

Algoritmos de Ordenação (Parte 2)

Prof. Jefferson T. Oliva

Algoritmos e Estrutura de Dados I (AE22CP)
Engenharia de Computação
Departamento Acadêmico de Informática (Dainf)
Universidade Tecnológica Federal do Paraná (UTFPR)
Campus Pato Branco

Sumário

- Quicksort
- Heapsort

Quicksort

- Ordenação por troca
- O *quicksort* adota a estratégia de divisão e conquista
 - Divisão: particionar o arranjo $X[p...q]$ em dois sub-arranjos $X[p...r-1]$ e $X[r+1...q]$, tais que $X[p...r-1] \leq X[r] \leq X[r+1...q]$
 - Conquista: ordenar os dois sub-arranjos $X[p...r-1]$ e $X[r+1...q]$ por chamadas recursivas do *quicksort*

- Procedimento $quicksort(X, p, q)$
 - 1 definir o pivô r e as posições $i = p$ e $j = q$
 - 2 enquanto $i \leq j$, trocar de posição os elementos maiores (lado esquerdo do arranjo) com os itens menores (lado direito) que o pivô
 - 3 $quicksort(X, p, j)$
 - 4 $quicksort(X, i, q)$

- Implementação (quando o pivô fica entre as posições *esq* e *dir*)

```
void quicksort(int x[], int esq, int dir){
    int i = esq, j = dir, pivo = x[(i + j) / 2], aux;

    do{
        while (x[i] < pivo)
            i++;

        while (x[j] > pivo)
            j--;

        if (i <= j){
            aux = x[i];
            x[i] = x[j];
            x[j] = aux;
            i++;
            j--;
        }
    }while (i <= j);

    if (j > esq)
        quicksort(x, esq, j);

    if (i < dir)
        quicksort(x, i, dir);
}
```

- Exemplo

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
25	57	48	37	12	92	86	33

- Para $esq = 0$, $dir = 7$ e $pivo = X[(0 + 7)/2] = X[3] = 37$, temos

i	j	pivo	x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
0	7	37	25	57	48	37	12	92	86	33
1	7	37	25	57	48	37	12	92	86	33
1	7	37	25	33	48	37	12	92	86	57
2	6	37	25	33	48	37	12	92	86	57
2	5	37	25	33	48	37	12	92	86	57
2	4	37	25	33	48	37	12	92	86	57
2	4	37	25	33	12	37	48	92	86	57
3	3	37	25	33	12	37	48	92	86	57
4	2	37	25	33	12	37	48	92	86	57

Troca
Após troca

Troca
Após troca

Novas partições

- Após a execução, fazemos mais duas chamadas recursivas
 - $quicksort(v, esq, j) \Rightarrow quicksort(x, 0, 2)$
 - $quicksort(v, i, dir) \Rightarrow quicksort(x, 4, 7)$

- Exemplo (continuação: quicksort(x, 0, 2))
 - Para $esq = 0$, $dir = 2$ e $pivo = X[(0 + 2)/2] = X[1] = 33$, temos

i	j	pivo	x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
0	2	33	25	33	12	37	48	92	86	57
1	2	33	25	33	12	37	48	92	86	57
1	2	33	25	12	33	37	48	92	86	57
2	1	33	25	12	33	37	48	92	86	57

Troca
Após Troca
Nova partição

- Após a execução, fazemos mais uma chamada recursiva (uma chamada não é realizada porque i é igual a dir)
 - $quicksort(v, esq, j) \Rightarrow quicksort(x, 0, 1)$

- Exemplo (continuação: quicksort(x, 4, 7)), supondo que quicksort(x, 0, 1) foi executada
 - Para $esq = 4$, $dir = 7$ e $pivo = X[(4 + 7)/2] = X[5] = 92$, temos

i	j	pivo	x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
4	7	92	12	25	33	37	48	92	86	57
5	7	92	12	25	33	37	48	92	86	57
5	7	92	12	25	33	37	48	57	86	92
6	6	92	12	25	33	37	48	57	86	92
7	5	92	12	25	33	37	48	57	86	92

Troca
Após Troca

Nova partição

- Após a execução, fazemos mais uma chamada recursiva (uma chamada não é realizada porque i é igual a dir)
 - quicksort(v, esq, j) \Rightarrow quicksort(x, 4, 5)

