Ordenação por intercalação (mergeSort)

Professor: Mário César San Felice

Disciplina: Algoritmos e Estruturas de Dados 2

felice@ufscar.br

23 de setembro de 2019

Recapitulando

Definição de ordenação (crescente):

- Um vetor $v[0..n-1]$ está ordenado se $v[0] \leq v[1] \leq \cdots \leq v[n-1]$.

Problema da ordenação:

- Dado um vetor v de tamanho n, permutar os elementos de v até ele ficar ordenado.

Exemplo:

Entrada

0							7
77	55	11	44	33	22	88	66

Saída

0							7
11	22	33	44	55	66	77	88

Recapitulando

Vimos algoritmos iterativos com tempo $\mathrm{O}(n^2)$ no pior caso:

- insertionSort,
- selectionSort,
- bubbleSort.

Também vimos o heapSort, que usa uma estrutura de dados para atingir tempo $\mathrm{O}(n \log n)$.

Na aula de hoje:

- Técnica de projeto de algoritmos divisão-e-conquista.
- Algoritmo de ordenação mergeSort.
- Árvore de recursão para analisar eficiência de algoritmo recursivo.

Divisão-e-conquista

Uma das principais técnicas de projeto de algoritmos.

Apresenta três passos em cada nível da recursão:

Dividir o problema é dividido em subproblemas menores do mesmo tipo.

Conquistar os subproblemas são resolvidos recursivamente, sendo que os subproblemas pequenos (casos bases) são resolvidos diretamente.

Combinar as soluções dos subproblemas são combinadas numa solução do problema original.

Divisão-e-conquista e o mergeSort

Algoritmo usa abordagem não trivial para vencer a barreira do n^2.

Exemplo:

0							7
77	55	11	44	33	22	88	66

Dividir em dois subvetores.

0			3	4			7
77	55	11	44	33	22	88	66

Conquistar recursivamente (lembrar dos casos base).

0			3	4			7
11	44	55	77	22	33	66	88

Combinar por intercalação (merge).

0							7
11	22	33	44	55	66	77	88

Problema da intercalação

Dados $v[p..q-1]$ e $v[q..r-1]$ ordenados, obter $v[p..r-1]$ ordenado.

Exemplo:

vetor v

$i = p$			$q-1$	$j = q$			$r-1$
11	44	55	77	22	33	66	88

vetor w

$k = 0$							$r-p-1$

Exemplo: rotina de intercalação

A cada iteração o menor elemento é colocado em w.

Exemplo:

vetor v

i					j			
11	44	55	77		22	33	66	88

vetor w

k							

Exemplo: rotina de intercalação

A cada iteração o menor elemento é colocado em w.

Exemplo:

vetor v

i j

11	44	55	77	22	33	66	88

vetor w

k

11							

Exemplo: rotina de intercalação

A cada iteração o menor elemento é colocado em w.

Exemplo:

		i				j		
vetor v	11	44	55	77	22	33	66	88

			k					
vetor w	11	22						

Exemplo: rotina de intercalação

A cada iteração o menor elemento é colocado em w.

Exemplo:

vetor v

	i						j	
11	44	55	77		22	33	66	88

vetor w

			k				
11	22	33					

Exemplo: rotina de intercalação

A cada iteração o menor elemento é colocado em w.

Exemplo:

vetor v

		i	
11	44	55	77

		j	
22	33	66	88

vetor w

				k			
11	22	33	44				

Exemplo: rotina de intercalação

A cada iteração o menor elemento é colocado em w.

Exemplo:

vetor v

			i				j	
11	44	55	77		22	33	66	88

vetor w

					k		
11	22	33	44	55			

Exemplo: rotina de intercalação

A cada iteração o menor elemento é colocado em w.

Exemplo:

			i					j
vetor v	11	44	55	77	22	33	66	88

							k	
vetor w	11	22	33	44	55	66		

Exemplo: rotina de intercalação

A cada iteração o menor elemento é colocado em w.

Exemplo:

vetor v

				i			j
11	44	55	77	22	33	66	88

vetor w

							k
11	22	33	44	55	66	77	

Exemplo: rotina de intercalação

A cada iteração o menor elemento é colocado em w.

