Análise de Algoritmos

2008

Paulo Feofiloff

IME, Universidade de São Paulo

Aula 1: Introdução

CLRS: Cormen, Leiserson, Rivest, Stein

- "Having a solid base of algorithmic knowledge and technique is one characteristic that separates the truly skilled programmers from the novices. With modern computing technology, you can accomplish some tasks without knowing much about algorithms, but with a good background in algorithms, you can do much, much more."

- "Uma base sólida de conhecimento e técnica de algoritmos é uma das características que separa o programador experiente do aprendiz. Com a moderna tecnologia de computação, você pode realizar algumas tarefas sem saber muito sobre algoritmos, mas com um boa base em algoritmos você pode fazer muito, muito mais."

O que é AA? Um exemplo

Problema: Rearranjar um vetor em ordem crescente

$A[1\,..\,n]$ é crescente se $A[1] \leq \cdots \leq A[n]$

22	33	33	33	44	55	11	99	22	55	77

11	22	22	33	33	33	44	55	55	77	99

Algoritmo: Rearranja $A[1\,..\,n]$ em ordem crescente

ORDENA-POR-INSERÇÃO (A, n)

1 para $j \leftarrow 2$ até n faça
2 $\quad$ $chave \leftarrow A[j]$
3 $\quad$ $i \leftarrow j - 1$
4 $\quad$ enquanto $i \geq 1$ e $A[i] > chave$ faça
5 $\quad\quad$ $A[i+1] \leftarrow A[i]$
6 $\quad\quad$ $i \leftarrow i - 1$
7 $\quad$ $A[i+1] \leftarrow chave$

1						j				n
22	33	33	33	44	55	11	99	22	55	77

Note a documentação

Análise da correção: O algoritmo faz o que prometeu?

- Invariante: no início de cada iteração, $A[1\,..\,j{-}1]$ é crescente
- Se vale na última iteração, o algoritmo está correto!

ORDENA-POR-INSERÇÃO (A, n)

1 para $j \leftarrow 2$ até (*) n faça
2 $\quad$ $chave \leftarrow A[j]$
3 $\quad$ $i \leftarrow j - 1$
4 $\quad$ enquanto $i \geq 1$ e $A[i] > chave$ faça
5 $\quad\quad$ $A[i+1] \leftarrow A[i]$
6 $\quad\quad$ $i \leftarrow i - 1$
7 $\quad$ $A[i+1] \leftarrow chave$

1						j				n
22	33	33	33	44	55	11	99	22	55	77

- vale na primeira iteração
- se vale em uma iteração, vale na seguinte

Análise do desempenho: Quanto tempo consome?

- Regra de três
- Regra de três não funciona!
- Suponha 1 unidade de tempo por linha

linha	total de unidades de tempo			
1	$=$	n		
2	$=$	$n-1$		
3	$=$	$n-1$		
4	$\leq$	$2+3+\cdots+n$	$=$	$(n-1)(n+2)/2$
5	$\leq$	$1+2+\cdots+(n{-}1)$	$=$	$n(n-1)/2$
6	$\leq$	$1+2+\cdots+(n{-}1)$	$=$	$n(n-1)/2$
7	$=$	$n-1$		
total	$\leq$	$\frac{3}{2}n^2 + \frac{7}{2}n - 4$ unidades de tempo		

- Algoritmo consome $\leq \frac{3}{2}n^2 + \frac{7}{2}n - 4$ unidades de tempo
- $\frac{3}{2}$ é pra valer?

  Não, pois depende do computador
- n^2 é pra valer?

  Sim, pois só depende do algoritmo
- Queremos um resultado que só dependa do algoritmo

Outra tática:

- número de comparações "$A[i] > chave$"

  $\leq\ 2 + 3 + \cdots + n\ =\ \frac{1}{2}(n-1)(n+2)\ \leq\ \frac{1}{2}n^2 + \frac{1}{2}n - 1$
- $\frac{1}{2}$ é pra valer?

  Não, pois depende do computador
- n^2 é pra valer?

  Sim, pois só depende do algoritmo
- Queremos resultado que só dependa do algoritmo

- "n^2" é informação valiosa
- mostra evolução do tempo com n:

n	n^2
$2n$	$4n^2$
$10n$	$100n^2$

AA "in a nutshell"

- Problema
- Instâncias do problema
- Tamanho de uma instância
- Algoritmo
- Análise da correção do algoritmo
- Análise do desempenho (velocidades) do algoritmo
- Desempenho em função do tamanho das instâncias
- Pior caso
- Evolução do consumo de tempo em função de n
- Independente do computador e da implementação

Aula 2

Notação O

Notação O: comparação assintótica de funções

- Comparação assintótica grosseira de funções $f(n)$ e $g(n)$
- Qual das duas cresce mais rápido?
- Algo com sabor de $f(n)$ "$\leq$" $g(n)$
- Despreze valores pequenos de n: $n \to \infty$
- Despreze constantes multiplicativas: $100\,n^2$ "$\leq$" n^2
- Despreze termos de ordem inferior: $n^2 + 10n + 10$ "$\leq$" $2\,n^2$

$f(n)$ e $g(n)$ assintoticamente não-negativas

Definição: Dizemos que $f = \mathrm{O}(g)$ se existem constantes $c > 0$ e $N > 0$ tais que $f(n) \leq c \cdot g(n)$ para todo $n \geq N$

Em outras palavras: para todo n suficientemente grande, $f(n)$ é dominado por um múltiplo de $g(n)$

Comentários:

- constante = não dependende de n
- "$f \in \mathrm{O}(g)$" seria mais correto

Exemplo 1

$n^2 + 10n = \mathrm{O}(n^2)$

Prova:

- se $n \geq 10$ então $n^2 + 10n \leq n^2 + n \cdot n = n^2 + n^2 = 2 \cdot n^2$
- resumo: $n^2 + 10n \leq 2\,n^2$ para todo $n \geq 10$

Como adivinhei que 2 e 10 são bons valores para c e N respectivamente?
Resposta: fiz o seguinte rascunho:

- quero $n^2 + 10n \leq c\,n^2$
- dividindo por n^2, quero $1 + 10/n \leq c$
- se $n \geq 10$ então $1 + 10/n \leq 2$
- parece que basta tomar $c \geq 2$ e $N \geq 10$

Exemplo 2

$9n + 9 = O(n^2)$

Prova:

- se $n \geq 9$ então $9n + 9 \leq n \cdot n + n^2 = 2 \cdot n^2$
- resumo: $9n + 9 \leq 2\,n^2$ para todo $n \geq 9$

Outra prova:

- se $n \geq 1$ então $9n + 9 \leq 9n \cdot n + 9 \cdot n^2 = 18\,n^2$

Exemplo 3

$n^2 \stackrel{?}{=} O(9n + 9)$

Não é verdade! Prova, por contradição:

- suponha que existem c e N tais que
  $n^2 \leq c \cdot (9n + 9)$ para todo $n \geq N$
- então $n^2 \leq c \cdot (9n + 9n) = 18\,c\,n$ para todo $n \geq N$
- então $n \leq 18\,c$ para todo $n \geq N$
- absurdo

Exemplo 4

$3n \;=\; O(2^n)$

Prova:

- vou mostrar que $3n \leq 2^n$ para $n \geq 4$
- prova por indução matemática:
- se $n = 4$ então $3n = 3 \cdot 4 = 12 \leq 16 = 2^4 = 2^n$
- se $n > 4$ então

$$\begin{aligned} 3n &= 3(n-1+1) \\ &= 3(n-1)+3 \\ &\leq 3(n-1)+3(n-1) \\ &\leq 2^{n-1}+2^{n-1} \quad \text{(hipótese de indução)} \\ &= 2^n \end{aligned}$$

