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

- Algoritmos Ordenação
 - QuickSort
 - CountingSort
 - RadixSort
- Procura e Selecção
- Struturas de Dados Elementares

QuickSort - Pseudo-Código

QuickSort

```
QuickSort(A, p, r)
   if p < r
      then q = Partition(A, p, r)
3
            QuickSort(A, p, q-1)
4
            QuickSort(A, q+1, r)
Partition(A, p, r)
   x = A[r]
  i=p-1
3
   for j = p to r-1
4
         do if A[i] \leq x
5
               then i = i + 1
6
                     swap(A[i], A[j])
   swap(A[i+1], A[i])
8
   return i+1
```

QuickSort vs. MergeSort

QuickSort

- Vector n\u00e3o 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

6

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(A)
   for i = 1 to k
```

```
do C[i] = 0
```

3 **for**
$$j = 1$$
 to length[A]

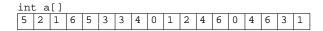
5 **for**
$$I = 2$$
 to k

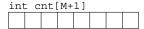
do
$$C[I] = C[I] + C[I-1]$$

7 **for**
$$j = length[A]$$
 downto 1

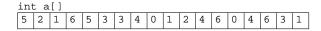
8 **do**
$$B[C[A[j]]] = A[j]$$

$$N = 18, M = 7$$





$$N = 18, M = 7$$



$$N = 18, M = 7$$



$$N = 18, M = 7$$

$$N = 18, M = 7$$

Copiar os elementos de b para a!

RadixSort.

RadixSort

RadixSort(A, d)

for i = 1 to d

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	a	ce	а	ce	ac	e
tip	a	go	а	go	ag	0
ilk	C	WO	С	OW	CO	W
tar	f	or	f	<mark>e</mark> e	fe	e
ace	f	ee	f	<mark>o</mark> r	fo	r
fee	i	lk	i	<mark>l</mark> k	il	k
ago	t	ip	t	<mark>a</mark> r	ta	9
COW	t	ar	t	<mark>a</mark> g	ta	r
tag	t	ag	t	<mark>e</mark> a	te	a
tea	t	ea	t	<mark>i</mark> p	ti	р
wee	W	ee	W	<mark>e</mark> e	we	e

LSD RadixSort

for	te <mark>a</mark>	t <mark>a</mark> g	<mark>a</mark> ce
tip	ac <mark>e</mark>	t <mark>a</mark> r	<mark>a</mark> go
ilk	fe <mark>e</mark>	a <mark>c</mark> e	<mark>C</mark> OW
tar	we <mark>e</mark>	t <mark>e</mark> a	<mark>f</mark> ee
ace	ta <mark>g</mark>	f <mark>e</mark> e	for
fee	il <mark>k</mark>	w <mark>e</mark> e	<mark>i</mark> lk
ago	ag <mark>o</mark>	a <mark>g</mark> o	<mark>t</mark> ag
COW	ti <mark>p</mark>	t <mark>i</mark> p	<mark>t</mark> ar
tag	fo <mark>r</mark>	i <mark>l</mark> k	<mark>t</mark> ea
tea	ta <mark>r</mark>	f <mark>o</mark> r	<mark>t</mark> ip
wee	CO <mark>W</mark>	C <mark>O</mark> W	<mark>w</mark> ee

Procurar elemento em vector (ou lista)

- Vector ou lista podem n\u00e3o estar ordenados
- Complexidade: O(n)

Pesquisa Linear

```
LinearSearch(A, key)
```

- 1 for i = 1 to length[A]
- 2 **do if** A[i] = key
- 3 **then** return i
- 4 return 0

Procurar elemento em vector

- Vector tem que estar ordenado
- Complexidade: O(Ign)

Pesquisa Binária

```
\begin{array}{lll} BinarySearch(A,\ I,\ r,\ key) \\ 1 & \textbf{if}\ I <= r \\ 2 & \textbf{then}\ m = \lfloor\ (I+r)\ /\ 2\ \rfloor \\ 3 & \textbf{if}\ A[m] = key \\ 4 & \textbf{then return}\ m \\ 5 & \textbf{if}\ A[m] < key \\ 6 & \textbf{then return}\ BinarySearch(A,\ m+1,\ r,\ key) \\ 7 & \textbf{else return}\ BinarySearch(A,\ I,\ m-1,\ key) \\ 8 & \textbf{return}\ 0 \end{array}
```

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?

ullet 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?

- \bullet 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

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

Operações sobre vectores e listas

	Aceder/Modificar		Inser	ir	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.

k é a posição do elemento em questão.

[†]É necessário manter uma referência para o elemento anterior.

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(Ign)
- Operação dequeue corresponde a ExtractMin(). Complexidade: O(lgn)

Min e ExtractMin para MinHeaps

Min

Min(A)

1 return A[1]

 \triangleright Complexidade: O(1)

ExtractMin

ExtractMin(A)

- $1 \quad \min = A[1]$
- $2 \quad A[1] = A[heap_size[A]]$
- 3 heap_size[A]— —
- 4 SiftDown(A, 1)
- 5 **return** min

 \triangleright Complexidade: O(lgn)

Insert e SiftUp para MinHeaps

Insert

```
Insert(A, key)
1  heap_size[A] ++
2  i = heap_size[A]
3  A[i] = key
```

4 SiftUp(A, i)

 \triangleright Complexidade: O(lgn)

SiftUp

```
\begin{array}{ll} SiftUp(A, i) \\ 1 & j = Parent(i) \\ 2 & \textbf{if} \ j > 0 \ and \ A[j] > A[i] \\ 3 & \textbf{then} \ swap(A[i], A[j]) \\ 4 & SiftUp(A, j) \end{array}
```

 \triangleright 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