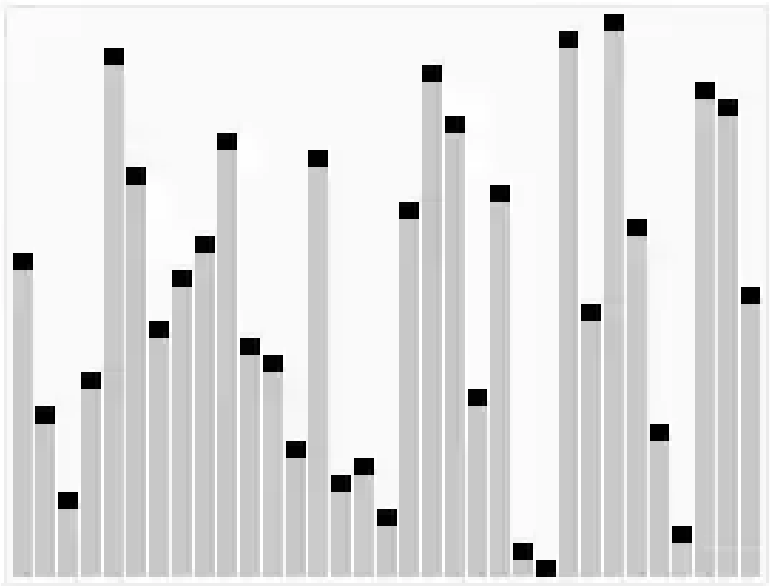
Quicksort

Quicksort



classe	Algoritmo de ordenação
estrutura de dados	Array, Listas ligadas
complexidade pior caso	{\displaystyle {O}(n^{2})}
complexidade caso médio	{\displaystyle {O}(n\log n)}
complexidade melhor caso	{\displaystyle {O}(n\log n)}
complexidade de espaços pior caso	{\displaystyle O(n)}
otimo	Não
estabilidade	não-estável

Algoritmos

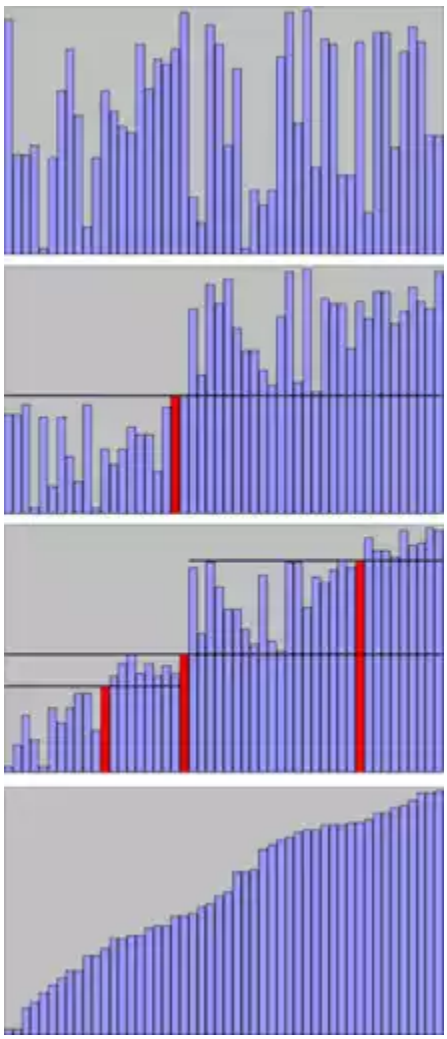
ver

O algoritmo **Quicksort** é um método de ordenação muito rápido e eficiente, inventado por C.A.R. Hoare em 1960[1], quando visitou a Universidade de Moscovo como estudante. Naquela época, Hoare trabalhou em um projeto de tradução de máquina para o National Physical Laboratory. Ele criou o '*Quicksort* ao tentar traduzir um dicionário de inglês para russo, ordenando as palavras, tendo como objetivo reduzir o problema original em subproblemas que

possam ser resolvidos mais fácil e rápido. Foi publicado em 1962 após uma série de refinamentos.[2]

Quicksort é um algoritmo de ordenação por comparação não-estável.