Também poderia mostrar que $3n \leq 2 \cdot 2^n$ para $n \geq 1$

Exemplo 5

$\log_2 n \;=\; O(n)$

Prova:

- $n \leq 2^n$ quando $n \geq 1$ (como no exemplo anterior)
- $\log_2$ é crescente
- logo, $\log_2 n \leq n$ quando $n \geq 1$

$a \leq b$ se e somente se $2^a \leq 2^b$

logo, $m \leq n$ implica $\log_2 m \leq \log_2 n$

Bases de logaritmos

- $\log_2 n = \dfrac{\log_3 n}{\log_3 2} = \dfrac{1}{\log_3 2} \log_3 n$

- todos os log, qualquer que seja a base, estão na mesma order O

Reprise: algoritmo da ordenação por inserção

Desempenho de ORDENA-POR-INSERÇÃO:

- algoritmo consome $O(n^2)$ unidades de tempo

- não é verdade que consome $O(n)$ unidades de tempo

- existem instâncias que levam o algoritmo a consumir tempo proporcional a n^2

Aula 3

Problema da intercalação (merge)

Mergesort

Novo problema/algoritmo: intercalação

Problema: Rearranjar um vetor em ordem crescente sabendo que o lado e o lado direito são crescentes

Sei fazer em $O(n^2)$ unidades de tempo

Dá pra melhorar?

p				q						r
22	44	55	88	88	33	66	77	99	99	99

p				q						r
22	33	44	55	66	77	88	88	99	99	99

Possíveis valores de p, q e r?

Algoritmo: Recebe $A[p\,..\,r]$ tal que $A[p\,..\,q]$ e $A[q{+}1\,..\,r]$ são crescentes. Rearranja o vetor todo em ordem crescente.

INTERCALA (A, p, q, r)

$n_1 \leftarrow q - p + 1$
$n_2 \leftarrow r - q$
crie vetores $L[1\,..\,n_1+1]$ e $R[1\,..\,n_2+1]$
para $i \leftarrow 1$ até n_1 faça $L[i] \leftarrow A[p+i-1]$
para $j \leftarrow 1$ até n_2 faça $R[j] \leftarrow A[q+j]$
$L[n_1+1] \leftarrow R[n_2+1] \leftarrow \infty$
⋮

⋮
$i \leftarrow j \leftarrow 1$
para $k \leftarrow p$ até r faça
  se $L[i] \leq R[j]$
    então $A[k] \leftarrow L[i]$
        $i \leftarrow i+1$
    senão $A[k] \leftarrow R[j]$
        $j \leftarrow j+1$

Análise do desempenho:

- tamanho de instância: $n := r - p + 1$
- algoritmo consome $\mathrm{O}(n)$ unidades de tempo
- isso é muito rápido!

Novo problema/algoritmo: Mergesort

Problema: Rearranjar um vetor em ordem crescente

Algoritmo: Rearranja $A[p\,..\,r]$ em ordem crescente supondo $p \leq r$

MERGE-SORT (A, p, r)

se $p < r$
  então $q \leftarrow \lfloor (p+r)/2 \rfloor$
      MERGE-SORT (A, p, q)
      MERGE-SORT $(A, q+1, r)$
      INTERCALA (A, p, q, r)

Método de divisão e conquista

Segredo da velocidade do MERGE-SORT: INTERCALA é rápido

Desempenho: Quanto tempo MERGE-SORT consome?

- Tamanho de instância: $n := r - p + 1$
- $T(n)$:= tempo para pior instância de tamanho n
- Quero cota superior: $T(n) = \mathrm{O}(n \lg n)$
- Se $n > 1$ então $T(n) \leq T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + f(n)$

  sendo $f(n)$ é consumo de tempo de INTERCALA: $f(n) = \mathrm{O}(n)$
- Vamos provar que $T(n) = \mathrm{O}(n \lg n)$

Notação: $\lg n := \log_2 n$ e $\ln n := \log_e n$

Aula 4

Análise do Mergesort

Solução de recorrências

Novo problema/algoritmo: Mergesort

Problema: Rearranjar um vetor em ordem crescente

Algoritmo: Rearranja $A[p\,..\,r]$ em ordem crescente, supondo $p \le r$

```
MERGE-SORT (A, p, r)
1  se p < r
2     então q ← ⌊(p + r)/2⌋
3           MERGE-SORT (A, p, q)
4           MERGE-SORT (A, q + 1, r)
5           INTERCALA (A, p, q, r)
```

Desempenho: Quanto tempo MERGE-SORT consome?

- Tamanho de instância: $n := r - p + 1$
- $T(n)$:= tempo de pior instância de tamanho n
- Vou mostrar que $T(n) = \mathrm{O}(n \lg n)$


```
MERGE-SORT (A, p, r)
1  se p < r
2     então q ← ⌊(p + r)/2⌋
3           MERGE-SORT (A, p, q)
4           MERGE-SORT (A, q + 1, r)
5           INTERCALA (A, p, q, r)
```

$$T(n) \le \begin{cases} a & \text{se } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + f(n) & \text{se } n = 2, 3, 4, 5, \ldots \end{cases}$$

a é o consumo de tempo da linha 1

$f(n)$ é o consumo de INTERCALA mais o das linhas 1 e 2

Queremos uma "fórmula fechada" para $T(n)$

Exemplo: solução de recorrência

Recorrência semelhante à do Mergesort:

$$T(n) = \begin{cases} 3 & \text{se } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + 10\,n & \text{se } n = 2, 3, 4, , \ldots \end{cases}$$

Recorrência define função T sobre inteiros positivos:

n		$T(n)$
1	3	3
2	$3 + 3 + 10 \cdot 2$	26
3	$3 + 26 + 10 \cdot 3$	59
4	$26 + 26 + 10 \cdot 4$	92
5	$26 + 59 + 10 \cdot 5$	135

Quero fórmula fechada $T(n) = \cdots$

Não sei dar uma fórmula fechada...

Vou começar tentando algo mais simples: somente potências de 2

$$S(n) = \begin{cases} 3 & \text{se } n = 1 \\ 2\,S(\frac{n}{2}) + 10\,n & \text{se } n = 2^1, 2^2, 2^3, \ldots \end{cases}$$

Fórmula fechada: $S(n) = 10\,n \lg n + 3n$

Prova?

Lembrete: $\lg n := \log_2 n$

Teorema: $S(n) = 10\,n \lg n + 3n$ para $n = 2^0, 2^1, 2^2, 2^3, \ldots$

Prova, por indução matemática:

- Base: se $n = 1$ então $S(n) = 3 = 10n \lg n + 3n$
- Passo: se $n > 1$ então

$$\begin{aligned} S(n) &= 2\,S(\tfrac{n}{2}) + 10\,n \\ &= 2\left(10\,\tfrac{n}{2} \lg \tfrac{n}{2} + 3\tfrac{n}{2}\right) + 10\,n \quad \text{(hipótese de indução)} \\ &= 10\,n(\lg n - 1) + 3n + 10\,n \\ &= 10\,n \lg n - 10n + 3n + 10\,n \\ &= 10\,n \lg n + 3n \end{aligned}$$

Resposta mais grosseira: $S(n) = \mathrm{O}(n \lg n)$

Prova: ...

Como adivinhei a fórmula "$10\,n \lg n + 3n$"?