- Implementação (quando o pivô fica na posição *esq*)

```
void troca(int *a, int *b){
    int aux = *a;
    *a = *b;
    *b = aux;
}

int particionar(int v[], int esq, int dir){
    int pivo = v[esq], i = esq + 1, j = dir;
    while (i <= j){
        while ((v[i] <= pivo) && (i <= dir))
            i++;
        while ((v[j] > pivo) && (j >= esq))
            j--;
        if (i < j)
            troca(&v[i], &v[j]);
    }
    troca(&v[esq], &v[j]);
    return j;
}

void quicksort2(int v[], int esq, int dir){
    if (esq < dir){
        int j = particionar(v, esq, dir);
        quicksort2(v, esq, j - 1);
        quicksort2(v, j + 1, dir);
    }
}
```

- Exemplo

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
25	57	48	37	12	92	86	33

- Para $esq = 0$, $dir = 7$ e $pivo = X[esq] = X[0] = 25$, temos

i	j	pivo	X[0]	X[1]	X[2]	X[3]	X[4]	X[5]	X[6]	X[7]
-	-	-	25	57	48	37	12	92	86	33
1	7	25	25	57	48	37	12	92	86	33
1	6	25	25	57	48	37	12	92	86	33
1	5	25	25	57	48	37	12	92	86	33
1	4	25	25	57	48	37	12	92	86	33
1	4	25	25	12	48	37	57	92	86	33
2	3	25	25	12	48	37	57	92	86	33
2	2	25	25	12	48	37	57	92	86	33
2	1	25	25	12	48	37	57	92	86	33
2	1	25	25	12	48	37	57	92	86	33
2	1	25	12	25	48	37	57	92	86	33

troca
após troca

Parada do loop
troca

- Após a execução, fazemos mais duas chamadas recursivas
 - $quicksort(v, esq, j - 1) \Rightarrow quicksort(x, 0, 0)$
 - $quicksort(v, j + 1, dir) \Rightarrow quicksort(x, 2, 7)$

- Exemplo

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
25	57	48	37	12	92	86	33

- Para $esq = 2$, $dir = 7$ e $pivo = X[esq] = X[2] = 8$, temos

i	j	pivo	X[0]	X[1]	X[2]	X[3]	X[4]	X[5]	X[6]	X[7]
-	-	-	12	25	48	37	57	92	86	33
3	7	48	12	25	48	37	57	92	86	33
4	7	48	12	25	48	37	57	92	86	33
4	7	48	12	25	48	37	33	92	86	57
5	7	48	12	25	48	37	33	92	86	57
7	6	48	12	25	48	37	33	92	86	57
7	5	48	12	25	48	37	33	92	86	57
7	4	48	12	25	48	37	33	92	86	57
7	4	48	12	25	48	37	33	92	86	57
7	4	48	12	25	33	37	48	92	86	57

troca
após troca

s/ troca ($i \geq j$)
parada do loop

- Após a execução, fazemos mais duas chamadas recursivas
 - $quicksort(v, esq, j - 1) \Rightarrow quicksort(x, 2, 3)$ // já ordenado
 - $quicksort(v, j + 1, dir) \Rightarrow quicksort(x, 5, 7)$

- Exemplo

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
25	57	48	37	12	92	86	33

- Para $esq = 5$, $dir = 7$ e $pivo = X[esq] = X[5] = 92$, temos

i	j	pivo	X[0]	X[1]	X[2]	X[3]	X[4]	X[5]	X[6]	X[7]
-	-	-	12	25	33	37	48	92	86	57
6	7	92	12	25	33	37	48	92	86	57
7	7	92	12	25	33	37	48	92	86	57
8	7	92	12	25	33	37	48	92	86	57
8	7	92	12	25	33	37	48	92	86	57
8	7	92	12	25	33	37	48	57	86	92

s/ troca ($i \geq j$)
parada do loop

- Após a execução, fazemos mais duas chamadas recursivas
 - $quicksort(v, esq, j - 1) \Rightarrow quicksort(x, 5, 3) // (i > j)$
 - $quicksort(v, j + 1, dir) \Rightarrow quicksort(x, 5, 5)$

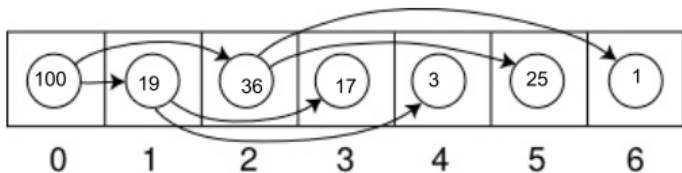
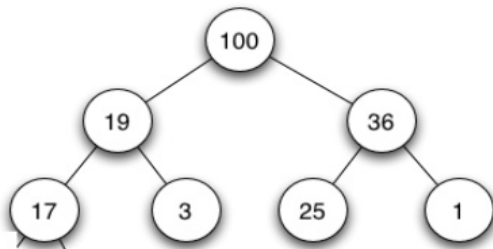
- No melhor caso e caso médio, o algoritmo tem custo de tempo de $O(n \log_2 n)$
- Dependendo da forma em que o pivô é escolhido e da forma que o arranjo está organizado, o custo pode ser de $O(n^2)$ (pior caso)
- O método não é estável