O algoritmo Computacional



Algumas etapas do algoritmo Quicksor t.

O Quicksort adota a estratégia de divisão e conquista. A estratégia consiste em rearranjar as chaves de modo que as chaves "menores" precedam as chaves "maiores". Em seguida o Quicksort ordena as duas sublistas de chaves menores e maiores recursivamente até que a lista completa se encontre ordenada. [3]Os passos são:

Escolha um elemento da lista, denominado *pivô*;

Rearranje a lista de forma que todos os elementos anteriores ao pivô sejam menores que ele, e todos os elementos posteriores ao pivô sejam maiores que ele. Ao fim do processo o pivô estará em sua posição final e haverá duas sublistas não ordenadas. Essa operação é denominada *partição*;

Recursivamente ordene a sublista dos elementos menores e a sublista dos elementos maiores;

A base da recursão são as listas de tamanho zero ou um, que estão sempre ordenadas. O processo é finito, pois a cada iteração pelo menos um elemento é posto em sua posição final e não será mais manipulado na iteração seguinte.

Complexidade

Complexidade de tempo:

Comportamento no pior caso

O pior caso de particionamento ocorre quando o elemento pivô divide a lista de forma desbalanceada, ou seja, divide a lista em duas sublistas: uma com tamanho 0 e outra com tamanho *n - 1* (no qual *n* se refere ao tamanho da lista original). Isso pode ocorrer quando o elemento pivô é o maior ou menor elemento da lista, ou seja, quando a lista já está ordenada, ou inversamente ordenada.

Se isso acontece em todas as chamadas do método de particionamento, então cada etapa recursiva chamará listas de tamanho igual à lista anterior - 1. Teremos assim, a seguinte relação de recorrência:

$T(n) = T(n - 1) + T(0) + \theta(n)$

$= T(n - 1) + \theta(n)$.

Se somarmos os custos em cada nível de recursão, teremos uma série aritmética que tem valor $\theta(n^2)$, assim, o algoritmo terá tempo de execução igual à $\theta(n^2)$.

Comportamento no melhor caso

O melhor caso de particionamento acontece quando ele produz duas listas de tamanho não maior que *n/2,* uma vez que uma lista terá tamanho [n/2] e outra tamanho [n/2] - 1. Nesse caso, o quicksort é executado com maior rapidez. A relação de recorrência é a seguinte:

$T(n)$
$\displaystyle \leq$
$2T(n/2) + \theta(n)$

que, a partir do teorema mestre, terá solução $T(n) = O(n \log n)$.

Complexidade de espaço: $\theta(\log_2 n)$ no melhor caso e no caso médio e $\theta(\log_2 n)$ no pior caso. R. Sedgewick desenvolveu uma versão do Quicksort com partição recursão de cauda que tem complexidade $\theta(n^2)$ no pior caso.

Implementações

Pseudocódigo

```
procedimento QuickSort(X[], IniVet, FimVet)
var
  i, j, pivo, aux
início
  i <- IniVet
  j <- FimVet
  pivo <- X[(IniVet + FimVet) div 2]
    enquanto(i < j)
    |     enquanto (X[i] <= pivo) faça
    |       |   i <- i + 1
    |     fimEnquanto
    |     enquanto (X[j] > pivo) faça
    |       |   j <- j - 1
    |     fimEnquanto
    |     se (i < j) então
    |       |   aux  <- X[i]
    |       |   X[i] <- X[j]
    |       |   X[j] <- aux
    |     fimSe
```

```
    |     i <- i + 1
```

```
    |     j <- j - 1
    fimEnquanto
    se (j > IniVet) então
    |  QuickSort(X, IniVet, j)
    fimSe
    se (i < FimVet) então
    |  QuickSort(X, j+1, FimVet)
    fimse
```

```
fimprocedimento
```

