Algoritmos de ordenação eficientes - Quick sort

emanoelim@utfpr.edu.br

Quick Sort

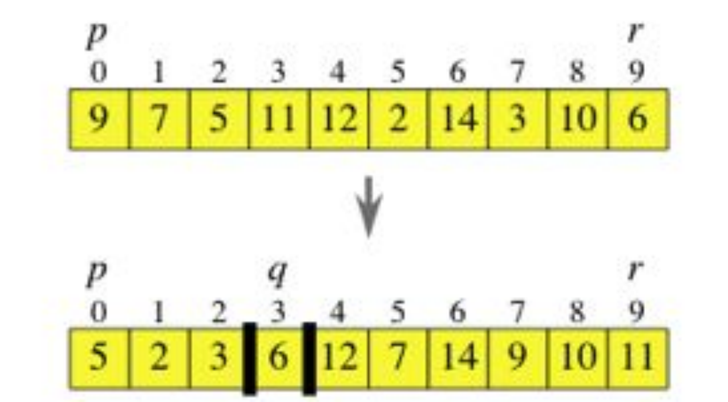
- Assim como o merge sort, o quick sort é um algoritmo de ordenação que usa a ideia de divisão e conquista. Portanto, podemos deduzir que ele é um algoritmo recursivo.
- A diferença entre os dois algoritmos está na maneira como é feita a divisão do problema.

Quick Sort

- O ponto de início do quick sort é escolher um pivô, que é um item qualquer do vetor.
- A partir do pivô, o vetor é separado em duas partes:
  - Todos os itens menores que o pivô irão formar um subvetor;
  - Todos os itens maiores que o pivô irão formar outro subvetor.
- Esse processo é feito repetidas vezes até encontrar subvetores com um único item (caso base).

Quick Sort

- Exemplo onde o pivô escolhido foi o 6:



Quick Sort

- Apesar de falarmos do conceito de subvetor, não é necessário criar vetores auxiliares para cada subvetor, como era feito no merge sort.
- A maneira mais simples de trabalhar com esse algoritmo é ir movendo todos itens maiores que o pivô para a sua direita e todos menores que o pivô para a sua esquerda (tudo no próprio vetor).

Quick Sort

- A seguir é apresentado um exemplo de método para particionamento;
- Este método é atribuído a Nico Lomuto e popularizou-se em livros sobre algoritmos;
- O último item do vetor é selecionado como pivô.

Quick Sort

```c
int separa(int v[], int primeiro, int ultimo){
    int pivo = v[ultimo];
    int esquerda = primeiro;
    int atual;
    for(atual = primeiro; atual < ultimo; atual++){
        if (v[atual] <= pivo) {
            troca(&v[atual], &v[esquerda]);
            esquerda++;
        }
    }
    troca(&v[esquerda], &v[ultimo]);
    return esquerda;
}
```

Quick Sort

- Passo-a-passo:

4	2	6	1	3	5
esq					pivo

atual = 4
atual <= pivo? Sim, então troca atual e esquerda e incrementa esquerda:

4	2	6	1	3	5
	esq				pivo

Quick Sort

- Passo-a-passo:

4	2	6	1	3	5
	esq				pivo

atual = 2
atual <= pivo? Sim, então troca atual e esquerda e incrementa esquerda:

4	2	6	1	3	5
		esq			pivo

Quick Sort

- Passo-a-passo:

4	2	6	1	3	5
		esq			pivo

atual = 6
atual <= pivo? Não, então não faz nada:

4	2	6	1	3	5
		esq			pivo

Quick Sort

- Passo-a-passo:

4	2	6	1	3	5
		esq			pivo

atual = 1
atual <= pivo? Sim, então troca atual e esquerda e incrementa esquerda:

4	2	1	6	3	5
			esq		pivo

Quick Sort

- Passo-a-passo:

4	2	1	6	3	5
			esq		pivo

atual = 3
atual <= pivo? Sim, então troca atual e esquerda e incrementa esquerda:

4	2	1	3	6	5
				esq	pivo

Quick Sort

- Passo-a-passo:

4	2	1	3	6	5
				esq	pivo

atual = 5
atual <= pivo? Sim, então troca atual e esquerda e incrementa esquerda:

4	2	1	3	5	6
				pivo	esq

Quick Sort

- Passo-a-passo:

4	2	1	3	5	6
				pivo	esq

Após percorrer todo o vetor, troca esquerda e último:

4	2	1	3	5	6
				pivo	

Quick Sort

- Passo-a-passo:
  - Nesse ponto já foi percorrido o vetor todo. Do lado direito do pivo (5) existe um subvetor de apenas 1 elemento, ou seja, já chegou ao caso base.
  - O pivô já está na posição certa, então pode ser ignorado na próxima chamada.

Quick Sort

- Passo-a-passo:

4	2	1	3
esq			pivo

atual = 4
atual <= pivo? Não, então não faz nada:

4	2	1	3
esq			pivo

Quick Sort

- Passo-a-passo:

4	2	1	3
esq			pivo

atual = 2
atual <= pivo? Sim, então troca atual e esquerda e incrementa esquerda:

2	4	1	3
	esq		pivo

Quick Sort

- Passo-a-passo:

2	4	1	3
	esq		pivo

atual = 1
atual <= pivo? Sim, então troca atual e esquerda e incrementa esquerda:

2	1	4	3
		esq	pivo

Quick Sort

- Passo-a-passo:

	2	1	4	3
			esq	pivo

atual = 3
atual <= pivo? Sim, então troca atual e esquerda e incrementa esquerda:

	2	1	3	4
			pivo	esq

Quick Sort

- Passo-a-passo:

2	1	3	4
		pivo	esq

Após percorrer todo o vetor, troca esquerda e último:

2	1	3	4
		pivo	

Quick Sort

- Passo-a-passo:
  - Nesse ponto já foi percorrido o vetor todo. Do lado direito do pivo (3) existe um subvetor de apenas 1 elemento, ou seja, já chegou ao caso base.
  - O pivo já está na posição certa, então pode ser ignorado na próxima chamada.

Quick Sort

- Passo-a-passo:

2	1
esq	pivo

atual = 2
atual <= pivo? Não, então não faz nada:

2	1
esq	pivo

Quick Sort

- Passo-a-passo:

2	1
esq	pivo

atual = 1
atual <= pivo? Sim, então troca atual e esquerda e incrementa esquerda:

1	2
pivo	esq

Quick Sort

- Passo-a-passo:

1	2
pivo	esq

Após percorrer todo o vetor, troca esquerda e último:

1	2
pivo	

Quick Sort

- Passo-a-passo:
  - Nesse ponto já foi percorrido o vetor todo. Do lado direito do pivo (1) existe um subvetor de apenas 1 elemento, ou seja, já chegou ao caso base.
  - O pivo já está na posição certa, então o algoritmo termina.

Quick Sort

- Neste passo-a-passo, é possível ver que sempre que houver um item maior que o pivô, esse item vai sendo empurrado para a direita, até que ele fique após o pivô.

Quick Sort

- No passo-a-passo trabalhamos com um vetor onde ao final de uma iteração havia apenas um item à direita do pivô.
- Assim nós só precisávamos continuar ordenando os itens que restavam do lado esquerdo.
- Nem sempre isso vai acontecer. Considere o exemplo:

Quick Sort

- Passo-a-passo:

4	3	9	8	5
esq				pivo

atual = 4
atual <= pivo? Sim, então troca atual e esquerda e incrementa esquerda:

4	3	9	8	5
	esq			pivo

Quick Sort

- Passo-a-passo:

4	3	9	8	5
	esq			pivo

atual = 3
atual <= pivo? Sim, então troca atual e esquerda e incrementa esquerda:

4	3	9	8	5
		esq		pivo

Quick Sort

- Passo-a-passo:

4	3	9	8	5
		esq		pivo

atual = 9
atual <= pivo? Não, então não faz nada:

4	3	9	8	5
		esq		pivo

Quick Sort

- Passo-a-passo:

4	3	9	8	5
		esq		pivo

atual = 8
atual <= pivo? Não, então não faz nada:

4	3	9	8	5
		esq		pivo

Quick Sort

- Passo-a-passo:

4	3	9	8	5
		esq		pivo

atual = 5
atual <= pivo? Sim, então troca atual e esquerda e incrementa esquerda:

4	3	5	8	9
		pivo	esq	

Quick Sort

- Passo-a-passo:

4	3	5	8	9
		pivo	esq	

Após percorrer todo o vetor, troca esquerda e último:

4	3	5	8	9
		pivo		

Quick Sort

- Existe um vetor com 2 itens menores que o pivô e existe um vetor com 2 itens maiores que o pivô.
- É por isso que o algoritmo quick sort faz duas chamadas recursivas: uma para o vetor mais a esquerda e uma para o vetor mais a direita.

Quick Sort

```
void quicksort(int v[], int p, int u)
{
    if(p < u) {
        int j = separa(v, p, u);
        quicksort(v, p, j - 1);
        quicksort(v, j + 1, u);
    }
    return;
}
```

Quick Sort

- Complexidade de tempo (função de particionamento):
    - Existe um laço que percorre todo o vetor, ou seja, se o vetor tem tamanho n, o laço executa n vezes. Dentro do laço é feita uma comparação e uma troca. Após o laço é feita uma troca. Essas operações têm custo constante, então a complexidade do método de particionamento é O(n).

Quick Sort

- Complexidade de tempo (quick sort completo):
  - Análise para um **vetor em ordem decrescente** (Ex.: [5, 4, 3, 2, 1]):
    - O if dentro do laço sempre tem resultado falso, porque nunca haverá um item menor que o pivô.
    - A variável "esquerda" vai continuar guardando a posição do primeiro item do vetor (posição 0).
    - No final do laço será feita a troca do item na posição "esquerda" com o último item do vetor:
      [1, 4, 3, 2, 5]
    - O pivô vai ficar em sua posição correta e vai existir um vetor de n - 1 elementos do lado direito.
    - Isso implica que:

Quick Sort

- Complexidade de tempo (quick sort completo):
  - Análise para um **vetor em ordem decrescente** (Ex.: [5, 4, 3, 2, 1]):
    - quicksort(v, p, j - 1) será chamada uma vez e já irá retornar.
    - quicksort(v, j + 1, u) será chamada para:
      - n - 1 itens;
      - n - 2 itens;
      - n - 3 itens;
      - …

Conforme as aulas anteriores, isto leva a uma complexidade de $O(n^2)$

Quick Sort

- Complexidade de tempo (quick sort completo):
  - Análise para um **vetor em ordem crescente** (Ex.: [1, 2, 3, 4, 5]):
    - O if dentro do laço sempre tem resultado verdadeiro, porque todo item será menor que o pivô.
    - A variável "esquerda" sempre vai ser incrementada. Ao sair do laço vai guardar a última posição do vetor.
    - No final do laço será feita a troca do item na posição "esquerda" com o último item do vetor (que não mudará nada):
      [1, 2, 3, 4, 5]
    - O pivô vai ficar em sua posição correta e vai existir um vetor de n - 1 elementos do lado esquerdo.
    - Isso implica que:

Quick Sort

- Complexidade de tempo (quick sort completo):
  - Análise para um **vetor em ordem crescente** (Ex.: [1, 2, 3, 4, 5]):
    - quicksort(v, p, j - 1) será chamada:
      - n - 1 itens;
      - n - 2 itens;
      - n - 3 itens;
      - …

      Conforme as aulas anteriores, isto leva a uma complexidade de $O(n^2)$
    - quicksort(v, j + 1, u) será chamada uma vez e já irá retornar.

