Ordenação de Dados

Algoritmos eficientes

Algoritmo	Melhor caso	Pior caso	Caso médio
Inserção	$O(n)$	$O(n^2)$	$O(n^2)$
Seleção	$O(n^2)$	$O(n^2)$	$O(n^2)$
Bolha	$O(n)$	$O(n^2)$	$O(n^2)$
Bogosort	$O(n)$	$O(\infty)$	$O(n*n!)$

Algoritmos eficientes:

Caso médio

$O(n \log n)$

Algoritmos eficientes

- **Quick sort**
- **Heap sort**
- **Merge sort**

Quicksort/ Quick Sort (1961)

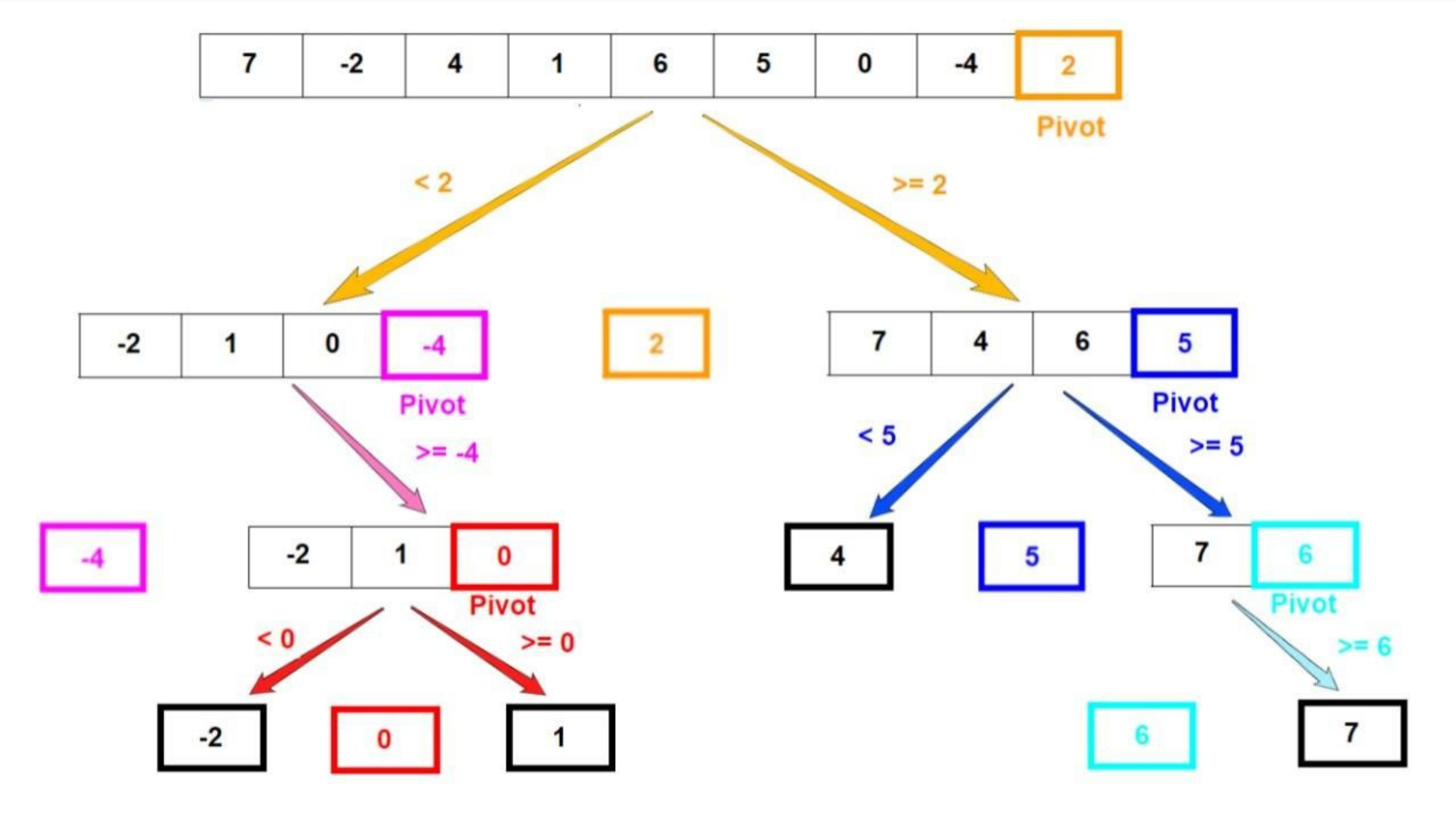
- Método dividir para conquistar
- Algoritmo
  - 1) escolher um valor (**pivô**) do conjunto a ordenar
  - 2) particionar o conjunto em dois subconjuntos: menores e maiores que o pivô
  - 3) repetir recursivamente para cada subconjunto (subconjuntos de tamanho zero ou um estão ordenados e não são mais repassados)