Desenrolei a recorrência:

$$\begin{aligned} S(n) &= 2\,S(n/2) + 10\,n \\ &= 4\,S(n/4) + 20\,n \\ &= 8\,S(n/8) + 30\,n \\ &\vdots \quad \vdots \\ &= n\,S(1) + 10 \lg n\, n \\ &= 3\,n + 10\,n \lg n \end{aligned}$$

De volta ao desempenho do Mergesort

$$T(n) \;\le\; T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) \;+\; \mathrm{O}(n)$$

Diferenças em relação ao exemplo anterior:

- "$\le$" no lugar de "$=$"
- $\lceil . \rceil$ e $\lfloor . \rfloor$
- "$1, 2, 3, 4, 5, \ldots$" no lugar de "$2^0, 2^1, 2^2, 2^3, \ldots$"
- "$\mathrm{O}(n)$" no lugar de "$f(n)$"

Apesar dessas diferenças, CLRS (sec.4.3 p.73–75) garante que

$$T(n) = \mathrm{O}(n \lg n)$$

Exercício

Resolva a recorrência

$$R(n) \;=\; \begin{cases} a & \text{se } n = 1 \\ 2\,R(\frac{n}{2}) + b\,n & \text{se } n = 2^1, 2^2, 2^3, \ldots \end{cases}$$

Solução: $R(n) = b\,n \lg n + a\,n$

Prova, por indução:

- Base: se $n = 1$ então $R(n) = a$ e $bn \lg n + an = a$
- Passo: se $n > 1$ então

$$\begin{aligned} R(n) &= 2\,R(\tfrac{n}{2}) + b\,n \\ &= 2\left(b\tfrac{n}{2} \lg \tfrac{n}{2} + a\tfrac{n}{2}\right) + b\,n \\ &= b\,n(\lg n - 1) + a\,n + b\,n \\ &= b\,n \lg n - b\,n + a\,n + b\,n \\ &= b\,n \lg n + a\,n \end{aligned}$$

Exercício

O algoritmo supõe $n \ge 1$ e devolve o valor de um elemento máximo de $A[1\,..\,n]$

```
MAX (A, n)
se n = 1
    então devolva A[1]
    senão x ← MAX (A, n − 1)
          se x ≥ A[n]
             então devolva x
             senão devolva A[n]
```

Análise do desempenho:

- tamanho de instância: n
- $T(n)$: tempo de pior caso para instância de tamanho n

$$T(n) \;\le\; \begin{cases} a & \text{se } n = 1 \\ T(n-1) + b & \text{se } n = 2, 3, 4, 5, \ldots \end{cases}$$

sendo a e b constantes

Teorema: $T(n) \le a + b(n-1)$ para todo inteiro positivo n

Prova: ...

Colorário: $T(n) = \mathrm{O}(n)$

Prova: ...

Aula 5

Ordens Ω e Θ

Algoritmo Heapsort

Ordem Ω

Definição: Dizemos que $f = \Omega(g)$ se existem constantes $c > 0$ e $N > 0$ tais que $f(n) \geq c \cdot g(n)$ para todo $n \geq N$

Exemplos:

- $n^2 = \Omega(n)$
- $n^2 = \Omega(n^2)$
- $n^2 - 10n - 100 = \Omega(n^2)$
- $n \lg n = \Omega(n)$
- $1.001^n = \Omega(n^{100})$
- n não é $\Omega(n^2)$

$f = \Omega(g)$ se e somente se $g = \mathrm{O}(f)$

Ordem Θ

Definição: Dizemos que $f = \Theta(g)$ se $f = \mathrm{O}(g)$ e $f = \Omega(g)$

Exemplos:

- $100\,n^2 = \Theta(n^2)$
- $n^2 - 10n - 100 = \Theta(n^2)$
- $\lg n = \Theta(\ln n)$
- $n \lg n$ não é $\Theta(n^2)$

Novo algoritmo: Heapsort

Rearranja $A[1\,..\,n]$ em ordem crescente

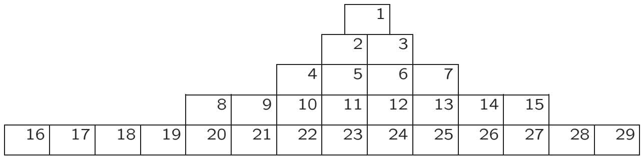
```
HEAPSORT (A, n)
para i ← ⌊n/2⌋ decrescendo até 1
    faça MAX-HEAPIFY (A, n, i)
para m ← n decrescendo até 2
    faça A[1] ↔ A[m]
          MAX-HEAPIFY (A, m − 1, 1)
```

1						m				n
66	44	55	44	22	33	11	77	88	99	99

Invariante: no início de cada iteração

- $A[m+1\,..\,n]$ grandes, ordem crescente
- $A[1\,..\,m]$ pequenos, max-heap

Vetor $A[1\,..\,m]$:



filho esquerdo do nó i	$2i$
filho direito do nó i	$2i+1$
pai do nó i	$\lfloor i/2 \rfloor$
raiz (nó 1)	nível 0
nível do nó i	$\lfloor \lg i \rfloor$
nível p	tem 2^p nós
altura do nó i	$\lfloor \lg \frac{m}{i} \rfloor$
folha	altura 0

$A[1\,..\,m]$ é um *max-heap* se $A[\lfloor i/2 \rfloor] \geq A[i]$ para cada i

MAX-HEAPIFY recebe $A[1\,..\,m]$ e $i \geq 1$ tais que subárvores com raiz $2i$ e $2i+1$ são max-heaps.
Rearranja de modo que subárvore com raiz i seja max-heap.

MAX-HEAPIFY (A, m, i)

```
e ← 2i
d ← 2i + 1
se e ≤ m e A[e] > A[i]
    então x ← e
    senão x ← i
se d ≤ m e A[d] > A[x]
    então x ← d
se x ≠ i
    então A[i] ↔ A[x]
          MAX-HEAPIFY (A, m, x)
```


Aula 6

Algoritmo Heapsort (continuação)

Filas de prioridade

MAX-HEAPIFY recebe $A[1\,..\,m]$ e $i \geq 1$ tais que subárvores com raiz $2i$ e $2i+1$ são max-heaps.
Rearranja de modo que subárvore com raiz i seja max-heap.

MAX-HEAPIFY (A, m, i)

```
e ← 2i
d ← 2i + 1
se e ≤ m e A[e] > A[i]
    então x ← e
    senão x ← i
se d ≤ m e A[d] > A[x]
    então x ← d
se x ≠ i
    então A[i] ↔ A[x]
          MAX-HEAPIFY (A, m, x)
```


Correto?

Desempenho de MAX-HEAPIFY, versão 1:

- tamanho da instância: altura do nó i
  $h := \lfloor \lg \frac{m}{i} \rfloor$
- consumo de tempo no pior caso: $T(h)$
- "recorrência": $T(h) = T(h-1) + \mathrm{O}(1)$
- mais concreto:
  $T(1) = a$ e $T(h) = T(h-1) + b$ para $h > 1$

  solução: $T(h) = bh - b + a$ para $h \geq 1$

  resumo: $T(h) = \mathrm{O}(h)$
- como $h \leq \lg m$, podemos dizer que consumo é $\mathrm{O}(\lg m)$

Desempenho de MAX-HEAPIFY, versão 2:

- tamanho da instância: número de nós na subárvore com raiz i
  $M \cong m/i$
- consumo de tempo no pior caso: $T(M)$
- "recorrência": $T(M) \leq T(2M/3) + \mathrm{O}(1)$

  justificativa do "2/3"...
- mais concreto (a e b números positivos):
  $T(1) = a$ e $T(M) \leq T(2M/3) + b$ para $M > 1$

  solução: $T(M) \leq b \log_{3/2} M + a$

  resumo: $T(M) = \mathrm{O}(\lg M)$
- como $M < \frac{2m}{i} \leq 2m$, o algoritmo é $\mathrm{O}(\lg m)$

Construção de um max-heap:

```
para i ← ⌊n/2⌋ decrescendo até 1
    faça MAX-HEAPIFY (A, n, i)
```

Invariante: no início de cada iteração
$i+1, \ldots, n$ são raízes de max-heaps

Desempenho:

- tamanho de instância: n
- consumo de tempo no pior caso: $T(n)$
- $T(n) = \frac{n}{2}\,\mathrm{O}(\lg n) = \mathrm{O}(n \lg n)$.
- análise mais cuidadosa: $T(n) = \mathrm{O}(n)$.