Exemplo:

vetor v

					i				j
11	44	55	77		22	33	66	88	

vetor w

								k
11	22	33	44	55	66	77	88	

Código: rotina de intercalação

Intercala subvetores ordenados $v[p..q-1]$ e $v[q..r-1]$.

```
void intercala (int *v, int p, int q, int r) {



}
```

Código: rotina de intercalação

Inicializa variáveis para percorrer os vetores.

```
void intercala (int *v, int p, int q, int r) {
    int i = p, j = q, k = 0; int tam = r-p;
















}
```

Código: rotina de intercalação

Aloca vetor auxiliar.

```
void intercala (int *v, int p, int q, int r) {
    int i = p, j = q, k = 0; int tam = r-p;
    int * w = malloc(tam*sizeof(int));




}
```

Código: rotina de intercalação

Enquanto vetor auxiliar não estiver completo.

```
void intercala (int *v, int p, int q, int r) {
    int i = p, j = q, k = 0; int tam = r-p;
    int * w = malloc(tam*sizeof(int));

    while (k < tam) {


    }



}
```

Código: rotina de intercalação

Copia o menor elemento dentre os subvetores.

```
void intercala (int *v, int p, int q, int r) {
    int i = p, j = q, k = 0; int tam = r-p;
    int * w = malloc(tam*sizeof(int));

    while (k < tam) {
        if (v[i] <= v[j]) w[k++] = v[i++];

    }



}
```

Código: rotina de intercalação

Copia o menor elemento dentre os subvetores.

```
void intercala (int *v, int p, int q, int r) {
    int i = p, j = q, k = 0; int tam = r-p;
    int * w = malloc(tam*sizeof(int));

    while (k < tam) {
        if (v[i] <= v[j]) w[k++] = v[i++];
        else /* v[i] > v[j] */ w[k++] = v[j++];
    }



}
```

Código: rotina de intercalação

Copia os elementos em ordem para o vetor original.

```
void intercala (int *v, int p, int q, int r) {
    int i = p, j = q, k = 0; int tam = r-p;
    int * w = malloc(tam*sizeof(int));

    while (k < tam) {
        if (v[i] <= v[j]) w[k++] = v[i++];
        else /* v[i] > v[j] */ w[k++] = v[j++];
    }


    for (k = 0; k < tam; k++)
        v[p+k] = w[k];
    free(w);
}
```

Código: rotina de intercalação

O que acontece se $i \geq q$ ou $j \geq r$ no primeiro laço?

```
void intercala (int *v, int p, int q, int r) {
    int i = p, j = q, k = 0; int tam = r-p;
    int * w = malloc(tam*sizeof(int));

    while (k < tam) {
        if (v[i] <= v[j]) w[k++] = v[i++];
        else /* v[i] > v[j] */ w[k++] = v[j++];
    }


    for (k = 0; k < tam; k++)
        v[p+k] = w[k];
    free(w);
}
```

Código: rotina de intercalação

O que acontece se $i \geq q$ ou $j \geq r$ no primeiro laço?

```
void intercala (int *v, int p, int q, int r) {
    int i = p, j = q, k = 0; int tam = r-p;
    int * w = malloc(tam*sizeof(int));

    while (i < q && j < r) {
        if (v[i] <= v[j]) w[k++] = v[i++];
        else /* v[i] > v[j] */ w[k++] = v[j++];
    }


    for (k = 0; k < tam; k++)
        v[p+k] = w[k];
    free(w);
}
```

Código: rotina de intercalação

E se sobrarem elementos em um dos subvetores?

```
void intercala (int *v, int p, int q, int r) {
    int i = p, j = q, k = 0; int tam = r-p;
    int * w = malloc(tam*sizeof(int));

    while (i < q && j < r) {
        if (v[i] <= v[j]) w[k++] = v[i++];
        else /* v[i] > v[j] */ w[k++] = v[j++];
    }


    for (k = 0; k < tam; k++)
        v[p+k] = w[k];
    free(w);
}
```

Código: rotina de intercalação

Copia os elementos que sobraram para o final de *w*.

```
void intercala (int *v, int p, int q, int r) {
    int i = p, j = q, k = 0; int tam = r-p;
    int * w = malloc(tam*sizeof(int));

    while (i < q && j < r) {
        if (v[i] <= v[j]) w[k++] = v[i++];
        else /* v[i] > v[j] */ w[k++] = v[j++];
    }
    while (i < q) w[k++] = v[i++];
    while (j < r) w[k++] = v[j++];
    for (k = 0; k < tam; k++)
        v[p+k] = w[k];
    free(w);
}
```

Análise de corretude: rotina de intercalação

Exercício: mostrar a corretude da rotina de intercalação.

Dica: Provar por indução usando os seguintes invariantes.

No início de cada iteração temos que:

- $w[0..k-1]$ contém os elementos de $v[p..i-1]$ e $v[q..j-1]$,
- $w[0..k-1]$ está ordenado.
- $w[h] \leq v[l]$ para $0 \leq h < k$ e $i \leq l < q$.
- $w[h] \leq v[l]$ para $0 \leq h < k$ e $j \leq l < r$.