- Links interessantes:
 - Dança húngara:
<https://www.youtube.com/watch?v=ywWBy6J5gz8>
 - Simulador gráfico do *quicksort*:
<https://visualgo.net/bn/sorting>

Heapsort

- Ordenação por seleção
- Baseado no princípio de ordenação por seleção em árvore binária
- O método consiste em duas fases distintas:
 - 1 Montagem da árvore binária (HEAP)
 - 2 Seleção dos elementos na ordem desejada

Heapsort



- Em uma *heap*
 - O sucessor à esquerda do elemento de índice i é o elemento de índice $2 * (i + 1) - 1$, se $2 * (i + 1) - 1 < n$, caso contrário não existe
 - O sucessor à direita do elemento de índice i é o elemento de índice $2 * (i + 1)$, se $2 * (i + 1) < n$, caso contrário não existe

- Código para rearranjar um vetor para que o mesmo atenda a condição para ser uma *heap*

```
void gerarHeap(int v[], int n){  
    int esq = n / 2;  
  
    while (esq >= 0){  
        refazer(v, esq, n - 1);  
        esq--;  
    }  
}
```

- Código para reorganizar um vetor para que o mesmo atenda a condição para ser uma *heap* (continuação)

```
void refazer(int v[], int esq, int dir){
    int j = (esq + 1) * 2 - 1;
    int x = v[esq];

    while (j <= dir){
        if ((j < dir) && (v[j] < v[j + 1]))
            j++;

        if (x >= v[j])
            break;

        v[esq] = v[j];
        esq = j;
        j = (esq + 1) * 2 - 1;
    }

    v[esq] = x;
}
```

- Função principal

```
void heapsort(int v[], int n){
    int x;
    int dir = n - 1;

    gerarHeap(v, n);

    while (dir > 1){
        x = v[0];
        v[0] = v[dir];
        v[dir] = x;
        dir--;
        refazer(v, 0, dir);
    }
}
```

- Exemplo

<i>iteração</i>	x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
<i>fazheap: i=3</i>	25	57	48	37	12	92	86	33
<i>fazheap: i=2</i>	25	57	92	37	12	48	86	33
<i>fazheap: i=1</i>	25	57	92	37	12	48	86	33
<i>fazheap: i=0</i>	92	57	86	37	12	48	25	33
<i>heapsort: i=7</i>	86	57	48	37	12	33	25	92
<i>heapsort: i=6</i>	57	37	48	25	12	33	86	92
<i>heapsort: i=5</i>	48	37	33	25	12	57	86	92
<i>heapsort: i=4</i>	37	25	33	12	48	57	86	92
<i>heapsort: i=3</i>	33	25	12	37	48	57	86	92
<i>heapsort: i=2</i>	25	12	33	37	48	57	86	92
<i>heapsort: i=1</i>	12	25	33	37	48	57	86	92

- À primeira vista, parece que o *heap sort* não apresenta bons resultados
- Não é um algoritmo de ordenação estável
- O algoritmo não é recomendado para pequenos conjuntos de elementos
- Custo de tempo (melhor, médio e pior caso): $O(n \log_2(n))$

- Exercício: aplicar o *heap sort* em um arranjo de 10 elementos:
 - Organizado em ordem crescente
 - Organizado em ordem decrescente

- Links interessantes:
 - Dança húngara:
<https://www.youtube.com/watch?v=Xw2D9aJRBY4&list=RDCMUClqiLefbVHs0AXDaxQJH7Xw&index=10>
 - Simulador gráfico do *heap sort*: <https://www.cs.usfca.edu/~galles/visualization/HeapSort.html>



Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C.
Introduction to Algorithms.
Third edition, The MIT Press, 2009.



Horowitz, E., Sahni, S. Rajasekaran, S.
Computer Algorithms.
Computer Science Press, 1998.



Rosa, J. L. G.
Métodos de Ordenação. SCE-181 – Introdução à Ciência da
Computação II.
Slides. Ciência de Computação. ICMC/USP, 2018.



Ziviani, N.
Projeto de Algoritmos - com implementações em Java e C++.
Thomson, 2007.