frequências

- R = 256
- Frequência de k: count[k+1]++

```
    k            0    1    2    3    4    5
    v[4][0] [ aab  aa   bbaa aabb baa  bc ]

                ...  a    b    c    d    e    f   ...
                                =b
count[R+1] [...  0    3    2    0    0    0   ...]
```

Radix Sort - MSD - Etapa 1 - Contar as frequências

- R = 256
- Frequência de k: count[k+1]++

```
    k            0    1    2    3    4    5
    v[5][0]  [ aab  aa   bbaa aabb baa  bc ]

                ...  a    b    c    d    e    f   ...
                                =b
count[R+1]  [...  0    3    3    0    0    0   ...]
```

```
1  typedef char Item;
2  #define maxstring 101
3  #define bitsbyte 8
4  #define R (1 << bitsbyte)
5
6  //Strings: ordena para o d-ésimo caractere
7  void radixMSD(char a[][maxstring], int l, int r, int d) {
8
9      int count[R+1];
10     memset(count, 0, sizeof(int)*(R+1));
11
12     //frequencia dos d-ésimos caracteres
13     //caractere '\0' → count[1]++
14     for(int i=l; i<=r; i++) count[a[i][d] + 1]++;
15
16
17     ...
18
19
20 }
```

Radix Sort - MSD - Etapa 2 - Posições

- $0 \leq k \leq R$
- Posição de k: `count[k] = count[k-1]+count[k]`

```
                ...   ‘    a    b    c    d    e    ...
count[R+1] [...   0    0    0    3    3    0    ...]
                      |_+_|
```

Radix Sort - MSD - Etapa 2 - Posições

- $0 \leq k \leq R$
- Posição de k: `count[k] = count[k-1]+count[k]`

```
                  ...   `    a    b    c    d    e    ...
count[R+1] [...   0     0    0    3    3    0    ...]
                          |_+_|
```

Radix Sort - MSD - Etapa 2 - Posições

- $0 \leq k \leq R$
- Posição de k: `count[k] = count[k-1]+count[k]`

```
                ...   `    a    b    c    d    e    ...
count[R+1] [...   0    0    0    3    3    0    ...]
                                |_+_|
```

Radix Sort - MSD - Etapa 2 - Posições

- $0 \leq k \leq R$
- Posição de k: `count[k] = count[k-1]+count[k]`

```
                ...   ‘    a    b    c    d    e    ...
count[R+1] [...  0    0    0    3    3    0    ...]
                                |_+_|
```

Radix Sort - MSD - Etapa 2 - Posições

- $0 \leq k \leq R$
- Posição de k: `count[k] = count[k-1]+count[k]`

```
                  ...   ‘    a    b    c    d    e    ...
count[R+1] [...   0    0    0    3    6    0    ...]
                                      |_+_|
```

Radix Sort - MSD - Etapa 2 - Posições

- $0 \leq k \leq R$
- Posição de k: `count[k] = count[k-1]+count[k]`

```
                  ...   `    a    b    c    d    e    ...
count[R+1] [...   0     0    0    3    6    6    ...]
                                          |_+_|
```

```
1  typedef char Item;
2  #define maxstring 101
3  #define bitsbyte 8
4  #define R (1 << bitsbyte)
5
6  //Strings: ordena para o d-ésimo caractere
7  void radixMSD(char a[][maxstring], int l, int r, int d) {
8      int count[R+1];
9      memset(count, 0, sizeof(int)*(R+1));
10
11     //frequencia dos d-ésimos caracteres
12     for(int i=l; i<=r; i++) count[a[i][d] + 1]++;
13
14     //calculando as posições
15     //'\0': count[0]=0
16     for (int i=1; i<R; i++) count[i] += count[i-1];
17
18     ...
19
20
21
22 }
```

Radix Sort - MSD - Etapa 3 - Distribuição

- `char aux[r-l+1][MAXSTRING]`
- `strcpy(aux[count[ v[k][0] ]++], v[k])`

```
  k            0    1    2    3    4    5
  v[0][0]   [ aab  aa   bbaa aabb baa  bc ]

                ...  ‘   a    b    c    d    e   ...
count[R+1]   [...  0   0    3    6    6    6   ...]

                0    1    2    3    4    5
  aux[0]    [ aab                          ]
```