Quicksort: algoritmo de Lomuto

```
algorithm quicksort(A, lo, hi) is

    if lo < hi then

        p := partition(A, lo, hi)

        quicksort(A, lo, p - 1)

        quicksort(A, p + 1, hi)
```

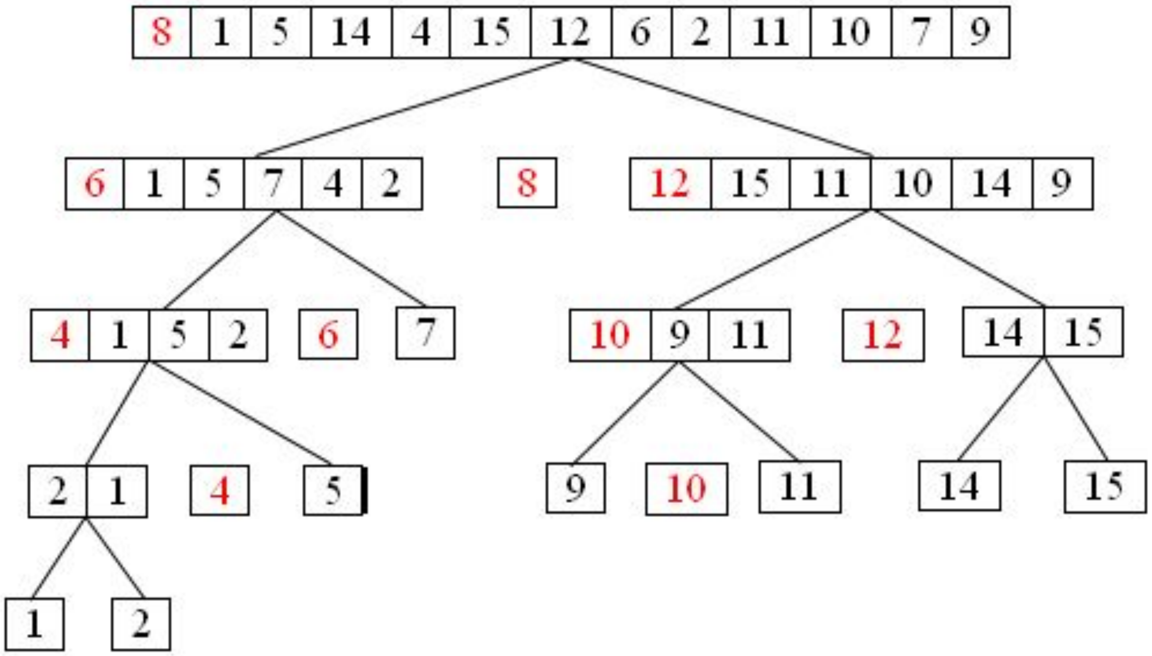
```
algorithm partition(A, lo, hi) is

    pivot := A[hi]
    i := lo
    for j := lo to hi do

        if A[j] < pivot then

            swap A[i] with A[j]

            i := i + 1

    swap A[i] with A[hi]

    return i
```

Quicksort



7
-2
4
1
6
5
0
-4
2
Pivot
< 2
>= 2
-2
1
0
-4
Pivot
2
7
4
6
5
Pivot
>= -4
< 5
>= 5
-4
-2
1
0
Pivot
4
5
7
6
Pivot
< 0
>= 0
>= 6
-2
0
1
6
7