Estrutura da prova por indução:

Caso base mostrar que vale quando $k = 0$.

Hipótese de Indução o próprio invariante para $k' < k$.

Passo mostrar, usando a H.I., que o comportamento do algoritmo preserva o invariante na iteração k.

Análise de eficiência: rotina de intercalação

O número de operações é proporcional ao tamanho do vetor,

ou seja, $\mathrm{O}(tam) = \mathrm{O}(r - p)$.

Isso pode não parecer evidente por conta dos vários laços do algoritmo.

No entanto, basta perceber que em cada iteração, de qualquer laço, i ou j são incrementados.

Como i é sempre menor que q e j é sempre menor que r,

temos no máximo $(q - p) + (r - q)$ iterações.

Como $(q - p) + (r - q) = r - p$, o resultado segue.

Divisão-e-conquista e o mergeSort, o retorno

Sabendo como a intercalação funciona, voltamos ao mergeSort.

Exemplo:

0							7
77	55	11	44	33	22	88	66

Dividir em dois subvetores.

0			3
77	55	11	44

4			7
33	22	88	66

Conquistar recursivamente (lembrar dos casos base).

0			3
11	44	55	77

4			7
22	33	66	88

Combinar por intercalação (merge).

0							7
11	22	33	44	55	66	77	88

Código: mergeSort

Ordena os elementos do vetor v entre as posições p e $r - 1$.

```
void mergeSort (int *v, int p, int r) {




}
```

Código: mergeSort

Dividir: Encontra o meio do vetor.

```
void mergeSort (int *v, int p, int r) {
    int m;

        m = (p + r) / 2;




}
```

Código: mergeSort

Notem que $(p + r)/2 = p + (r - p)/2 = p + tam/2$.

```
void mergeSort (int *v, int p, int r) {
    int m;

        m = (p + r) / 2;



}
```

Código: mergeSort

Conquistar: chamadas recursivas nos dois subvetores.

```
void mergeSort (int *v, int p, int r) {
    int m;

        m = (p + r) / 2;
        mergeSort (v, p, m);
        mergeSort (v, m, r);


}
```

Código: mergeSort

Combinar: intercala os dois subvetores ordenados.

```
void mergeSort (int *v, int p, int r) {
    int m;

        m = (p + r) / 2;
        mergeSort (v, p, m);
        mergeSort (v, m, r);
        intercala (v, p, m, r);

}
```

Código: mergeSort

Esse algoritmo para?

```
void mergeSort (int *v, int p, int r) {
    int m;

        m = (p + r) / 2;
        mergeSort (v, p, m);
        mergeSort (v, m, r);
        intercala (v, p, m, r);

}
```

Código: mergeSort

Tratar casos base em que o vetor tem tamanho 0 ou 1, i.e., $r - p \leq 1$.

```
void mergeSort (int *v, int p, int r) {
    int m;
    if (r - p > 1) {
        m = (p + r) / 2;
        mergeSort (v, p, m);
        mergeSort (v, m, r);
        intercala (v, p, m, r);
    }
}
```

Código: mergeSort

Para ordenar o vetor v inteiro chamar a função com $p = 0$ e $r = n$.

```
void mergeSort (int *v, int p, int r) {
    int m;
    if (r - p > 1) {
        m = (p + r) / 2;
        mergeSort (v, p, m);
        mergeSort (v, m, r);
        intercala (v, p, m, r);
    }
}
```

Código: mergeSort

Bônus: cálculo levemente diferente de m para evitar erro numérico.

```
void mergeSort (int *v, int p, int r) {
    int m;
    if (r - p > 1) {
        // m = (p + r) / 2;
        m = p + (r - p) / 2;
        mergeSort (v, p, m);
        mergeSort (v, m, r);
        intercala (v, p, m, r);
    }
}
```

Exemplo: mergeSort

Chamadas recursivas em paralelo para facilitar a compreensão.

Início: 1 vetor de tamanho 8.

0							7
77	55	11	44	33	22	88	66

Divisão: 2 vetores de tamanho 4.

0			3
77	55	11	44

4			7
33	22	88	66

Divisão: 4 vetores de tamanho 2.

0	1
77	55

2	3
11	44

4	5
33	22

6	7
88	66

Divisão: 8 vetores de tamanho 1.

0
77

1
55

2
11

3
44

4
33

5
22

6
88

7
66

Caso base: vetores com tamanho 1.

Exemplo: mergeSort

Chamadas recursivas em paralelo para facilitar a compreensão.

Caso base: chamadas recursivas voltam.

0	1	2	3	4	5	6	7
77	55	11	44	33	22	88	66

Combinar: intercalação ordena 4 vetores de tamanho 2.