Prova de que a construção do heap consome $\mathrm{O}(n)$:

Para simplificar, suponha árvore é completa e tem altura h
portanto $n = 2^{h+1} - 1$, portanto $\frac{1}{2}(n+1) = 2^h$

altura	cada nó consome	quantos nós
1	1	2^{h-1}
2	2	2^{h-2}
3	3	2^{h-3}
$\vdots$	$\vdots$	
h	h	2^0

$$1 \cdot 2^{h-1} + 2 \cdot 2^{h-2} + \ldots + h \cdot 2^0$$

$$\left(1 \cdot 2^{-1} + 2 \cdot 2^{-2} + \ldots + h \cdot 2^{-h}\right) \cdot 2^h$$

$$\left(1 \cdot \tfrac{1}{2}^1 + 2 \cdot \tfrac{1}{2}^2 + \ldots + h \cdot \tfrac{1}{2}^h + \cdots\right) \cdot \tfrac{1}{2}(n+1)$$

$$\tfrac{1/2}{(1-1/2)^2} \cdot \tfrac{1}{2}(n+1)$$

$$n+1$$

Detalhes da prova:

$$x^0 + x^1 + x^2 + x^3 + \cdots = \frac{1}{1-x}$$

$$1x^0 + 2x^1 + 3x^2 + \cdots = \frac{1}{(1-x)^2}$$

$$1x^1 + 2x^2 + 3x^3 + \cdots = \frac{x}{(1-x)^2}$$

Consumo de tempo do Heapsort:

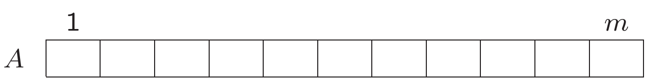
```
HEAPSORT (A, n)
para i ← ⌊n/2⌋ decrescendo até 1          O(n)
    faça MAX-HEAPIFY (A, n, i)             O(n)
para m ← n decrescendo até 2              O(n)
    faça A[1] ↔ A[m]                       O(n)
            MAX-HEAPIFY (A, m − 1, 1)      n O(lg n)
```

Total: $O(n \lg n)$

Consumo de tempo no pior caso: $\Omega(n \lg n)$.

Fila de prioridades (priority queue)

Tipo abstrato de dados (= *abstract data type*)



Organizar $A[1\,..\,m]$ de modo que as operações

- consulte máximo
- extraia máximo
- aumenta valor de elemento
- insira novo elemento

sejam eficientes

Vetor sem ordem alguma? crescente? decrescente?

Implementações com max-heap

```
HEAP-MAXIMUM (A, m)
devolva A[1]
```

Consome $O(1)$ unidades de tempo

```
HEAP-EXTRACT-MAX (A, m)  ▷ m ≥ 1
max ← A[1]
A[1] ← A[m]
m ← m − 1
MAX-HEAPIFY (A, m, 1)
devolva max
```

Consomem $O(\lg m)$ unidades de tempo

HEAP-INCREASE-KEY $(A, i, chave)$ $\triangleright$ $chave \geq A[i]$

$A[i] \leftarrow chave$
enquanto $i > 1$ e $A[\lfloor i/2 \rfloor] < A[i]$
    faça $A[i] \leftrightarrow A[\lfloor i/2 \rfloor]$
        $i \leftarrow \lfloor i/2 \rfloor$

No início de cada iteração,
$A[1\,..\,m]$ é um max-heap exceto talvez pela violação $A[\lfloor i/2 \rfloor] < A[i]$

MAX-HEAP-INSERT $(A, m, chave)$

$m \leftarrow m + 1$
$A[m] \leftarrow -\infty$
HEAP-INCREASE-KEY $(A, m, chave)$

Todos consomem $\mathrm{O}(\lg m)$

Aula 7

Quicksort

CLRS cap.7

Novo algoritmo: Quicksort

O algoritmo QUICKSORT rearranja $A[p\,..\,r]$ em ordem crescente

QUICKSORT (A, p, r)

1 se $p < r$
2   então $q \leftarrow$ PARTICIONE (A, p, r)
3   QUICKSORT $(A, p, q-1)$
4   QUICKSORT $(A, q+1, r)$

p					q					r
22	22	22	11	11	44	88	99	88	99	88

No começo da linha 3, $A[p\,..\,q{-}1] \;\leq\; A[q] \;\leq\; A[q{+}1\,..\,r]$

PARTICIONE rearranja o vetor de modo que $p \leq q \leq r$ e
$A[p\,..\,q{-}1] \;\leq\; A[q] \;<\; A[q{+}1\,..\,r]$

PARTICIONE (A, p, r)

$x \leftarrow A[r]$ $\triangleright$ x é o "pivô"
$i \leftarrow p - 1$
para $j \leftarrow p$ até $r - 1$
    faça se $A[j] \leq x$
        então $i \leftarrow i + 1$
            $A[i] \leftrightarrow A[j]$
$A[i+1] \leftrightarrow A[r]$
devolva $i + 1$

Invariantes: no começo de cada iteração
$A[p\,..\,i] \leq x$ $A[i{+}1\,..\,j{-}1] > x$ $A[r] = x$

p		i				j				r
22	22	22	88	88	88	11	99	11	99	44

Consumo de tempo de PARTICIONE

- tamanho de instância: $n := r - p + 1$
- consumo de tempo: $\Theta(n)$ unidades de tempo

Consumo de tempo do QUICKSORT

- tamanho de instância: $n := r - p + 1$
- o algoritmo é $\mathrm{O}(n^2)$
- o algoritmo é $\Omega(n \lg n)$

como veremos adiante

Exemplos exploratórios

- se PARTICIONE divide sempre $n/2$–para–$n/2$

  então $T(n) = 2\,T(n/2) + \Theta(n)$

  donde $T(n) = \Theta(n \lg n)$

- se PARTICIONE divide $n/10$–para–$9n/10$

  então $T(n) = T(n/10) + T(9n/10) + \Theta(n)$

  done $T(n) = \Theta(n \lg n)$

- se PARTICIONE divide 0–para–$n-1$

  então $T(n) = T(n-1) + \Theta(n)$

  done $T(n) = \Theta(n^2)$

Desempenho do QUICKSORT no pior caso

- $P(n)$: consumo de tempo no pior caso
- $P(n) \;\le\; \max_{0 \le k < n} \big(P(k) + P(n{-}k{-}1)\big) + \Theta(n)$
- cota superior: $P(n) = \mathrm{O}(n^2)$
- prova?
- em lugar de uma prova, vou examinar um exemplo concreto

Exemplo concreto:

$S(0) = S(1) = 1$ e

$S(n) = \max_{0 \le k < n} \big(S(k) + S(n{-}k{-}1)\big) + n$ para $n \ge 2$

n	$S(n)$
0	1
1	1
2	$1+1+2=4$
3	$1+4+3=8$
4	$1+8+4=13$

Vou provar que $S(n) \le n^2 + 1$ para $n \ge 0$

Prova:

- se $n \le 1$ então ... trivial
- se $n \ge 2$ então ...