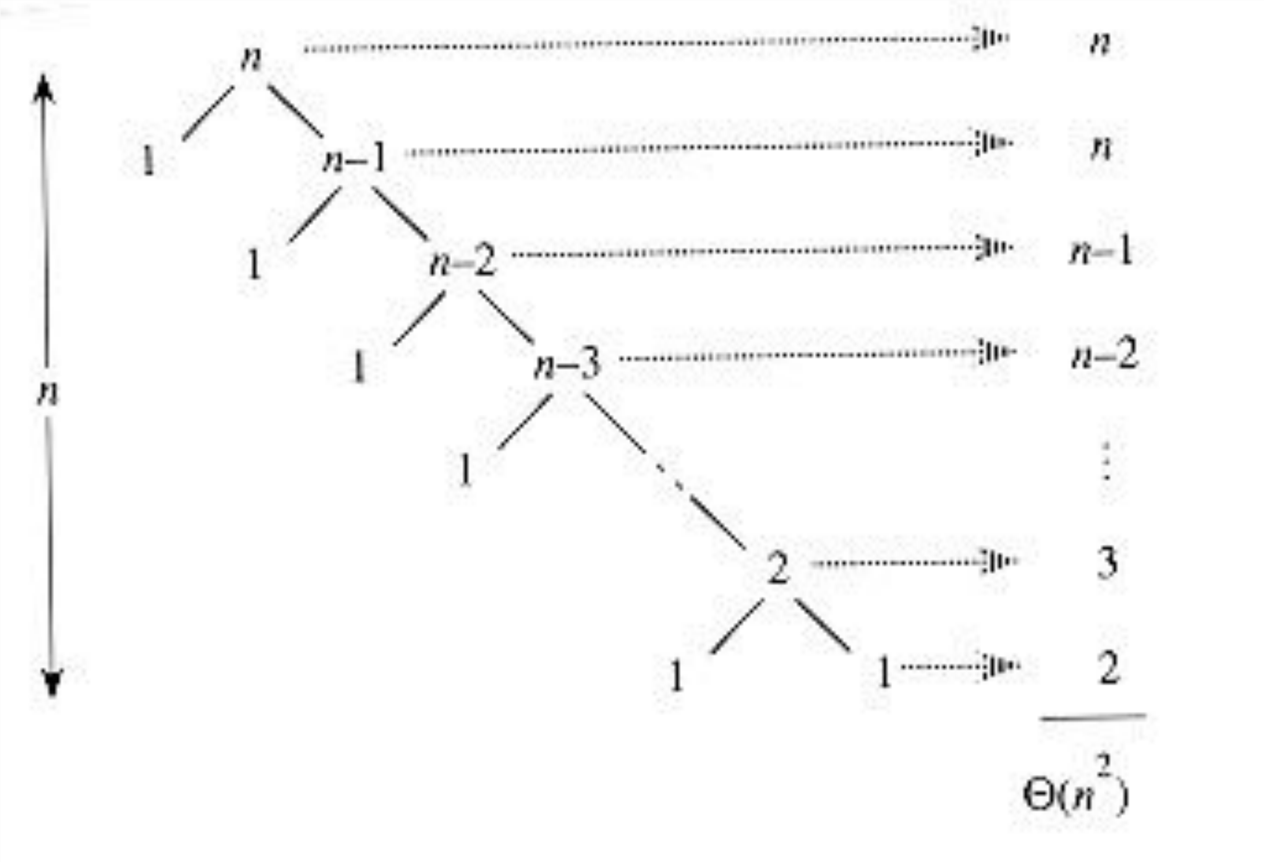
Quicksort



Ao final, junta-se os elementos

1	2	4	5	6	7	8	9	10	11	12	14	15

Quicksort: degeneração



Quicksort: problema

- Escolha do pivô
  - Pivô ótimo == mediana $\rightarrow$ $O(n \log n)$
  - Pivô ruim == degeneração $\rightarrow$ $O(n^2)$

Quicksort

- Veloz e simples de implementar
- Tende a dar melhor resultado que outros algoritmos O(n log n)
- Pode degenerar, mas há formas de evitar este problema
- Não usa estruturas auxiliares explícitas, pois as implementações são recursivas (usam espaço de pilha de execução)