Linguagem de programação JAVA

```java
//Código para Ordenar um vetor de um tipo genérico.
public class QuickSort<T extends Comparable<T>> implements Sorting<T> {
  public void sort(T[] elements) {
    int inicio = 0;
    int fim = elements.length;
    quickSort(elements, inicio, fim);
  }
  private void quickSort(T[] elements, int inicio, int fim) {
    //Verifica se o inteiro inicio é menor que o inteiro fim
    if(inicio < fim) {
      //Ocorre a chamada do método particiona e a chamada
      //recursiva do método quickSort, recebendo diferentes parâmetros
      int pivo = particiona(elements, inicio, fim);
      quickSort(elements, inicio, pivo - 1);
      quickSort(elements, pivo + 1, fim);
    }
  }
  //Este ocorre o particionamento virtual do vetor
  private int particiona(T[] elements, int inicio, int fim) {
    T pivo = elements[fim];
    int i = inicio;
    for (int j = inicio ; j <= fim - 1 ; j++){
      //Verifica se cada elemento é menor do que o pivo
      if(elements[j].compareTo(pivo) < 1) {
        /Aqui realiza o swap (atualização dinâmica dos elementos)
        T aux = elements[i];
        elements[i] = elements[j];
        elements[j] = aux;
        i += 1;
      }
    }
    T aux = elements[i];
    elements[i] = elements[fim];
    elements[fim] = aux;
    return i;
```

```
    }
}
```

C

Uma forma de se fazer o quickSort é considerar o primeiro elemento como pivô, sempre organizando o vetor de tal forma que, para ordem crescente, os elementos menores que o pivô sejam postos à sua esquerda e os maiores, à sua direita. O processo utiliza recursão até que se chegue a apenas um elemento do vetor.

```
//Código para Ordenar um vetor de 10 inteiros.
# include<stdio.h>
# include<stdlib.h>
# define TAM 10
void quick(int vet[], int esq, int dir){
   int pivo = esq, i,ch,j;       //Declaração das variavés e inicialização do pivo com o primeiro algarismo da sequencia
   for(i=esq+1;i<=dir;i++){      //Percorre todos os espaços do vetor
      j = i;                     //atribuição de valor
      if(vet[j] < vet[pivo]){    //verifica se o vetor da posição pivo é maior que de outra posição
       ch = vet[j];              //ch recebe o valor que é menor
       while(j > pivo){          //repete enquanto o j que é a posição do algarismo menor que o pivo ficar na posição 0
         vet[j] = vet[j-1];      //reorganiza a posição de vetores
         j--;                    //decremento para a organização
       }
       vet[j] = ch;              // atribuição da variavel menor que o pivo na posição inicial
       pivo++;                   // aumenta a posição do pivo em uma unidade
      }
   }
   if(pivo-1 >= esq){            // verifica se o valor do pivo é maior que o final do vetor.
      quick(vet,esq,pivo-1);     //final da execursão da função
   }
   if(pivo+1 <= dir){            //verifica se o valor do pivo é menor, indicando que ainda estar dentro das limitações do vetor
      quick(vet,pivo+1,dir);     //chama a função para eecutar novamente
   }
}
```

```
int main(){
   int vet[TAM],i;              //Declara a variavel i e o vetor vet com
{{subst:Número2palavra2|10}} posições de 0 a 9.
   printf("Digite 10 numeros");    //Imprime na tela a mensagem.
   for(i=0;i<TAM;i++)            //Percorrertodo os espaços do vetor
      scanf("%d",&vet[i]);       //armazena os dados coletados todo no vetor
   quick(vet,0,TAM-1);           //Chama a função quick com os tres parametros: o vetor, 0 o
inicio do vetor e o fim.
   for(i=0;i<TAM;i++)            //percorre o vetor
      printf("%d ",vet[i]);      //imprime o vetor reorganizado
   printf("\n");
   return 0;
}
```

