Análise e Síntese de Algoritmos

Revisão [CLRS, Cap. 7-10]

Contexto

- Revisão [CLRS, Cap.1-13]
  - Fundamentos; notação; exemplos
- Algoritmos em Grafos [CLRS, Cap.21-26]
  - Algoritmos elementares
  - Caminhos mais curtos
  - Fluxos máximos
  - Árvores abrangentes
- Técnicas de Síntese de Algoritmos [CLRS, Cap.15-16]
  - Programação dinâmica
  - Algoritmos greedy
- Programação Linear [CLRS, Cap.29]
  - Algoritmos e modelação de problemas com restrições lineares
- Tópicos Adicionais [CLRS, Cap.32-35]
  - Emparelhamento de Cadeias de Caracteres
  - Complexidade Computacional
  - Algoritmos de Aproximação

Resumo

QuickSort - Pseudo-Código

QuickSort

```
QuickSort(A, p, r)
if p < r
    then q = Partition(A, p, r)
         QuickSort(A, p, q-1)
         QuickSort(A, q+1, r)

Partition(A, p, r)
x = A[r]
i=p-1
for j = p to r-1
    do if A[j] ≤ x
        then i = i + 1
             swap(A[i], A[j])
swap(A[i+1], A[j])
return i+1
```

QuickSort vs. MergeSort

QuickSort

- Vector não necessariamente dividido em 2 partes iguais
- Constantes menores
- Pior caso (vector ordenado): $O(n^2)$

MergeSort

- Vector dividido em 2 partes iguais
- Necessário fazer Merge (constantes maiores)
- Pior caso: $O(n \; lgn)$

Na prática: QuickSort (aleatorizado) é normalmente mais rápido

CountingSort

- Chaves: inteiros de 1 a k
- Contar ocorrências de cada chave, e inserir na região respectiva do vector
- Complexidade: $O(n+k) = O(n)$, se $k = O(n)$

CountingSort

```
CountingSort(A)
for i = 1 to k
        do C[i] = 0
for j = 1 to length[A]
        do C[A[j]]++
for l = 2 to k
        do C[l] = C[l] + C[l-1]
for j = length[A] downto 1
        do B[C[A[j]]] = A[j]
           C[A[j]]- -
```

CountingSort

$N = 18,\ M = 7$

`int a[]`

5	2	1	6	5	3	3	4	0	1	2	4	6	0	4	6	3	1

`int cnt[M+1]`

`int b[]`

CountingSort

$N = 18, \; M = 7$

`int a[]`

5	2	1	6	5	3	3	4	0	1	2	4	6	0	4	6	3	1

`int cnt[M+1]`

0	0	0	0	0	0	0	0

`int b[]`

CountingSort

$N = 18, \; M = 7$

int a[]

5	2	1	6	5	3	3	4	0	1	2	4	6	0	4	6	3	1

int cnt[M+1]

0	2	3	2	3	3	2	3

```
for (i = l; i <= r; i++)
  cnt[a[i]+1]++;
```

int b[]

CountingSort

$N = 18, \ M = 7$

`int a[]`

5	2	1	6	5	3	3	4	0	1	2	4	6	0	4	6	3	1

`int cnt[M+1]`

0	2	5	7	10	13	15	18

```
for (j = 1; j < M; j++)
    cnt[j] += cnt[j-1];
```

`int b[]`

CountingSort

$N = 18, \ M = 7$

`int a[]`

5	2	1	6	5	3	3	4	0	1	2	4	6	0	4	6	3	1

`int cnt[M+1]`

0	2	5	7	10	13	15	18

```
for (i = l; i <= r; i++)
  b[cnt[a[i]]++] = a[i];
```

`int b[]`

													5				

CountingSort

$N = 18,\ M = 7$

int a[]

5	2	1	6	5	3	3	4	0	1	2	4	6	0	4	6	3	1

int cnt[M+1]

0	2	5	7	10	13	15	18

```
for (i = l; i <= r; i++)
  b[cnt[a[i]]++] = a[i];
```

int b[]

													5				

CountingSort

$N = 18, \; M = 7$

int a[]

5	2	1	6	5	3	3	4	0	1	2	4	6	0	4	6	3	1

int cnt[M+1]

0	2	5	7	10	14	15	18

```
for (i = l; i <= r; i++)
  b[cnt[a[i]]++] = a[i];
```

int b[]

													5				

CountingSort

$N = 18,\ M = 7$

int a[]

5	2	1	6	5	3	3	4	0	1	2	4	6	0	4	6	3	1

int cnt[M+1]

2	5	7	10	13	15	18	18

```
for (i = l; i <= r; i++)
  b[cnt[a[i]]++] = a[i];
```

int b[]

0	0	1	1	1	2	2	3	3	3	4	4	4	5	5	6	6	6

Copiar os elementos de b para a!

RadixSort

RadixSort

RadixSort(A, d)

1 **for** i = 1 **to** d

2 **do** Ordenar A no dígito i com algoritmo de ordenação estável

Se utilizar CountingSort

- Complexidade: $O(d(n+k))$
- Não ordena no lugar, i.e. requer memória adicional

MSD RadixSort

for	ace	ace	ace
tip	ago	ago	ago
ilk	cow	cow	cow
tar	for	fee	fee
ace	fee	for	for
fee	ilk	ilk	ilk
ago	tip	tar	tag
cow	tar	tag	tar
tag	tag	tea	tea
tea	tea	tip	tip
wee	wee	wee	wee

LSD RadixSort

for	tea	tag	ace
tip	ace	tar	ago
ilk	fee	ace	cow
tar	wee	tea	fee
ace	tag	fee	for
fee	ilk	wee	ilk
ago	ago	ago	tag
cow	tip	tip	tar
tag	for	ilk	tea
tea	tar	for	tip
wee	cow	cow	wee