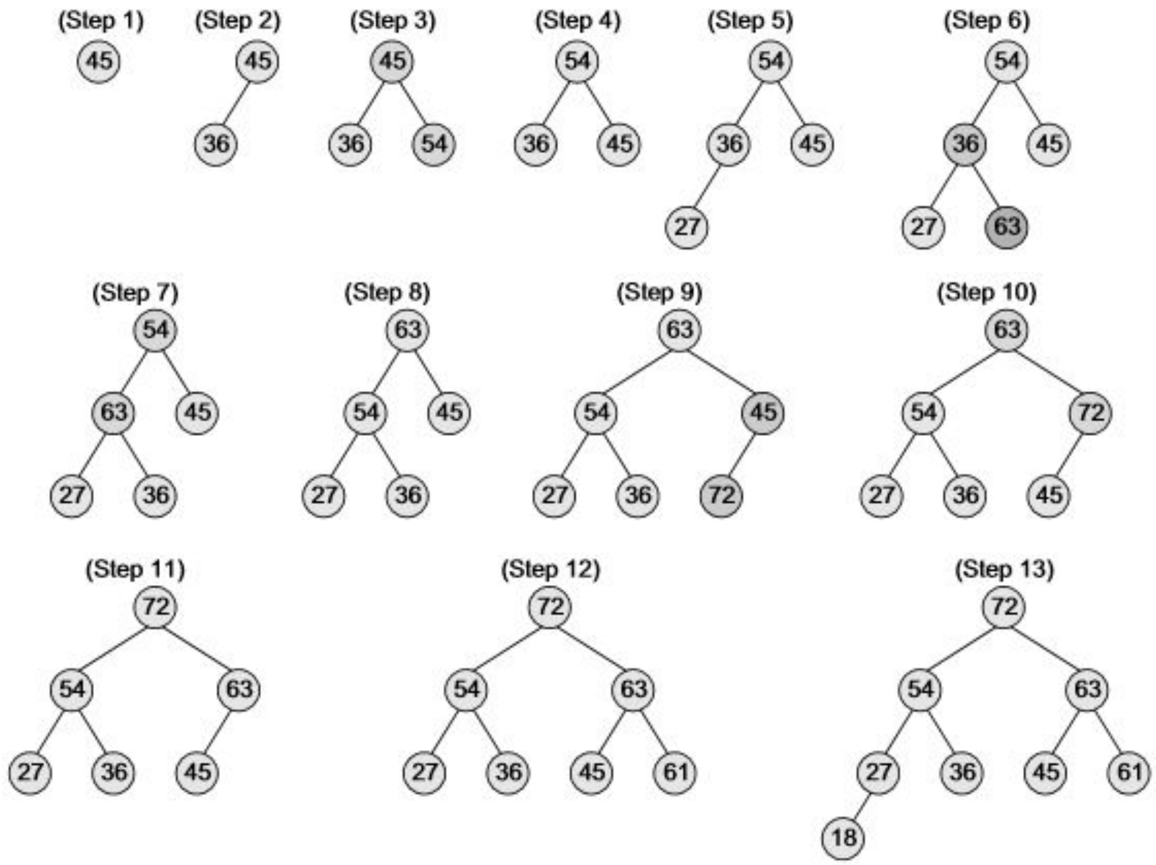
Heapsort / Heap Sort (1964)