- se $n \geq 2$ então

$$\begin{aligned} S(n) &= \max_{0 \leq k < n} \Big(S(k) + S(n-k-1)\Big) + n \\ &\overset{\text{hi}}{\leq} \max \Big(k^2 + 1 + (n-k-1)^2 + 1\Big) + n \\ &= (n-1)^2 + 2 + n \qquad \triangleright \text{ CLRS 7.4-3} \\ &= n^2 - n + 3 \\ &\leq n^2 + 1 \end{aligned}$$

Exercício (CLRS 7.4-1): $S(n) \geq \frac{1}{2}n^2$ para todo $n \geq 1$

Desempenho do Quicksort no melhor caso

- $M(n)$: consumo de tempo no melhor caso
- $M(n) \ \geq \min_{0 \leq k < n} \Big(M(k) + M(n-k-1)\Big) + \Theta(n)$
- cota inferior: $M(n) = \Omega(n \lg n)$
- prova? CLRS 7.4-2

Quicksort aleatorizado

Particione-Ale (A, p, r)
$i \leftarrow$ Random (p, r)
$A[i] \leftrightarrow A[r]$
devolva Particione (A, p, r)

Quicksort-Ale (A, p, r)
se $p < r$
então $q \leftarrow$ Particione-Ale (A, p, r)
Quicksort-Ale $(A, p, q-1)$
Quicksort-Ale $(A, q+1, r)$

Consumo de tempo esperado (ou seja, médio): $\Theta(n \lg n)$

Aula 8

i-ésimo menor elemento

Mediana

CLRS cap.9

Novo problema: i-ésimo menor elemento

Problema: Encontrar o i-ésimo menor elemento de $A[p\,..\,r]$

Suponha $A[p\,..\,r]$ sem elementos repetidos

Uma instância: 33 é o 4-o menor elemento

22	99	32	88	34	**33**	11	97	55	66	A
11	22	32	**33**	34	55	66	88	97	99	ordenado

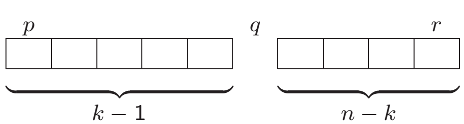
Mediana: $\lfloor \frac{n+1}{2} \rfloor$-ésimo menor

Observações:

- tamanho de instância: $n := r - p + 1$
- problema só tem solução se i está no intervalo $1\,..\,n$
- caso $i = 1$: algoritmo $\mathrm{O}(n)$
- caso $i = 2$: algoritmo $\mathrm{O}(n)$
- mediana: algoritmo $\mathrm{O}(n \lg n)$
- i arbitrário: algoritmo $\mathrm{O}(n \lg n)$
- existe algo melhor?

Algoritmo: Recebe vetor $A[p\,..\,r]$ (elementos todos diferentes)
e i no intervalo $1\,..\,r{-}p{+}1$
e devolve i-ésimo menor elemento

```
SELECT (A, p, r, i)
se p = r
     então devolva A[p]
q ← PARTICIONE (A, p, r)
k ← q − p + 1
se k = i
     então devolva A[q]
se k > i
     então devolva SELECT (A, p, q − 1, i)
     senão devolva SELECT (A, q + 1, r, i − k)
```



Consumo de tempo

- proporcional ao número de comprações entre elementos de A
- todas as comparações acontecem no PARTICIONE
- PARTICIONE(A, p, r) faz $n - 1$ comparações
- $T(n)$: número comparações entre elementos de A no pior caso
- $T(n) = T(n-1) + n{-}1$

  solução: $T(n) = \Theta(n^2)$
- caso médio?

EXEMPLO

Número médio de comparações no pior caso

Examina todas as pertmutações $1, 2, \ldots, n$

Supõe linha 6 não se aplica e linhas 8–9 escolhem o lado maior

$A[p\,.\,.\,r]$	comps
1,2	1+0
2,1	1+0
média	2/2

$A[p\,.\,.\,r]$	comps
1,2,3	2+1
2,1,3	2+1
1,3,2	2+0
3,1,2	2+0
2,3,1	2+1
3,2,1	2+1
média	16/6

$A[p\,.\,.\,r]$	comps
1,2,3,4	3+3
2,1,3,4	3+3
1,3,2,4	3+2
3,1,2,4	3+2
2,3,1,4	3+3
3,2,1,4	3+3
1,2,4,3	3+1
2,1,4,3	3+1
1,4,2,3	3+1
4,1,2,3	3+1
2,4,1,3	3+1
4,2,1,3	3+1

$A[p\,.\,.\,r]$	comps
1,3,4,2	3+1
3,1,4,2	3+1
1,4,3,2	3+1
4,1,3,2	3+1
3,4,1,2	3+1
4,3,1,2	3+1
2,3,4,1	3+3
3,2,4,1	3+3
2,4,3,1	3+2
4,2,3,1	3+2
3,4,2,1	3+3
4,3,2,1	3+3
média	116/24

Mais um caso: quando $r - p + 1 = 5$, a média é 864/120

Esperança (número médio) de comparações

- $\mathrm{E}[T(n)] = ?$
- notação: $E(n) := \mathrm{E}[T(n)]$
- probab de que $A[p\,.\,.\,q]$ tenha exatamente k elementos: $\frac{1}{n}$
- recorrência:

$$E(n) \;\leq\; n{-}1 \;+\; \sum_{k=1}^{n} \frac{1}{n} \cdot E\Big(\max(k-1, n-k)\Big)$$

$$\leq\; n{-}1 \;+\; \frac{2}{n}\Big(E(\lfloor\tfrac{n}{2}\rfloor) + E(\lfloor\tfrac{n}{2}\rfloor{+}1) + \cdots + E(n{-}1)\Big)$$

- solução: $E(n) \stackrel{?}{=} \mathrm{O}(n)$

Implementação: SELECT-ALEATORIZADO

- troque PARTICIONE por PARTICIONE-ALEATORIZADO
- número esperado de comparações:

  $E(1) = 0$

  $E(n) \;\leq\; n{-}1 \;+\; \frac{2}{n}\Big(E(\lfloor\tfrac{n}{2}\rfloor) + \cdots + E(n{-}1)\Big)$ para $n \geq 2$

- alguns valores:

n	$E(n)$	$\cong$
1	0	0
2	2/2	1
3	16/6	2.7
4	116/24	4.8
5	864/120	7.2

compare com EXEMPLO acima

Teorema: $E(n) \leq 4n$ para todo inteiro $n \geq 1$

Prova:

- se $n = 1$ então $E(n) = E(1) = 0 \leq 4 = 4 \cdot 1 = 4n$
- se $n > 1$ então $E(n) \leq n - 1 + \frac{2}{n}\Big(E(\lfloor\frac{n}{2}\rfloor) + \cdots + E(n-1)\Big)$

$$\leq n - 1 + \frac{2}{n}\Big(4\lfloor\frac{n}{2}\rfloor + \cdots + 4(n-1)\Big)$$
$$< 4n - 1$$
$$< 4n$$

Confira, pois as contas não "batem" com as do livro!