Quick Sort

- Complexidade de tempo (quick sort completo):
  - Tanto para um **vetor em ordem decrescente** quanto para um **vetor em ordem crescente** a complexidade será $O(n^2)$, ou seja, essas duas situações constituem o **pior caso** do quick sort.

Quick Sort

- Complexidade de tempo (quick sort completo):
  - O melhor caso para o quick sort seria quando o pivô sempre consegue dividir o vetor em dois subvetores de mesmo tamanho, pois o problema seria sempre quebrado ao meio.
  - As duas chamadas de quicksort() executariam o mesmo número de vezes, o que indica a seguinte fórmula de recorrência:

    $$T(n) = n + 2T(n / 2)$$

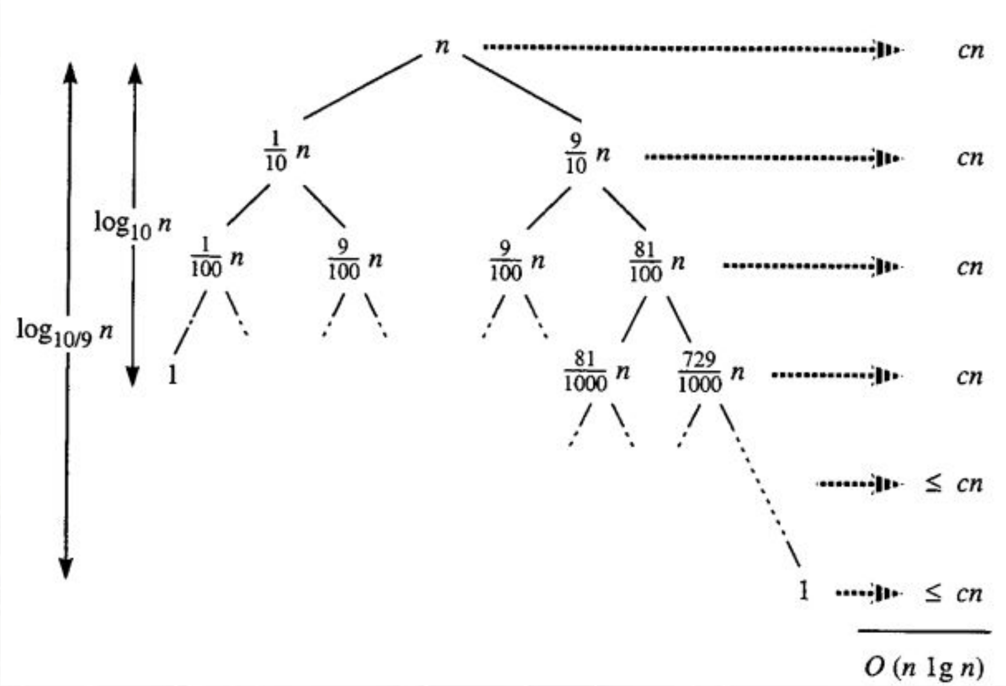
    $$T(1) = 1$$

  - Que é a mesma fórmula de recorrência do merge sort, que leva a uma complexidade de O(n log n).

Quick Sort

- Complexidade de tempo (quick sort completo):
  - Dificilmente o quick sort vai cair no melhor ou pior caso, ele geralmente fará uma divisão desbalanceada.
  - Imagine uma divisão onde um subvetor fica com cerca de 10% dos itens enquanto o outro subvetor fica com cerca de 90% dos itens, ou seja:

Quick Sort



- Mesmo em uma divisão que parece intuitivamente desbalanceada, o algoritmo consegue executar a um custo de O(n log n)
- Qualquer divisão proporcionalmente constante irá ter um custo de O(n log n)

Quick Sort

- Complexidade de tempo (quick sort completo):
  - Dizemos então que este seria um caso médio do quick sort: o pivô não divide os vetores ao meio, mas consegue fazer uma divisão proporcionalmente constante.