- Baseado no uso da estrutura de Heap
- Algoritmo

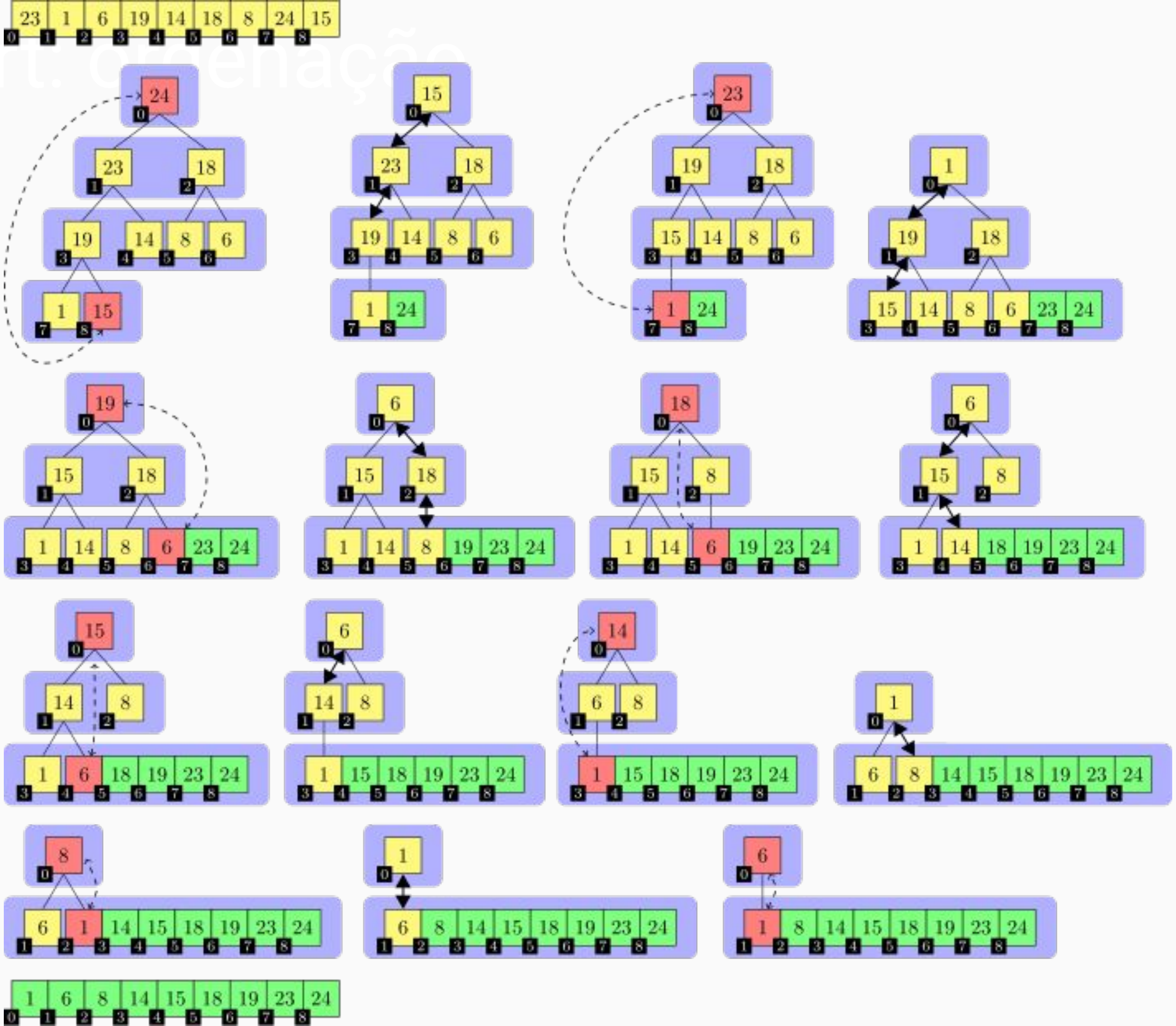
    1) construir heap (buildMaxHeap( ) ou heapify( ))
    2) retirar raiz inserindo último elemento em seu lugar. Colocar elemento retirado na posição final do conjunto ordenado.
    3) chamar siftDown( ) para a nova raiz
    4) enquanto houver elementos no heap, voltar para o passo 2

Heapsort: buildMaxHeap()