Corolário: $E(n) = \mathrm{O}(n)$

Aula 9

Árvores binárias de busca

Programação dinâmica: números de Fibonacci e multiplicação de cadeia de matrizes

CLRS cap.12, cap.15

Árvores binárias de busca

Árvores binárias:

- vetor $\times$ árvore binária
- vetor crescente $\times$ árvore binária de busca
- busca binária em vetor $\times$ "busca binária" em lista encadeada

Estrutura:

- nó x
- $chave[x]$
- pai: $p[x]$
- filhos: $esq[x]$ e $dir[x]$
- valor especial NIL

Árvore de busca: para todo nó x
para todo y na subárvore esquerda de x
e todo nó z na subárvore direita de x

$$chave[y] \leq chave[x] \leq chave[z]$$

Varredura esquerda-raiz-direita:
imprime os nós da subárvore que tem raiz x

```
VARREDURA-E-R-D (x)  ▷ inorder traversal
se x ≠ NIL
    então VARREDURA-E-R-D (esq[x])
          imprima chave[x]
          VARREDURA-E-R-D (dir[x])
```

- árvore é de busca $\Leftrightarrow$ varredura erd dá chaves em ordem crescente
- tamanho da instância: número de nós na subárvore de raiz x notação: n
- consumo de tempo: $T(n) = T(k) + T(n-k-1) + \Theta(1)$ para algum $0 \leq k \leq n-1$
- $T(n) = \Theta(n)$

Tipo abstrato de dados

- operações: busca, mínimo, máximo, sucessor, predecessor, inserção, remoção
- todas consomem $O(h)$, sendo h a altura da árvore


```
BUSCA-EM-ÁRVORE (x, k)  ▷ Tree-Search
se x = NIL ou k = chave[x]
    então devolva x
se k < chave[x]
    então BUSCA-EM-ÁRVORE (esq[x], k)
    senão BUSCA-EM-ÁRVORE (dir[x], k)
```

```
MIN-DE-ÁRVORE (x)  ▷ Tree-Minimum
enquanto esq[x] ≠ NIL
    faça x ← esq[x]
devolva x
```

SUCESSOR-EM-ÁRVORE (x) ▷ Tree-Successor

se $dir[x] \neq$ NIL
    então devolva MIN-DE-ÁRVORE $(dir[x])$
$y \leftarrow p[x]$
enquanto $y \neq$ NIL e $x = dir[y]$
    faça $x \leftarrow y$
        $y \leftarrow p[y]$
devolva y

INSERÇÃO-EM-ÁRVORE?
REMOÇÃO-EM-ÁRVORE?

Todas as operações consomem $O(h)$
sendo h a altura da árvore

Árvore balanceada: $h = O(\lg n)$ sendo n o número de nós

Programação dinâmica

- "recursão–com–tabela"

- transformação inteligente de recursão em iteração

Exemplos:

- números de Fibonacci
- árvore ótima de busca (CLRS sec.15.2)
- multiplicação de cadeias de matrizes
- subseqüência comum máxima
- mochila booleana

Aqui, a palavra "programação" não tem nada a ver com
programação de computadores

Problema 1: números de Fibonacci

$$F_0 = 0 \qquad F_1 = 1 \qquad F_n = F_{n-1} + F_{n-2}$$

Algoritmo recursivo:

FIBO-REC (n)

se $n \leq 1$
    então devolva n
    senão $a \leftarrow$ FIBO-REC $(n-1)$
        $b \leftarrow$ FIBO-REC $(n-2)$
        devolva $a + b$

Consumo de tempo é proporcional ao número de somas:

$$T(0) = T(1) = 0$$
$$T(n) = T(n-1) + T(n-2) + 1 \quad \text{se } n \geq 2$$

Solução: $T(n) \geq 3^n/2^n$ para $n \geq 6$

Consumo exponencial: $T(n) = \Omega\big((\frac{3}{2})^n\big)$

Por que tão ineficiente?
Algoritmo resolve a mesma subinstância muitas vezes:

```
                F5
             /      \
          F4          F3
         /  \        /  \
       F3    F2    F2    F1
      /  \  /  \  /  \
    F2  F1 F1  F0 F1  F0
```

Fibonacci via programação dinâmica

```
FIBO (n)
f[0] ← 0
f[1] ← 1
para i ← 2 até n
    faça f[i] ← f[i−1] + f[i−2]
devolva f[n]
```

Tabela $f[0\,..\,n{-}1]$

Consumo de tempo: $\Theta(n)$

Problema 2: multiplicação de cadeias de matrizes

se A é $p \times q$ e B é $q \times r$ então AB é $p \times r$

$(AB)[i,j] = \sum_k A[i,k]\, B[k,j]$

número de multiplicações escalares $= p \cdot q \cdot r$

Multiplicação em cadeia: $A_1 \cdot A_2 \cdot A_3$

$$\begin{array}{ccccccc} 10 & A_1 & 100 & A_2 & 5 & A_3 & 50 \end{array}$$

$((A_1\, A_2)\, A_3)$	7500	mults escalares
$(A_1\, (A_2\, A_3))$	75000	mults escalares

Problema: Encontrar número mínimo de multiplicações escalares necessário para calcular produto $A_1\, A_2 \cdots A_n$

$$\begin{array}{ccccccccc} p_0 & & p_1 & & p_2 & \cdots & p_{n-1} & & p_n \\ & A_1 & & A_2 & & \cdots & & A_n & \end{array}$$

cada A_i é $p_{i-1} \times p_i$

Continua...

Aula 10

Programação dinâmica:

Multiplicação de cadeia de matrizes

Subset sum (= soma de cheques)

CLRS cap.15

Programação dinâmica

Problema 2: multiplicação de cadeia de matrizes

se A é $p \times q$ e B é $q \times r$ então AB é $p \times r$

$(AB)[i,j] = \sum_k A[i,k]\, B[k,j]$

número de multiplicações escalares $= p \cdot q \cdot r$

Multiplicação em cadeia: $A_1 \cdot A_2 \cdot A_3$

$$10 \quad A_1 \quad 100 \quad A_2 \quad 5 \quad A_3 \quad 50$$

$((A_1\, A_2)\, A_3)$	7500	mults escalares
$(A_1\, (A_2\, A_3))$	75000	mults escalares

Problema: Encontrar número mínimo de multiplicações escalares necessário para calcular produto $A_1\, A_2 \cdots A_n$

$$p_0 \quad A_1 \quad p_1 \quad A_2 \quad p_2 \quad \cdots \quad p_{n-1} \quad A_n \quad p_n$$

cada A_i é $p_{i-1} \times p_i$

Passo 1: Estrutura recursiva do problema

Soluções ótimas contêm soluções ótimas: se

$$(A_1 A_2)\,(A_3((A_4 A_5) A_6))$$

minimiza multiplicações então

$$(A_1 A_2) \quad \text{e} \quad (A_3((A_4 A_5) A_6))$$

também minimizam

Passo 2: Algoritmo recursivo:

recebe $p_{i-1}, \ldots, p_j$ com $i \le j$ e devolve número mínimo de multiplicações escalares

```
REC-MAT-CHAIN (p, i, j)
se i = j
    então devolva 0
x ← ∞
para k ← i até j − 1 faça
    a ← REC-MAT-CHAIN (p, i, k)
    b ← REC-MAT-CHAIN (p, k + 1, j)
    q ← a + p_{i−1} p_k p_j + b
    se q < x
        então x ← q
devolva x
```

- tamanho de uma instância: $n := j - i + 1$
- consumo de tempo: $\Omega(2^n)$
- demora tanto porque mesma instância resolvida muitas vezes

Passo 3: Programação dinâmica

$m[i,j] =$	número mínimo de multiplicações escalares para calcular $A_i \cdots A_j$

decomposição: $(A_i \cdots A_k)\,(A_{k+1} \cdots A_j)$

recorrência:

se $i = j$ então $m[i,j] = 0$

se $i < j$ então $m[i,j] = \min\limits_{i \le k < j} \Big(m[i,k] + p_{i-1} p_k p_j + m[k{+}1,j] \Big)$

Exemplo: $m[3,7] = \min\limits_{3 \le k < 7} \{\, m[3,k] + p_2 p_k p_7 + m[k{+}1,7] \,\}$

- cada instância $A_i \cdots A_j$ resolvida uma só vez
- em que ordem calcular os componentes da tabela m?
- para calcular $m[2,6]$ preciso de
  $m[2,2]$, $m[2,3]$, $m[2,4]$, $m[2,5]$ e de
  $m[3,6]$, $m[4,6]$, $m[5,6]$, $m[6,6]$

	1	2	3	4	5	6	7	8	j
1	0								
2		0	$\star$	$\star$	$\star$	?			
3			0			$\star$			
4				0		$\star$			
5					0	$\star$			
6						0			
7							0		
8								0	
i									

Calcule todos os $m[i,j]$ com $j-i+1=2$,
depois todos com $j-i+1=3$,
depois todos com $j-i+1=4$,
etc.

