Algoritmos e Estrutura de Dados

Fabrício Olivetti de França

02 de Fevereiro de 2019

Topics

1. Algoritmos de Ordenação Eficientes

Algoritmos de Ordenação Eficientes

Uma outra forma de pensar em ordenar registros é utilizando a técnica **dividir e conquistar** conforme aplicado pelo algoritmo **Quick Sort**.

Em resumo, dividir e conquistar reduz o problema principal em problemas menores recursivamente até que seja possível resolver o problema de forma trivial.

Especificamente no problema de ordenação, imagine que estamos ordenando provas em ordem alfabética. Podemos iniciar com uma ordenação grosseira e depois refinar o resultado.

Quick Sort

Pegamos a primeira prova da pilha de provas e criamos duas pilhas: a pilha da direita contém todas as provas cujo nome vem antes do nome atual, a da esquerda todas provas cujo nome aparecem depois.

Repetindo o processo, em dado momento teremos n pilhas com uma prova cada, ao juntarmos a pilha da mais a esquerda para a mais a direita, teremos nossas provas ordenadas!

Em linguagem algorítimica:

- particionamos os registros em torno de um elemento fazendo com que ele fique na posição p e tal que $i < p \implies K_i < K_p$ e $i > p \implies K_i > K_p$.
- repetimos o processo nos elementos de 0 a p e $p + 1$ a n.

Quick Sort

```
void quickSort(registro *base, int n) {
    if (n > 0)
    {
        int p = partition(base, n);
        quickSort(base, p);
        quickSort(base + p + 1, n - p - 1);
    }
}
```

```c
int partition(registro *base, int n) {
    registro pivot = base[0];
    int i=1, j;
    for (j=1; j<n; j++)
    {
        if (base[j].key < pivot.key)
        {
            swap(base+i, base+j);
            ++i;
        }
    }
    swap(base+i-1, base);
    return i-1;
}
```

Quick Sort

```
swap(x[1], x[1])
```

i, j

0	1	2	3	4	5	6	7	8
88	56	100	2	25	32	1	99	21

Quick Sort

i, j

0	1	2	3	4	5	6	7	8
88	56	100	2	25	32	1	99	21

Quick Sort

```
swap(x[2], x[3])
```

		i	j					
0	1	2	3	4	5	6	7	8
88	56	100	2	25	32	1	99	21

Quick Sort

```
swap(x[3], x[4])
```

			i	j				
0	1	2	3	4	5	6	7	8
88	56	2	100	25	32	1	99	21

Quick Sort

```
swap(x[4], x[5])
```

				i	j			
0	1	2	3	4	5	6	7	8
88	56	2	25	100	32	1	99	21

Quick Sort

```
swap(x[5], x[6])
```

					i	*j*		
0	1	2	3	4	5	6	7	8
88	56	2	25	32	100	1	99	21

Quick Sort

							i	j	
	0	1	2	3	4	5	6	7	8
	88	56	2	25	32	1	100	99	21

Quick Sort

`swap(6,8)`

						i		j
0	1	2	3	4	5	6	7	8
88	56	2	25	32	1	100	99	21

Quick Sort

`swap(0,6)`

0	1	2	3	4	5	6	7	8
88	56	2	25	32	1	21	99	100

Quick Sort

```
return 5
```

0	1	2	3	4	5	6	7	8
21	56	2	25	32	1	88	99	100

Quick Sort

	i	j			
0	1	2	3	4	5
21	56	2	25	32	1

Quick Sort

		i	j		
0	1	2	3	4	5
21	2	56	25	32	1

Quick Sort

		i		j	
0	1	2	3	4	5
21	2	56	25	32	1

Quick Sort

		i			j
0	1	2	3	4	5
21	2	56	25	32	1

Quick Sort

			i		j
0	1	2	3	4	5
21	2	1	25	32	56

Quick Sort

retorna $p = 3$

			i		j
0	1	2	3	4	5
1	2	21	25	32	56

Quick Sort

retorna $p = 7$.

	i	j
0	1	2
88	99	100

Quick Sort

retorna $p = 7$.

	i	j
0	1	2
88	99	100

Quick Sort

retorna $p = 0$.

	i	j
0	1	2
88	99	100

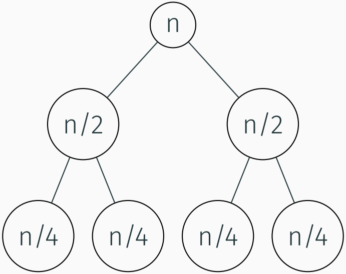
Quick Sort

	Insert	Bubble	Select	Quick
estável	✓	✓		
in-place	✓	✓	✓	✓
online	✓			
adaptivo	✓	✓		

A primeira chamada de `partition` percorre os n elementos da lista.

No melhor caso, o particionamento vai dividir a lista em duas partes iguais, ou seja, os dois próximos particionamentos percorrerão $n/2$ elementos (em um total de n elementos), cada uma das duas partições pode gerar duas chamadas de listas de tamanho $n/4$ (em um total de n elementos).