{45, 36, 54, 27, 63, 72, 61, 18}



Heapsort: ordenação

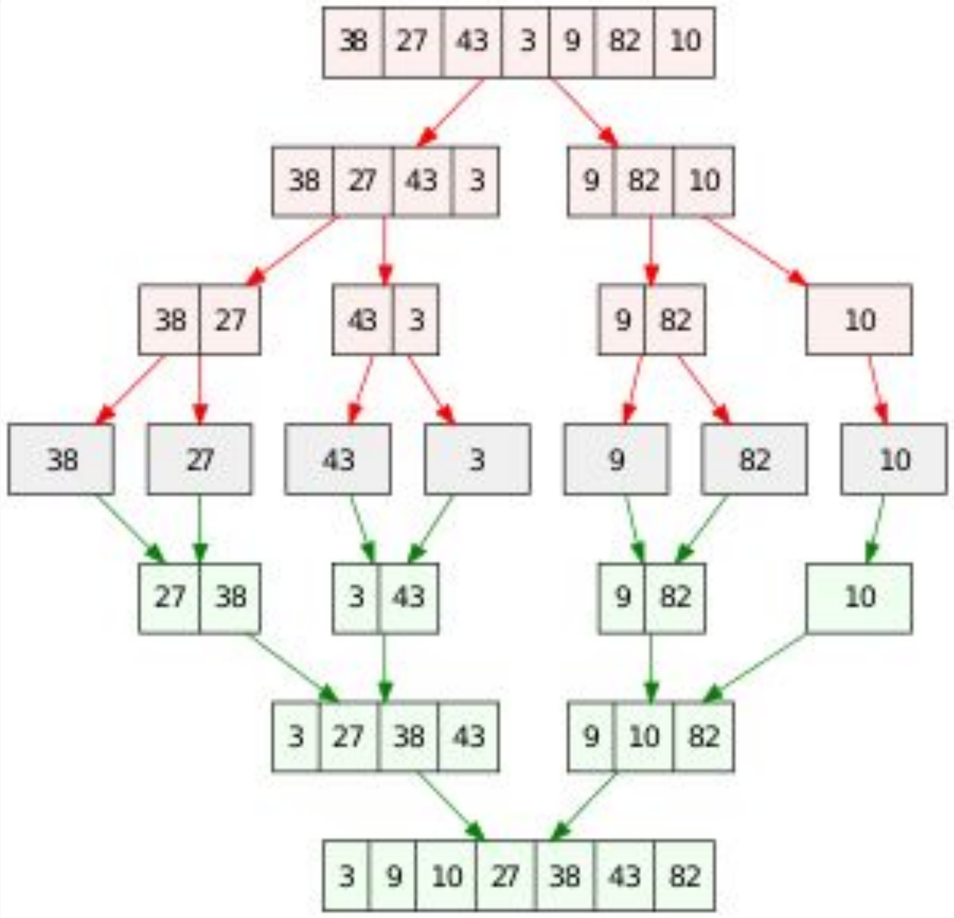


Heapsort

- Não degenera
- Limite superior de execução → O(n log n)
    - Adequado a sistemas de tempo-real e sistemas de segurança crítica (quicksort pode sofrer ataque que força degeneração)
- Em geral um pouco mais lento que melhores implementações do Quicksort
- Exige construção do Heap (estrutura extra)

Mergesort / Merge Sort

- Método divisão-e-conquista baseado em *merge*
- Algoritmo:
    1) Dividir o conjunto em *n* subconjuntos de tamanho 1
    2) Repetidamente realizar *merge* dos subconjuntos até haver apenas 1


38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

Mergesort

- Estável
- Usa memória extra (nas implementações mais comuns)
- Adequado à paralelização
- Adequado à uso externo

Algoritmo	Melhor caso	Pior caso	Caso médio
Bogosort	$O(n)$	$O(\infty)$	$O(n*n!)$
Inserção	$O(n)$	$O(n^2)$	$O(n^2)$
Seleção	$O(n^2)$	$O(n^2)$	$O(n^2)$
Bolha	$O(n)$	$O(n^2)$	$O(n^2)$
Quicksort	$O(n \log n)$	$O(n^2)$	$O(n \log n)$
Heapsort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Mergesort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

Comparação de algoritmos de ordenação

Outros algoritmos

- Cocktail Shaker
- Shell Sort
- Bucketsort
- Counting Sort
- LSD Radix
- MSD Radix
- outros...

Cocktail Shaker / Bolha bidirecional

- Variação do Bubble, com passadas bidirecionais
- Marginalmente melhor que o Bubble
  - Próximo a O(n) para listas quase ordenadas

Shell Sort / Shellsort

Gaps = {N/2, N/4, ..., 2, 1}

Para cada gap em Gaps

  Realizar insertion sort na lista gerada pelos elementos separados por gap

Shell Sort / Shellsort

Listas de gaps conhecidas:

Knuth Gaps (1973) = {1, 4, 13, 40, 121, 364, 1093, 3280, 9841}
Ciura Gaps (2001) = {1, 4, 10, 23, 57, 132, 301, 701, 1750}
Tokuda Gaps (2021) = {1, 4, 9, 20, 45, 102, 230, 516, 1158}

...