Programação dinâmica: recebe $p_0, p_1, \ldots, p_n$ e devolve $m[1,n]$

```
MATRIX-CHAIN-ORDER(p, n)
1  para i ← 1 até n faça
2      m[i,i] ← 0
3  para l ← 2 até n faça
4      para i ← 1 até n − l + 1 faça
5          j ← i + l − 1
6          m[i,j] ← ∞
7          para k ← i até j − 1 faça
8              q ← m[i,k] + p_{i−1} p_k p_j + m[k+1,j]
9              se q < m[i,j]
0                  então m[i,j] ← q
1  devolva m[1,n]
```

- compare com REC-MAT-CHAIN
- linhas 4–10 tratam das subcadeias $A_i \cdots A_j$ de comprimento l
- um dos invariantes menos importantes: $j-i+1=l$

Prova da correção do algoritmo:

- estrutura recursiva (optimal substructure property)
- (veja slide 93)

Consumo de tempo:

- tamanho de uma instância: $n := j-i+1$
- obviamente $\Theta(n^3)$
  (três loops encaixados)

Programação dinâmica
Problema 3: Subset sum (soma de cheques)

Problema: Dados inteiros não-negativos $w_1, \ldots, w_n$, W encontrar $K \subseteq \{1, \ldots, n\}$ tal que $\sum_{k \in K} w_k = W$

- Uma instância: $w = 100, 30, 90, 35, 40, 30, 10$ e $W = 160$
- motivação: problema dos cheques
  algum subconj dos cheques $w_1, \ldots, w_n$ tem soma W?
- solução força-bruta:
  examine todos os 2^n subconjuntos de $\{1, \ldots, n\}$

Algoritmo recursivo (ineficiente)

devolve 1 se instância tem solução e 0 em caso contrário

```
REC (w, n, W)
se W = 0
    então devolva 1
se n = 0
    então devolva 0
se REC (w, n − 1, W) = 1
    então devolva 1
se w_n > W
    então devolva 0
    senão devolva REC (w, n−1, W−w_n)
```


Prova da correção do algoritmo:
depende da subestrutura ótima (optimal substructure property):
se K é solução da instância (n, W) então

- se $n \in K$ então
  $K - \{n\}$ é solução da instância $(n-1, W - w_n)$
- se $n \notin K$ então
  K é solução da instância $(n-1, W)$

Consumo de tempo:

- $\Omega(2^n)$ (Prova?)
- demora tanto porque mesma instância resolvida muitas vezes

Algoritmo de programação dinâmica

$$s[i, Y] := \begin{cases} 1 & \text{se instância } (i, Y) \text{ tem solução} \\ 0 & \text{caso contrário} \end{cases}$$

Recorrência:
$$s[i, 0] = 1$$
$$s[0, Y] = 0 \quad \text{se } Y > 0$$
$$s[i, Y] = s[i-1, Y] \quad \text{se } w_i > Y$$
$$s[i, Y] = \max\big(s[i-1, Y], s[i-1, Y - w_i]\big)$$

	0	1	2	3	4
0	0	0	0	0	0
1					
2					
3	⋆	⋆	⋆	⋆	
4				?	
5					

colunas: Y vai de 0 a W
linhas: i vai de 0 a n

lembrete: W e $w_1, \ldots, w_n$ são inteiros!

Subset-Sum (w, n, W)

```
aloca s[0..n, 0..W]
para i ← 0 até n faça
    s[i, 0] ← 1
para Y ← 1 até W faça
    s[0, Y] ← 0
    para i ← 1 até n faça
        s[i, Y] ← s[i−1, Y]
        se s[i, Y] = 0 e w_i ≤ Y
            então s[i, Y] ← s[i−1, Y − w_i]
devolva s[n, W]
```

cada instância (i, Y) resolvida uma só vez

Prova da correção:

- propriedade da subestrutra ótima

Consumo de tempo:

- obviamente $\Theta(n\,W)$

Exercício:

- escreva uma versão que devolva K tal que $\sum_{k \in K} w_k = W$

Resumo do projeto do algoritmo

- passo 1: encontrar a propridade da subestrutura ótima (optimal substructure property)
- passo 2: escrever algoritmo recursivo
- passo 3: transformar algoritmo em programação dinâmica

Comentários sobre o consumo de tempo do algoritmo

Algoritmo consome tempo $\Theta(nW)$

A dependência de W é má notícia:

> se multiplicar $w_1, \ldots, w_n, W$ por 100
> a instância continua essencialmente a mesma
> mas o algoritmo consome 100 vezes mais tempo!

Qual o tamanho de uma instância $w_1, \ldots, w_n, W$ do problema?

- é muito razoável dizer que o tamanho é $n + 1$
- infelizmente, não dá pra escrever "$\Theta(nW)$" como função de $n + 1$ porque nW envolve o **valor** (e não só o tamanho) do W
- a definição de tamanho deveria levar em conta o **valor** de W e não apenas a *presença* de W
- outro detalhe: é necessário definir o tamanho por *dois* números
- adote o par (n, W) como tamanho de uma instância
- então $\Theta(n\,W)$ é função do tamanho

- mas (n, W) não é a definição ideal de tamanho:
  se multiplicarmos $w_1, \ldots, w_n, W$ por 100
  a instância continua essencialmente a mesma
  mas o seu tamanho passa de (n, W) a $(n, 100\,W)$,
  o que não é razoável!
- definição razoável de **tamanho de um número** (como W, por exemplo):
  número de caracteres
- exemplo: o tamanho do número 256877 é 6 (e não 256877)
- conclusão: tamanho de instância $w_1, \ldots, w_n, W$ é $(n, \lceil \log(W{+}1) \rceil)$
- o consumo de tempo de SUBSET-SUM é $\Theta(n\, 2^{\log W})$
- o algoritmo é considerado exponencial
- veja fim da página
  `www.ime.usp.br/~pf/analise_de_algoritmos/aulas/mochila.html`

Aula 11

Programação dinâmica: subseqüência comum máxima

CLRS sec.15.4

Problema: Subseqüência comum máxima

- Definição:
  $\langle z_1, \ldots, z_k \rangle$ é **subseqüência** de $\langle x_1, \ldots, x_m \rangle$ se
  existem índices $i_1 < \cdots < i_k$ tais que

$$z_1 = x_{i_1}\,,\ z_2 = x_{i_2}\,,\ \ldots\,,\ z_k = x_{i_k}$$

- Uma instância:
  $\langle 5, 9, 2, 7 \rangle$ é subseqüência de $\langle 9, 5, 6, 9, 6, 2, 7, 3 \rangle$

Pequenas observações:

- $\langle x_1, \ldots, x_m \rangle$ é o mesmo que $x[1\,..\,m]$
- x_j é o mesmo que $x[j]$
- não é bom escrever "$Z \subseteq X$"

Exercício preliminar:

- Decidir se $\langle z_1, \ldots, z_k \rangle$ é subseqüência de $\langle x_1, \ldots, x_m \rangle$
- Uma instância:
  `A A A` é subseq de `B A B B A B B B A A B B A B A B A B B`?
- Solução gulosa:

```
SUB-SEQ (z, k, x, m)
i ← k
j ← m
enquanto i ≥ 1 e j ≥ 1 faça
    se z_i = x_j
        então i ← i − 1
    j ← j − 1
se i ≥ 1
    então devolva "não"
    senão devolva "sim"
```

Prove que o algoritmo está correto!