Radix Sort - MSD - Etapa 3 - Distribuição

- `char aux[r-l+1][MAXSTRING]`
- `strcpy(aux[count[ v[k][0] ]++], v[k])`

```
  k            0    1    2    3    4    5
  v[0][0]   [ aab  aa   bbaa aabb baa  bc ]

              ...  ‘    a    b    c    d    e  ...
count[R+1] [...  0    1    3    6    6    6  ...]

              0    1    2    3    4    5
 aux[0]    [ aab                          ]
```

Radix Sort - MSD - Etapa 3 - Distribuição

- `char aux[r-l+1][MAXSTRING]`
- `strcpy(aux[count[ v[k][0] ]++], v[k])`

```
  k            0    1    2    3    4    5
  v[1][0]   [ aab  aa   bbaa aabb baa  bc ]

              ...  ‘    a    b    c    d    e  ...
count[R+1] [...  0    1    3    6    6    6  ...]

              0    1    2    3    4    5
 aux[1]    [ aab  aa                      ]
```

Radix Sort - MSD - Etapa 3 - Distribuição

- `char aux[r-l+1][MAXSTRING]`
- `strcpy(aux[count[ v[k][0] ]++], v[k])`

```
  k            0    1    2    3    4    5
  v[1][0]   [ aab  aa   bbaa aabb baa  bc ]

              ...  ‘    a    b    c    d    e  ...
count[R+1] [...  0    2    3    6    6    6  ...]

              0    1    2    3    4    5
  aux[1]   [ aab  aa                     ]
```

Radix Sort - MSD - Etapa 3 - Distribuição

- `char aux[r-l+1][MAXSTRING]`
- `strcpy(aux[count[ v[k][0] ]++], v[k])`

```
  k            0    1    2    3    4    5
  v[2][0]   [ aab  aa   bbaa aabb baa  bc ]

              ...  ‘    a    b    c    d    e  ...
count[R+1] [...  0    2    3    6    6    6  ...]

              0    1    2    3    4    5
 aux[3]    [ aab  aa        bbaa           ]
```

Radix Sort - MSD - Etapa 3 - Distribuição

- `char aux[r-l+1][MAXSTRING]`
- `strcpy(aux[count[ v[k][0] ]++], v[k])`

```
  k           0    1    2    3    4    5
  v[2][0]   [ aab  aa   bbaa aabb baa  bc ]

              ...  ‘    a    b    c    d    e  ...
count[R+1] [...  0    2    4    6    6    6  ...]

              0    1    2    3    4    5
  aux[3]    [ aab  aa        bbaa           ]
```

Radix Sort - MSD - Etapa 3 - Distribuição

- `char aux[r-l+1][MAXSTRING]`
- `strcpy(aux[count[ v[k][0] ]++], v[k])`

```
  k            0    1    2    3    4    5
  v[3][0]   [ aab  aa   bbaa aabb baa  bc ]

               ...  ‘    a    b    c    d    e  ...
count[R+1] [...  0    2    4    6    6    6  ...]

               0    1    2    3    4    5
  aux[2]   [ aab  aa   aabb bbaa           ]
```

Radix Sort - MSD - Etapa 3 - Distribuição

- `char aux[r-l+1][MAXSTRING]`
- `strcpy(aux[count[ v[k][0] ]++], v[k])`

```
  k            0    1    2    3    4    5
  v[3][0]   [ aab  aa   bbaa aabb baa  bc ]

              ...  ‘    a    b    c    d    e  ...
count[R+1] [...  0    3    4    6    6    6  ...]

              0    1    2    3    4    5
 aux[2]    [ aab  aa   aabb bbaa           ]
```

Radix Sort - MSD - Etapa 3 - Distribuição

- `char aux[r-l+1][MAXSTRING]`
- `strcpy(aux[count[ v[k][0] ]++], v[k])`

```
  k            0    1    2    3    4    5
  v[4][0]   [ aab  aa   bbaa aabb baa  bc ]

              ...  ‘    a    b    c    d    e  ...
count[R+1] [...  0    3    4    6    6    6  ...]

              0    1    2    3    4    5
 aux[4]    [ aab  aa   aabb bbaa baa       ]
```