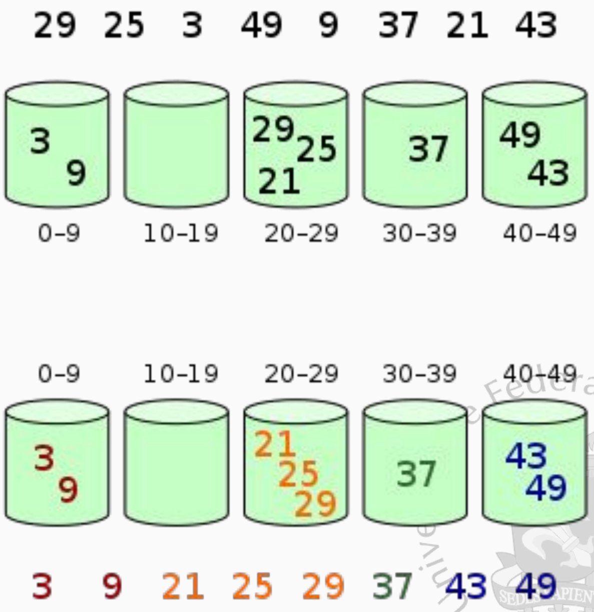
Bucket Sort

Inicializar um vetor de “baldes” vazios

Vá para o vetor original, colocando cada elemento em um balde

Ordenar cada balde não-vazio

Colocar os elementos dos baldes não-vazios no vetor original



Bucket Sort

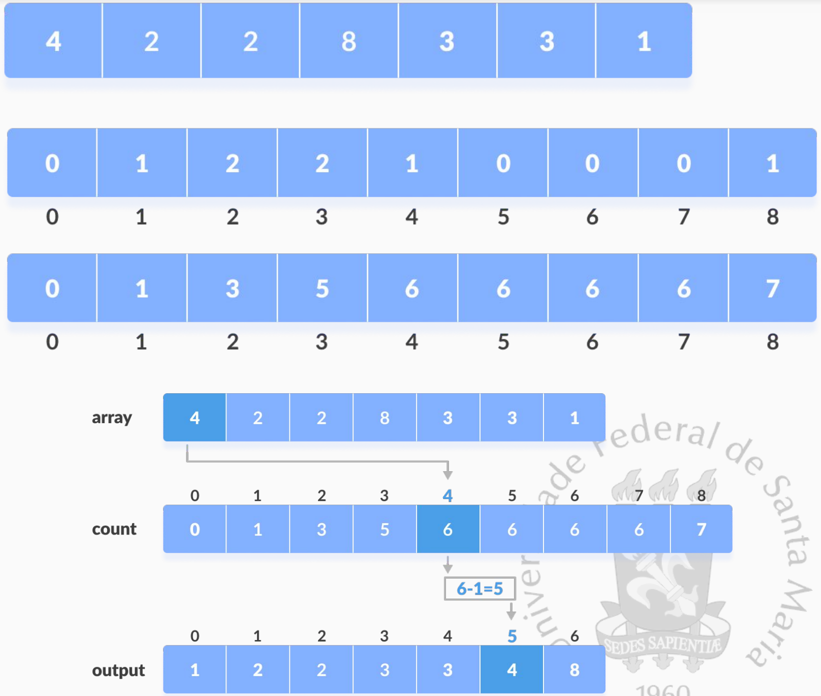
Que algoritmo usar para ordenar um balde?

O que acontece se distribuição não for uniforme? (ou pior, se todos os dados forem para o mesmo balde?)

Uso de estruturas dinâmicas para criar os baldes em memória (listas, arrays dinâmicos, etc)

Counting Sort

1- encontrar maior valor dos dados (max)
2- inicializar vetor cont, de tamanho max+1, com zeros
3- para cada elemento dos dados, incrementar o valor de cont[elemento] (criar um histograma)
4- realizar uma soma acumulativa ou soma de prefixos no vetor cont
5- criar um vetor de saída, com tamanho igual ao dos dados
6- para cada elemento dos dados, colocar o mesmo na posição saída[cont[elemento]] e decrementar o valor de cont[elemento]. Percorrer os dados em ordem inversa para manter estabilidade.



Counting Sort

Não comparativo

Algoritmo de ordenação de inteiros “pequenos” e positivos

Diferença entre menor e maior valor dita a complexidade

O(n+k) onde k é a faixa de valores não negativos de dados

Nos casos em que k (maior valor) for muito menor que n (número de dados), o algoritmo possui boa eficiência de espaço

Variante com uso de mapa de bits em caso de chaves únicas (ou para realizar unificação de chaves)

Radix Sort

Não comparativo

Dois subtipos: LSD (*Least Significant Digit*) e MSD (*Most Significant Digit*)

Radix Sort - LSD

Para cada dígito, a partir do menos significativo, nos dados de entrada, colocar o dado atual no *bucket* correspondente

Dados = [170, 45, 75, 90, 2, 802, 2, 66]

Primeira rodada:
Dados = [{170, 90}, {2, 802, 2}, {45, 75}, {66}]

Segunda rodada:
Dados = [{02, 802, 02}, {45}, {66}, {170, 75}, {90}]

Terceira rodada:
Dados = [{002, 002, 045, 066, 075, 090}, {170}, {802}]