No melhor caso, temos uma complexidade $O(n \cdot k)$ sendo k a altura da árvore de partições.



O valor de k é quantas vezes podemos dividir n por 2 até atingir 1, ou seja, $\frac{n}{2^k} = 1$, temos então que:

$n = 2^k$

$\lg n = k$

Com isso nosso melhor caso é $O\big(n \log n\big)$.

Por outro lado, se o particionamento faz com que uma sublista tenha $n-1$ elementos e a outra nenhum, teremos uma sequência de $\sum_{i=0}^{n} n - i$ operações, o que leva a um pior caso de $O(n^2)$.

Para evitar o pior caso, devemos escolher um pivot que esteja aproximadamente posicionado no meio da lista. Podemos, por exemplo, calcular a mediana de uma amostra pequena da lista.

O custo extra pode compensar pelo fato de evitar o pior caso.

Complexidade Assintótica

	Insert	Bubble	Select	Quick
melhor	$O(n)$	$O(n)$	$O(n^2)$	$O(n \log n)$
pior	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$
médio	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n \log n)$

Merge Sort

Similar ao Quick Sort, o **Merge Sort** divide o problema de ordenação em problemas menores.

Merge Sort

Para tanto, ele faz chamadas recursivas para a faixa de valores de $[0, n/2[$ e $[n/2, n[$, em seguida executando um procedimento chamado `merge` que concatena o resultado das duas chamadas recursiva.

Ou seja, ele ordena as sublistas da metade inicial e da metade final, e depois concatena as duas de forma eficiente.

Merge Sort

```
void mergeSort(registro *base, int n) {
    if (n > 1)
    {
        int middle = n/2;
        mergeSort(base, middle);
        mergeSort(base + middle, n - middle);
        merge(base, middle, n);
    }
}
```

Merge Sort

```c
void merge(registro *base, int m, int n) {
    registro x[n];
    int j=0, k=m;

    for (int i=0; i<n; ++i) {
        if (j==m)        x[i] = base[k++];
        else if (k==n)   x[i] = base[j++];
        else if (base[j].key < base[k].key)
                         x[i] = base[j++];
        else             x[i] = base[k++];
    }
    for (int i=0; i<n; i++) base[i]=x[i];
}
```

Merge Sort

Merge!

j				k				
0	1	2	3	4	5	6	7	8
2	56	88	100	1	21	25	32	99

Merge Sort

Merge!

j → 0, k → 5

0	1	2	3	4	5	6	7	8
2	56	88	100	1	21	25	32	99
1								

Merge!

	j				k			
0	1	2	3	4	5	6	7	8
2	56	88	100	1	21	25	32	99
1	2							

Merge!

	j					k		
0	1	2	3	4	5	6	7	8
2	56	88	100	1	21	25	32	99
1	2	21						

Merge!

	j						k	
0	1	2	3	4	5	6	7	8
2	56	88	100	1	21	25	32	99
1	2	21	25					

Merge Sort

Merge!

	j							k
0	1	2	3	4	5	6	7	8
2	56	88	100	1	21	25	32	99
1	2	21	25	32				

Merge!

		j						k
0	1	2	3	4	5	6	7	8
2	56	88	100	1	21	25	32	99
1	2	21	25	32	56			

Merge Sort

Merge!

			j					k
0	1	2	3	4	5	6	7	8
2	56	88	100	1	21	25	32	99
1	2	21	25	32	56	88		

Merge!

			j					k
0	1	2	3	4	5	6	7	8
2	56	88	100	1	21	25	32	99
1	2	21	25	32	56	88	99	

Merge!

			j					k
0	1	2	3	4	5	6	7	8
2	56	88	100	1	21	25	32	99
1	2	21	25	32	56	88	99	100

Merge Sort

	Insert	Bubble	Select	Quick	Merge
estável	✓	✓			✓
in-place	✓	✓	✓	✓	
online	✓				
adaptivo	✓	✓			

O algoritmo Merge Sort divide cada lista em duas sublistas de igual tamanho, o procedimento de merge tem complexidade $O(k)$ sendo k a soma do número de elementos das duas sublistas.

Em cada nível da árvore fazemos então n operações (somatória de todos os merges). Como nossa árvore é balanceada, temos uma altura de $\log n$. Portanto, mesmo no pior caso, a complexidade é $O(n \log n)$.

O pior caso do Merge Sort faz cerca de 40% menos comparações que o caso médio do Quick Sort, porém necessita de uma estrutura auxiliar para o procedimento de `merge`.

Geralmente ela é utilizada para casos em que os registros somente podem ser acessados de forma eficiente na sequência (arquivos externos, listas ligadas).

Complexidade Assintótica

	Insert	Bubble	Select	Quick	Merge
melhor	$O(n)$	$O(n)$	$O(n^2)$	$O(n \log n)$	$O(n \log n)$
pior	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n \log n)$
médio	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n \log n)$	$O(n \log n)$

Próxima aula

Na próxima aula aprenderemos os algoritmos **heap sort** e **bucket sort**.