Radix Sort - MSD - Etapa 3 - Distribuição

- `char aux[r-l+1][MAXSTRING]`
- `strcpy(aux[count[ v[k][0] ]++], v[k])`

```
  k            0    1    2    3    4    5
  v[4][0]   [ aab  aa   bbaa aabb baa  bc ]

              ...  ‘    a    b    c    d    e  ...
count[R+1] [...  0    3    5    6    6    6  ...]

              0    1    2    3    4    5
 aux[4]    [ aab  aa   aabb bbaa baa       ]
```

Radix Sort - MSD - Etapa 3 - Distribuição

- `char aux[r-l+1][MAXSTRING]`
- `strcpy(aux[count[ v[k][0] ]++], v[k])`

```
  k            0    1    2    3    4    5
  v[5][0]   [ aab  aa   bbaa aabb baa  bc ]

              ...  ‘    a    b    c    d    e  ...
count[R+1] [...  0    3    5    6    6    6  ...]

               0    1    2    3    4    5
  aux[5]    [ aab  aa   aabb bbaa baa  bc ]
```

Radix Sort - MSD - Etapa 3 - Distribuição

- `char aux[r-l+1][MAXSTRING]`
- `strcpy(aux[count[ v[k][0] ]++], v[k])`

```
  k            0    1    2    3    4    5
  v[5][0]   [ aab  aa   bbaa aabb baa  bc ]

              ...  ‘    a    b    c    d    e  ...
count[R+1] [...  0    3    6    6    6    6  ...]

               0    1    2    3    4    5
  aux[5]    [ aab  aa   aabb bbaa baa  bc ]
```

Radix Sort - MSD - Etapa 3 - Distribuição

- `char aux[r-l+1][MAXSTRING]`
- `strcpy(aux[count[ v[k][0] ]++], v[k])`

```c
typedef char Item;
#define maxstring 101
#define bitsbyte 8
#define R (1 << bitsbyte)

//Strings: ordena para o d-ésimo caractere
void radixMSD(char a[][maxstring], int l, int r, int d) {

    char aux[r-l+1][maxstring];
    int count[R+1];
    memset(count, 0, sizeof(int)*(R+1));

    //frequencia dos d-ésimos caracteres
    for(int i=l; i<=r; i++) count[a[i][d] + 1]++;

    //calculando as posições
    for (int i=1; i<R; i++) count[i] += count[i-1];

    //redistribui as chaves: ordena em aux
    for (int i=l; i<=r; i++) strcpy(aux[count[a[i][d]]++], a[i]);

    //copia para o original
    for (int i=l; i<=r; i++) strcpy(a[i], aux[i - l]);
    ...
}
```

Radix Sort - MSD - Etapa 4 - Subconjunto

- Subconjunto de `i`: `count[i-1]` até `count[i]-1`
  - `count[i-1]` primeira posição de `i`
  - `count[i]-1` última posição de `i`
- Chamadas recursivas para os subconjuntos

```
subconjunto a : count[a-1] até count[a]-1
    l = 0
    r = 3-1
```

```
              0    1    2    3    4    5
v[5]    [ aab  aa   aabb bbaa baa  bc ]

             ...   ‘    a    b    c    d    e    ...
count[R+1] [...   0    3    6    6    6    6    ...]
```

Radix Sort - MSD - Etapa 4 - Subconjunto

- Subconjunto de `i`: `count[i-1]` até `count[i]-1`
  - `count[i-1]` primeira posição de `i`
  - `count[i]-1` última posição de `i`
- Chamadas recursivas para os subconjuntos

```
subconjunto b : count[b-1] até count[b]-1
    l = 3
    r = 6-1

              0    1    2    3    4    5
    v[5]   [ aab  aa   aabb bbaa baa  bc ]

             ...  ‘   a    b    c    d    e   ...
count[R+1] [...  0   3    6    6    6    6   ...]
```