Radix Sort - MSD

Para cada dígito, a partir do mais significativo, nos dados de entrada, colocar o dado atual no *bucket* correspondente

Repetir recursivamente, para cada *bucket* com mais de um dado, para o próximo dígito mais significativo

Dados = [170, 045, 075, 025, 002, 024, 802, 066]
Primeira rodada:
Dados = [{045, 075, 025, 002, 024, 066}, {170}, {802}]

Segunda rodada:
Dados = [{ {002}, {025, 024}, {045}, {066}, {075} }, 170, 802]

Terceira rodada:
Dados = [002, { {024}, {025} }, 045, 066, 075 , 170, 802]

Ordenação de dados - algoritmos atuais

- Híbridos
  - Introsort
  - Timsort

Introsort / Introspective sort (1997)

- Desenvolvido para uso na biblioteca padrão do C++
- Presente nas bibliotecas padrão em Go, Java e .NET
- Combina quicksort, heapsort e inserção (ou Shellsort)
    - Começa com quicksort, alternando para heapsort se o nível de recursão ultrapassa um *threshold* baseado em log n, alternando para inserção quando o número de elementos está abaixo de um limite (arbitrariamente escolhido)
- Busca os benefícios de cada algoritmo

Introsort / Introspective sort (1997)

- ordenar(dados)
    - maxDepth = (log2 length(dados)) * 2
    - introsort(dados, maxDepth)

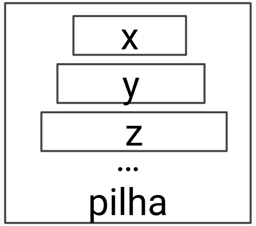
- introsort(dados, maxDepth)
    - N = length(dados)
    - if N < 16
        - insertion(dados)
    - else if maxDepth == 0
        - heapsort(dados)
    - Else
        - P = partition(dados)
        - introsort(A[1:p-1], maxdepth - 1)
        - introsort(A[p+1:N], maxdepth - 1)

Timsort (2002)

- Derivado de Merge sort e inserção
- Algoritmo padrão de ordenação para Python (desde versão 2.3)
  - Também usado em Java, Android, Swift, Rust
- Projetado para aproveitar *runs* (sequências ordenadas) que ocorrem naturalmente nos dados
- Reduz o número de comparações necessárias

Timsort (2002)

- Algoritmo:
  a. Iterar sobre os dados coletando *runs* e colocando-as em uma pilha (se não há uma *run*, ou não alcança um tamanho mínimo, cria-se uma por inserção)
     - Toda vez que as *runs* no topo da pilha preenchem um critério, elas sofrem merge
     - Quando os dados na entrada terminam, as *runs* na pilha sofrem merge até haver somente um conjunto ordenado restante
- Para obter merges balanceados, considere-se as 3 *runs* no topo da pilha (X, Y e Z) e as regras abaixo:
  a. length(Z) > length(X + Y)
  b. length(Y) > length(X)
- Se alguma regra for violada, fazer merge de Y com o menor entre X e Z até rebalancear, para depois procurar a próxima *run*



Timsort (2002)

- Os critérios de merge servem para:
    - manter os merges (aproximadamente) balanceados, enquanto mantém um compromisso entre não haver merges com muita frequencia, explorar a existência de *runs* na cache e manter a decisão sobre o merge simples
- Timsort original não é *in-place*. Implementações desta forma possuem overhead de espaço (O(N)).

Timsort (2002)

- Implementações *in-place* possuem overhead de tempo
  - Para minimizar isto, criou-se uma otimização que diminui tanto espaço quanto tempo:
    - Faz-se uma busca binária para encontrar onde o primeiro elemento da segunda *run* será inserido na primeira
    - Faz-se outra busca para encontrar a posição de inserção do último elemento da segunda *run* na primeira
    - Elementos antes da primeira inserção e depois da última não serão modificados, então são deslocados para suas posições finais
    - Faz-se o merge dos elementos restantes na primeira *run* com a segunda no espaço reservado entre as anteriores

Timsort (2002)

Outras otimizações incluem:

- Modo “galopante”
    - Se durante o merge forem encontrados muitos elementos consecutivos da mesma *run* na ordem correta, alterna-se para modo “galopante”, em que inicia-se uma busca exponencial pelo fim da sequencia
- *Runs* descendentes
    - Se for encontrada uma sequencia estritamente descendente, a mesma é revertida ao ser levada para a pilha
- Tamanho mínimo de *run*
    -