Python Quicksort também pode ser implementado de formar Recursiva.

```
1 def quickSort(vetor,inicio,fim):
2
  if(inicio < fim):
3
     q = pQSort(vetor,inicio,fim) #q[0] = pivo; q[1] = inicio; q[2] = fim
4
     quickSort(vetor, q[1],q[0]-1) #(vetor, inicio, pivo-1)
5
     quickSort(vetor, q[0]+1,q[2]) #(vetor, pivo+1, fim)
6


7
def pQSort(vetor,inicio,fim):
8
  pivo = vetor[fim]
9
  i = inicio-1
10

11
  for j in range(inicio,fim):
12
     if(vetor[j] <= pivo):
```

```
13
        i += 1
14
        vetor[i],vetor[j] = vetor[j],vetor[i]
15

16
    vetor[i+1],vetor[fim] = vetor[fim],vetor[i+1]
17
    return i+1,inicio,fim
```

Comparação com outros algoritmos de ordenação

Quicksort é uma versão optimizada de uma árvore binária ordenada. Em vez de introduzir itens sequencialmente numa árvore explicita, o Quicksort organiza-os correntemente na árvore onde está implícito, fazendo-o com chamadas recursivas à mesma. O algoritmo faz exactamente as mesmas comparações, mas com uma ordem diferente.

O algoritmo que mais se familiariza com o Quicksort é o Heapsort. Para o pior caso neste algoritmo temos
{\displaystyle {\mathcal {O}}(n\log 2n)}
. Mas, o Heapsort em média trata-se de um algoritmo mais lento que o Quicksort, embora essa afirmação já tenha sido muito debatida. No Quicksort permanece o caso do pior caso, à exceção quando se trata de usar a variante Intro sort, que muda para Heapsort quando um pior caso é detectado. Caso se saiba à partida que será necessário o uso do heapsort é aconselhável usá-lo directamente, do que usar o introsort e depois chamar o heapsort, torna mais rápido o algoritmo.

O Quicksort também compete com o Mergesort, outro algoritmo de ordenação recursiva, tendo este o benefício de ter como pior caso
{\displaystyle {\mathcal {O}}(n\log n)}
. Mergesort, ao contrário do Quicksort e do Heapsort, é estável e pode facilmente ser adptado para operar em listas encadeadas e em listas bastante grandes alojadas num tipo de acesso lento a média como um *Network-Attached Storage* ou num disco. Embora o Quicksort possa ser operado em listas encadeadas, por vezes escolhendo um mau pivô sem acesso aleatório. A maior desvantagem do Mergesort é que quando opera em *arrays*, requer
{\displaystyle {\mathcal {O}}(n)}
de espaço para o melhor caso, considerando que o Quicksort com um particionamento espacial e com recursão utiliza apenas
{\displaystyle {\mathcal {O}}(\log n)}

de espaço.

Bucket sort com dois *buckets* é muito parecido ao Quicksort (quase idêntico), o pivô neste caso é garantidamente o valor do meio do vector.

Referências

↑ AZEREDO, Paulo A. (1996). *Métodos de Classificação de Dados e Análise de suas Complexidades*. Rio de Janeiro: Campus. ISBN 85-352-0004-5

↑ «An Interview with C.A.R. Hoare» . Communications of the ACM, March 2009 ("premium content")

↑ BAASE, Sara (1988). *Computer Algorithms*. Introduction to Design and Analysis (em inglês) 2ª ed. Reading, Massachusetts: Addison-Wesley. 53 páginas. ISBN 0-201-06035-3

Ver também

Ordenação de vetor

Merge sort

Heapsort

Selection sort

Bubble sort

Busca linear

Ligações externas

Rápida aula de Quicksort

Animação do processo de ordenação pelo Quicksort

Explanação video de Quicksort usando cartões e de código em C++

QuickSort Code

Portal das tecnologias de informação

Última edição a 21 de maço de 2017, às12h43min