Radix Sort - MSD - Etapa 4 - Subconjunto

- Subconjunto de `i`: `count[i-1]` até `count[i]-1`
  - `count[i-1]` primeira posição de `i`
  - `count[i]-1` última posição de `i`
- Chamadas recursivas para os subconjuntos

```
subconjunto c : count[c-1] até count[c]-1
    l = 6
    r = 6-1
```

```
              0    1    2    3    4    5
v[5]    [ aab  aa   aabb bbaa baa  bc ]

             ...  ‘   a    b    c    d    e    ...
count[R+1] [...  0   3    6    6    6    6    ...]
```

```
1  //Strings: ordena para o d-ésimo caractere
2  void radixMSD(char a[][maxstring], int l, int r, int d) {
3
4      /*** subconjuntos unitários ou vazios ***/
5      if(r<=l) return;
6      ...
7      /*** subconjunto ***
8        count[i-1] posição da primeira chave com o caractere i
9        count[i]-1 posição da última chave com o caractere i
10
11       não vazios: l<r
12       vazios:     l>r (count[i-1]=count[i])
13       unitários:  l=r (count[i-1]=count[i]-1)
14             \0  1  ...  A  B  ...  a  b  c  d
15             2 | 2 | 2 | 2 | 4 | 4 | 4 | 8 | 9 | 9
16             count[0] : strings finalizadas
17             subconjuntos [1-A]: 2-1
18             subconjunto de B  : 2-3
19             subconjuntos ]B-a]: 4-3
20             subconjunto de b  : 4-7
21             subconjunto de c  : 8-8
22
23       todos finalizados: count[0]=(r-l+1), conjuntos vazios */
24     for (int i = 1; i < R; i++)
25       radixMSD(a, l + count[i-1], l + count[i]-1, d+1);
26 }
```

```
1  //Strings: ordena para o d-ésimo caractere
2  void radixMSD(char a[][maxstring], int l, int r, int d) {
3
4      if(r<=l) return;
5
6      char aux[r-l+1][maxstring];
7      int count[R+1];
8      memset(count, 0, sizeof(int)*(R+1));
9
10     //frequências
11     for(int i=l; i<=r; i++) count[a[i][d] + 1]++;
12
13     //posições
14     for (int i=1; i<R; i++) count[i] += count[i-1];
15
16     //distribuição e cópia
17     for (int i=l; i<=r; i++) strcpy(aux[count[a[i][d]]++], a[i]);
18     for (int i=l; i<=r; i++) strcpy(a[i], aux[i - l]);
19
20     //subconjuntos
21     for (int i = 1; i < R; i++)
22       radixMSD(a, l + count[i-1], l + count[i]-1, d+1);
23
24 }
```

Linearítmicos

- $O(n \log n)$
- Ordenação é por comparação do valor da chave
- Vantagem: mais amplo
  - Vários tipos de chaves podem usar o mesmo algoritmo
  - Chaves negativas

Lineares

- $O(n)$
- Ordenação é por comparação na estrutura da chave:
  - Comparação por unidade de representação
  - Unidade decimal, bytes, caracteres
  - byte: intervalo de 0 até 255
- Desvantagem: mais restrito
  - Dependente da forma da representação do tipo de dados
  - Quantidade de unidades impacta no custo
  - int (32 bits), long (64 bits), char (8 bits)
  - float: expoente+mantissa
  - negativos: bit de sinal (e com resto de divisão?)

Métodos de ordenação

algorithm	stable?	inplace?	order of growth of typical number calls to charAt() to sort N strings from an R-character alphabet (average length w, max length W)		sweet spot
			running time	extra space	
insertion sort for strings	yes	yes	between N and N^2	1	small arrays, arrays in order
quicksort	no	yes	$N \log^2 N$	$\log N$	general-purpose when space is tight
mergesort	yes	no	$N \log^2 N$	N	general-purpose stable sort
3-way quicksort	no	yes	between N and $N \log N$	$\log N$	large numbers of equal keys
LSD string sort	yes	no	NW	N	short fixed-length strings
MSD string sort	yes	no	between N and Nw	$N + WR$	random strings