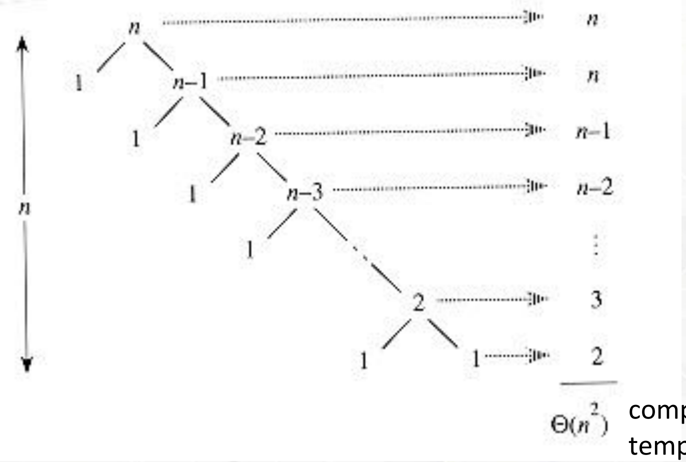
Quick Sort

- Complexidade de tempo (quick sort completo):
  - Melhor caso: $O(n \log n)$
  - Caso médio: $O(n \log n)$
  - Pior caso: $O(n^2)$

Quick Sort

- Complexidade de espaço:
  - No pior caso a complexidade de espaço será O(n), pois será necessário ter até n chamadas empilhadas:

complexidade de espaço

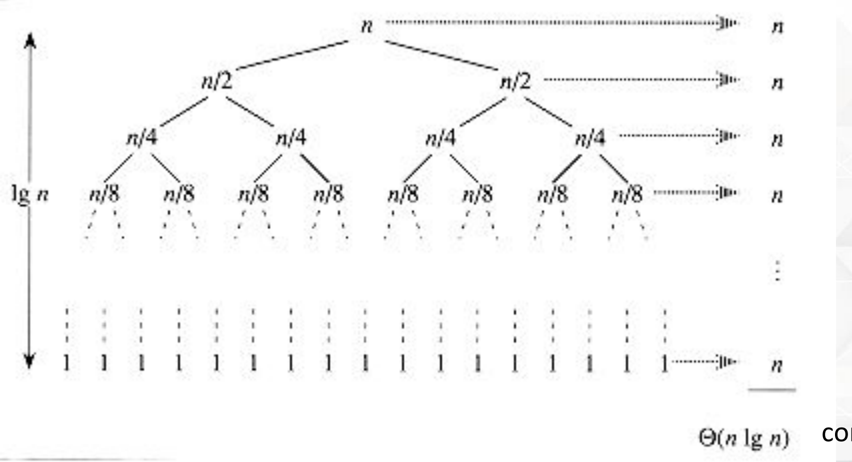


complexidade de tempo

Quick Sort

- Complexidade de espaço:
  - No melhor caso e no caso médio a complexidade de espaço será O(log n), pois será necessário ter até log n chamadas empilhadas:

complexidade de espaço



complexidade de tempo

Quick Sort

- Comparativo Quick x Merge
  - Em seu caso médio, o custo do quick sort em relação ao tempo é O(n log n).
  - O merge sort, em qualquer caso, tem um custo de tempo de O(n log n).
  - Como o quick sort não precisa de vetores auxiliares para fazer a ordenação, o seu consumo de espaço é O(log n).
  - Já o merge sort precisa criar vetores auxiliares, tendo consumo de espaço de O(n).
  - Eles "empatam" quanto ao consumo de tempo, mas o quick sort consome menos memória. Por este motivo, geralmente prefere-se usar o quick sort.

Quick Sort

Referências

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. “Algoritmos - teoria e prática” 6ª ed, Rio de Janeiro, Elsevier, 2002.
- Overview Quicksort: https://www.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/overview-of-quicksort
- Explicação da pilha de chamadas do quick sort: https://youtu.be/COk73cpQbFQ?t=672