Pesquisa Linear

Procurar elemento em vector (ou lista)

- Vector ou lista podem não estar ordenados
- Complexidade: $O(n)$

Pesquisa Linear

```
LinearSearch(A, key)
for i = 1 to length[A]
    do if A[i] = key
        then return i
return 0
```

Pesquisa Binária

Procurar elemento em vector

- Vector tem que estar ordenado
- Complexidade: $O(lgn)$

Pesquisa Binária

```
BinarySearch(A, l, r, key)
if l <= r
    then m = ⌊ (l + r) / 2 ⌋
        if A[m] = key
            then return m
        if A[m] < key
            then return BinarySearch(A, m+1, r, key)
            else  return BinarySearch(A, l, m-1, key)
return 0
```

Encontrar Máximo e Mínimo

Qual o número de comparações para encontrar valor mínimo em vector não ordenado?

- $n - 1$ comparações para percorrer todo o vector. Valor óptimo.

Encontrar Máximo e Mínimo

Qual o número de comparações para encontrar valor mínimo em vector não ordenado?

- $n - 1$ comparações para percorrer todo o vector. Valor óptimo.

Qual o número de comparações para encontrar valor máximo em vector não ordenado?

- $n - 1$ comparações para percorrer todo o vector. Valor óptimo.

Encontrar Máximo e Mínimo

Qual o número de comparações para encontrar valor mínimo em vector não ordenado?

- $n-1$ comparações para percorrer todo o vector. Valor óptimo.

Qual o número de comparações para encontrar valor máximo em vector não ordenado?

- $n-1$ comparações para percorrer todo o vector. Valor óptimo.

Qual o número de comparações para encontrar conjuntamente valor máximo e mínimo em vector não ordenado?

- $2n-2$ comparações, se selecção é realizada de forma independente
- $3\lfloor n/2 \rfloor$, se elementos analisados aos pares, com máximo e mínimo seleccionados conjuntamente

Estruturas de Dados Elementares

- Amontoados
- Pilhas, Filas
- Listas Ligadas
  - Simplesmente ligadas
  - Duplamente ligadas
- Árvores binárias
  - Árvores equilibradas

Operações sobre vectores e listas

	Aceder/Modificar		Inserir		Remover	
Posição	Vec	LList*	Vec	LList*	Vec	LList*
Primeira	$O(1)$	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(1)$
Última	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(1)$	$O(n)$
Específica	$O(1)$	$O(k)$	$O(n-k)$	$O(k)$	$O(n-k)$	$O(k)$
Próxima	$O(1)$	$O(1)$	$O(n-k)$	$O(1)$	$O(n-k)$	$O(1)^\dagger$

*Lista simplesmente ligada em que apenas é conhecida uma referência para o início.

†É necessário manter uma referência para o elemento anterior.

k é a posição do elemento em questão.

Problema

Como implementar uma fila (FIFO) utilizando uma fila com prioridade, i.e. usando operações sobre amontoados?

- Primeiro elemento inserido é sempre o primeiro retirado
- Operações a implementar: queue(value) e dequeue()

Solução

- Topo do amontoado é o valor mínimo
- Manter contador do número total de elementos inseridos
- Cada elemento inserido no amontoado é caracterizado pelo número de elementos inseridos antes desse elemento
- Operação queue corresponde a Insert(). Complexidade: $O(lgn)$
- Operação dequeue corresponde a ExtractMin(). Complexidade: $O(lgn)$

Min e ExtractMin para MinHeaps

Min

Min(A)

```
return A[1]                          ▷ Complexidade: O(1)
```

ExtractMin

ExtractMin(A)

```
min = A[1]
A[1] = A[heap_size[A]]
heap_size[A]−−
SiftDown(A, 1)
return min                           ▷ Complexidade: O(lgn)
```

Insert e SiftUp para MinHeaps

Insert

```
Insert(A, key)
heap_size[A] ++
i = heap_size[A]
A[i] = key
SiftUp(A, i)                                ▷ Complexidade: O(lgn)
```

SiftUp

```
SiftUp(A, i)
j = Parent(i)
if j > 0 and A[j] > A[i]
   then swap(A[i], A[j])
        SiftUp(A, j)                        ▷ Complexidade: O(lgn)
```

Amontoados e filas com prioridade

	Binary	*Binomial*	*Fibonacci*	*Rank relaxed*	*Run relaxed*
Insert	$O(\log n)$	$O(\log n)$	$O(1)$	$O(1)$	$O(1)$
Minimum	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Extract-Min	$O(\log n)$	$O(\log n)$	$O(\log n)^*$	$O(\log n)$	$O(\log n)$
Union	$O(n)$	$O(\log n)$	$O(1)$	$O(\log n)$	$O(\log n)^\dagger$
Decrease-Key	$O(\log n)$	$O(\log n)$	$O(1)^*$	$O(1)^*$	$O(1)$
Delete	$O(\log n)$	$O(\log n)$	$O(\log n)^*$	$O(\log n)$	$O(\log n)$

*Tempo amortizado.

†Brodal propôs uma implementação de filas com prioridade em 1996 que permite ainda a operação *union* em tempo $O(1)$, mas de implementação bastante difícil.

Outro problema

Considere uma matriz de dimensão $n \times n$ caracterizada por:

- Cada entrada na matriz tem valor 0 ou 1
- Todas as linhas são monotonamente crescentes
- Todas as colunas são monotonamente crescentes

Objectivo: Defina um algoritmo eficiente para contar número de ocorrências de 0 e 1 na matriz