Mergesort
Quicksort

Prof. Tiago Massoni

Engenharia da Computação

Poli - UPE

Quicksort

- Proposto por Hoare em 1960 e publicado em 1962
- É o algoritmo de ordenação mais rápido que se conhece para uma ampla variedade de situações
  - Provavelmente é o mais utilizado.
- A idéia básica é dividir o problema de ordenar um conjunto com n itens em dois problemas menores
- Os problemas menores são ordenados independentemente
- Os resultados são combinados para produzir a solução final

Particionamento do Quicksort

- A parte mais delicada do método é relativa ao método particao
- O vetor v[esq..dir] é rearranjado por meio da escolha arbitrária de um pivô x
- O vetor v é particionado em duas partes
  - A parte esquerda com chaves menores ouiguais a x
  - A parte direita com chaves maiores ou iguais a x

Algoritmo quicksort

1. Escolha arbitrariamente um **pivô** x
2. Percorra o vetor a partir da esquerda até que v[i] ≥ x
3. Percorra o vetor a partir da direita até que v[j] ≤ x
4. Troque v[i] com v[j]
5. Continue este processo até os apontadores i e j se cruzarem

- Ao final, o vetor v[esq..dir] está particionado de tal forma que
  - Os itens em v[esq], v[esq + 1], . . . , v[j] são menores ou iguais a x.
  - Os itens em v[i], v[i + 1], . . . , v[dir] são maiores ou iguais a x.

Ilustração

1	2	3	4	5	6
O	R	**D**	E	N	A
A	R	**D**	E	N	O
A	**D**	R	E	N	O

- O pivô x é escolhido como sendo v[(i+j)/2]
  - Como inicialmente i = 1 e j = 6, então x = v[3] = D
- Ao final do processo de partição i e j se cruzam em i = 3 e j = 2

Algoritmos (todos private static)

```
class LimiteParticoes{
  int i; int j;
}

LimiteParticoes particao(Comparable v[],int
esq,int dir){
  LimiteParticoes p= new LimiteParticoes();
  p.i=esq;p.j=dir;
  Comparable x= v[(p.i+p.j)/2]; //pivo x
  do {
    while (x.compareTo(v[p.i])>0) p.i++;
    while (x.compareTo(v[p.j])<0) p.j--;
    if(p.i<=p.j){
      exchange(v[p.i],v[p.j]);
      p.i++; p.j--;
    }
  }while (p.i<=p.j);
  return p;
}
```

Algoritmos

```
void ordena (Comparable v[],int esq,int dir) {
 LimiteParticoes p= particao(v,esq,dir);
 if(esq<p.j) ordena(v,esq,p.j);
 if(p.i<dir) ordena(v,p.i,dir) ;
}


public static void quicksort(Comparable v[],int n){
  ordena (v, 1, n);
}
```

Chamadas ao particao e ordena

Chaves iniciais:	*O*	*R*	*D*	*E*	*N*	*A*
1	*A*	**D**	*R*	*E*	*N*	*O*
2	**A**	*D*				
3			**E**	*R*	*N*	*O*
4				**N**	*R*	*O*
5					*O*	**R**
	A	*D*	*E*	*N*	*O*	*R*

- Pivôs em negrito

Análise do quicksort

- Seja *C*(n) a função que conta o número de comparações
- Pior caso:
  - *C*(n) = *O*(n²)
- O pior caso ocorre quando, sistematicamente, o pivô é escolhido como sendo um dos extremos de um arquivo já ordenado
  - Isto faz com que o procedimento ordena seja chamado recursivamente n vezes, eliminando apenas um item em cada chamada
- O pior caso pode ser evitado empregando pequenas modificações no algoritmo
  - Para isso basta escolher três itens quaisquer do vetor e usar a mediana dos três como pivô

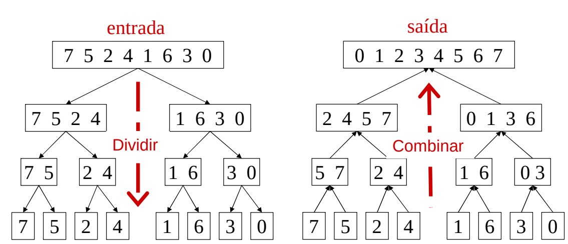
Análise do quicksort

- Melhor caso:
  - $C(n) = 2C(n/2) + n = n \log n - n + 1$
- Esta situação ocorre quando cada partição divide o arquivo em duas partes iguais
- Caso médio de acordo com Sedgewick e Flajolet
  - $C(n) \approx 1,386n \log n - 0,846n$
- Isso significa que em média o tempo de execução do Quicksort é O(n log n)

MergeSort

- Seja um array v de n elementos. O algoritmo consiste das seguintes fases
  - **Dividir** v em 2 sub-arrays de tamanho ≈ n/2
  - **Conquistar**: ordenar cada sub-array chamando MergeSort recursivamente
  - **Combinar** os sub-arrays ordenados formando um único array ordenado
- caso base: array com um elemento

MergeSort



Mergesort (considerando posição 1 primeiro)

```
void merge(Comparable v[],int b,int mid,int e){
  int n1= mid - b + 1;
  int n2= e - mid;
  Comparable [] L = new Comparable[n1+1];
  Comparable [] R = new Comparable[n2+1];

  for (int i=1;i<=n1;i++) L[i]= v[b+i-1];
  for (int j=1;i<=n2;i++) R[j]= v[mid+j];
  L[n1+1]= INFINITY; R[n2+1]= INFINITY;

  i=1;j=1;
  for(int k=b; k<=e; k++){
    if (L[i].compareTo(R[j]) <= 0){
      v[k]= L[i]; i++;
    }else{
      v[k]= R[j]; j++;
    }
}
```

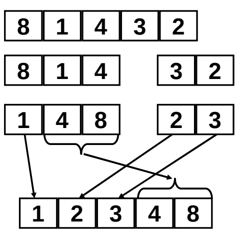
Mergesort

```
void mSort(Comparable v[],int b,int e){

  if (b<e){
      int mid=(b+e)/2;
      mSort(array,b,mid);
      mSort(array,mid+1,e);
      merge(array,b,mid,e);
  }
}


public static void mergesort(Comparable v[],int
n){
  mSort(v,0,n-1);
}
```

Mergesort



Análise do Merge

- Cada chamada mescla um total de n elementos
- Há três laços de repetição, sendo que cada iteração executa um número fixo de operações (que não depende de n)
- O total de iterações dos 2 primeiros laços não pode exceder n, já que cada iteração copia exatamente um elemento
- O total de iterações do terceiro laço não pode exceder n,
- Vemos então que no máximo 2n iterações são executadas no total e portanto o algoritmo é O(n)

Análise do mergesort

- Como o algoritmo opera dividindo os intervalos sempre por 2, um padrão pode emergir para valores de *n* iguais a potências de 2
- De fato observamos o seguinte quando consideramos o valor de $T(n) / n$:
  - $T(1) / 1=1$
  - $T(2) / 2=2$
  - $T(4) / 4=3$
  - $T(8) / 8=4$
  - $T(16) / 16=5$
  - ...
  - $T(2^k) / 2^k = k+1$
- Ou seja, para potências de 2, $T(n) / n = (\log_2 n) + 1$, ou
  $T(n) = (n \log_2 n) + n$