0	1	2	3	4	5	6	7
55	77	11	44	22	33	66	88

Combinar: intercalação ordena 2 vetores de tamanho 4.

0			3	4			7
11	44	55	77	22	33	66	88

Combinar: intercalação ordena 1 vetor de tamanho 8.

0							7
11	22	33	44	55	66	77	88

Fim: o vetor inteiro foi ordenado.

Análise de corretude: mergeSort

Usamos indução para mostrar que mergeSort ordena um vetor de tamanho $n = r - p$.

Caso base: ocorre quando o vetor tem tamanho 0 ou 1, já estando ordenado. Nestes casos $r - p \leq 1$ e o algoritmo termina.

Hipótese de Indução: o algoritmo ordena corretamente vetores de tamanho menor que $n = r - p$.

Passo: quando o algoritmo recebe um vetor de tamanho n, o divide em dois subvetores menores. Pela H.I. sabemos que os subvetores são ordenados corretamente. Finalmente, como a rotina de intercalação funciona, obtemos um vetor ordenado de tamanho n.

Curiosidade: esta demonstração não usa o fato do mergeSort dividir o vetor ao meio.

Análise de eficiência: mergeSort

No pior caso o mergeSort leva tempo proporcional a $\mathrm{O}(n \log n)$.

Dicas para a análise:

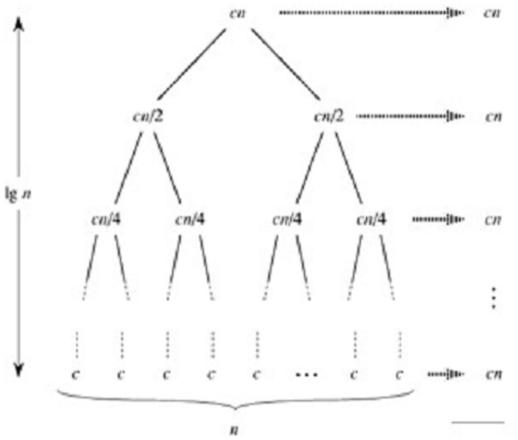
- Divida o trabalho realizado pelo mergeSort em local e recursivo.
- Note que a função $T(n) = 2T(n/2) + cn$ captura este trabalho.
- Construa uma árvore binária de recursão a partir de $T(n)$.

Questões:

- Qual o número de níveis desta árvore?
- Qual o número de subproblemas no nível j da árvore?
- Qual o tamanho de cada subproblema do nível j da árvore?

Curiosidade: a técnica da árvore de recursão generaliza para o Teorema Mestre.

Análise de eficiência: mergeSort



Características adicionais

Estabilidade:

- Ordenação é estável. Podemos mostrar isso usando indução e o fato da rotina de intercalação ser estável.

Eficiência de espaço:

- Ordenação não é in place, pois usa a rotina intercala que precisa de vetor auxiliar (e portanto memória) proporcional ao tamanho dos vetores sendo intercalados.

Curiosidade: podemos usar o algoritmo insertionSort como caso base do mergeSort.

- Isso traz vantagem pois o insertionSort tem constante menor que o mergeSort, sendo por isso mais rápido quando n é pequeno.

Comparação de funções

Quão felizes devemos ficar com a melhoria que obtivemos?

Considere ordenar vetores num computador que faz 10 bilhões de operações por segundo ($10GHz$).

Quanto tempo ele leva para ordenar vetores de tamanho n?

n	$\log n$	$n \log n$	tempo	n^2	tempo
10^3 ou $1K$	10	$10K$	$< 1s$	10^6 ou $1M$	$< 1s$
10^6 ou $1M$	20	$20M$	$< 1s$	10^{12} ou 10^3G	100s
10^9 ou $1G$	30	$30G$	$3s$	10^{18} ou 10^9G	$3,17$ anos

Vídeo com algoritmos de ordenação:
https://www.youtube.com/watch?v=ZZuD6iUe3Pc.

Bônus: mergeSort iterativo

```
void mergeSortI(int v[], int n) {
    int b = 1;
    while (b < n) {
        int p = 0;
        while (p + b < n) {
            int r = p + 2 * b;
            if (r > n)
                r = n;
            intercala(v, p, p + b, r);
            p = p + 2 * b;
        }
        b = 2 * b;
    }
}
```

Cenas dos próximos capítulos:

Na próxima aula: divisão-e-conquista junto com aleatoriedade para chegar ao mais rápido algoritmo de ordenação baseado em comparações, o quickSort.

Num tópico relacionado: para fazer ordenação externa são generalizadas as ideias do mergeSort e do intercala.