Problema: Encontrar uma subseqüência comum máxima de seqüências X e Y

- Z é subseq **comum** de X e Y se Z é subseq de X e de Y
- minha abreviatura: ssco = subseqüência comum
- abreviatura do livro: LCS = longest common subsequence
- uma instância (fig 15.6 de CLRS):

X	A B C B D A B
Y	B D C A B A
ssco	B C A
ssco maximal	A B A
ssco máxima	B C B A
outra ssco máxima	B D A B

- invente heurística gulosa para o problema
- mostre que ela não resolve o problema

Algoritmo de programação dinâmica

Problema simplificado:
encontrar o **comprimento** de uma sscomáx de X e Y

$$\boxed{c[i,j] = \text{comprimento de uma sscomáx de } X_i \text{ e } Y_j}$$

Recorrência:

- $c[0,j] = c[i,0] = 0$
- $c[i,j] = c[i-1,j-1] + 1$ se $i,j > 0$ e $x_i = y_j$
- $c[i,j] = \max\big(c[i,j-1], c[i-1,j]\big)$ se $i,j > 0$ e $x_i \neq y_j$

```
LCS-LENGTH (X, m, Y, n)
aloca c[0..m, 0..n]
para i ← 0 até m faça
    c[i, 0] ← 0
para j ← 1 até n faça
    c[0, j] ← 0
para i ← 1 até m faça
    para j ← 1 até n faça
        se x_i = y_j
            então c[i, j] ← c[i − 1, j − 1] + 1
            senão se c[i − 1, j] ≥ c[i, j − 1]
                      então c[i, j] ← c[i − 1, j]
                      senão c[i, j] ← c[i, j − 1]
devolva c[m, n]
```


horizontal: $X =$ A B C B D A B
vertical: $Y =$ B D C A B A

		0	1	2	3	4	5	6
			B	D	C	A	B	A
0		0	0	0	0	0	0	0
1	A	0						
2	B	0						
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

horizontal: $X =$ A B C B D A B
vertical: $Y =$ B D C A B A

		0	1	2	3	4	5	6
			B	D	C	A	B	A
0		0	0	0	0	0	0	0
1	A	0	0					
2	B	0						
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

horizontal: $X =$ A B C B D A B
vertical: $Y =$ B D C A B A

		0	1	2	3	4	5	6
			B	D	C	A	B	A
0		0	0	0	0	0	0	0
1	A	0	0	0				
2	B	0						
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

horizontal: $X =$ A B C B D A B
vertical: $Y =$ B D C A B A

		0	1	2	3	4	5	6
			B	D	C	A	B	A
0		0	0	0	0	0	0	0
1	A	0	0	0	0			
2	B	0						
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

horizontal: $X =$ A B C B D A B
vertical: $Y =$ B D C A B A

		0	1	2	3	4	5	6
			B	D	C	A	B	A
0		0	0	0	0	0	0	0
1	A	0	0	0	0	1		
2	B	0						
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

horizontal: $X =$ A B C B D A B
vertical: $Y =$ B D C A B A

		0	1	2	3	4	5	6
			B	D	C	A	B	A
0		0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0						
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

horizontal: $X =$ A B C B D A B
vertical: $Y =$ B D C A B A

		0	1	2	3	4	5	6
			B	D	C	A	B	A
0		0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

horizontal: $X =$ A B C B D A B
vertical: $Y =$ B D C A B A

		0	1	2	3	4	5	6
			B	D	C	A	B	A
0		0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	3	3
5	D	0	1	2	2	2	3	3
6	A	0	1	2	2	3	3	4
7	B	0	1	2	2	3		

horizontal: $X =$ A B C B D A B
vertical: $Y =$ B D C A B A

		0	1	2	3	4	5	6
			B	D	C	A	B	A
0		0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	3	3
5	D	0	1	2	2	2	3	3
6	A	0	1	2	2	3	3	4
7	B	0	1	2	2	3	4	

horizontal: $X =$ A B C B D A B
vertical: $Y =$ B D C A B A

		0	1	2	3	4	5	6
			B	D	C	A	B	A
0		0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	3	3
5	D	0	1	2	2	2	3	3
6	A	0	1	2	2	3	3	4
7	B	0	1	2	2	3	4	4

Prova da correção do algoritmo:

- $Z = \langle z_1, \ldots, z_k \rangle$, $X = \langle x_1, \ldots, x_m \rangle$, $Y = \langle y_1, \ldots, y_n \rangle$
- notação: $X_i := \langle x_1, \ldots, x_i \rangle$
- propriedade da subestrutura ótima:

  suponha Z é sscomáx de X e Y

  - se $x_m = y_n$
    então $z_k = x_m = y_n$ e Z_{k-1} é sscomáx de X_{m-1} e Y_{n-1}
  - se $x_m \neq y_n$ e $z_k \neq x_m$
    então Z é sscomáx de X_{m-1} e Y
  - se $x_m \neq y_n$ e $z_k \neq y_n$
    então Z é sscomáx de X e Y_{n-1}

Prove a propriedade!

Exemplo da propriedade da subestrutura ótima:

X	A B C B D A B
Y	B D C A B A
ssco máxima	B C B A
outra ssco máxima	B D A B

Consumo de tempo:

- tamanho de instância é o par (m, n)
- linha 7 ("se $x_i = y_j$") é executada mn vezes
- consumo de tempo: $O(mn)$

Aula 12

Algoritmos gulosos

O problema do disquete

CLRS cap 16

Algoritmos gulosos

Algoritmo guloso

- em cada iteração, acrescenta à solução o objeto que parece melhor naquele momento
- procura maximal e acaba obtendo máximo
- procura ótimo local e acaba obtendo ótimo global

Costuma ser

- muito simples e intuitivo
- muito eficiente
- *difícil provar que está correto*

Problema precisa ter

- subestrutura ótima (como na programação dinâmica)
- propriedade da escolha gulosa (*greedy-choice property*)

O problema do disquete

Instâncias "v=1" da mochila booleana

Problema: Dados inteiros não-negativos $w_1, \ldots, w_n$ e W encontrar $K \subseteq \{1, \ldots, n\}$ máximo tal que $\sum_{k \in K} w_k \leq W$

- Diferente do subset sum dos slides 101–106
- Motivação: gravação de arquivos em um disquete

Problema simplificado: encontrar $|K|$ máximo

Algoritmo guloso, versão recursiva:

dado $I \subseteq \{1, \ldots, n\}$ o algoritmo devolve $|K|$ sendo K subconjunto máximo de I tal que $\sum_{k \in K} w_k \leq W$

DISQUETE-REC (w, W, I)
se $I = \emptyset$
    então devolva 0
escolha m em I tal que $w_m = \min\{w_i : i \in I\}$
se $w_m > W$
    então devolva 0
    senão devolva $1 +$ DISQUETE-REC $(w, W - w_m, I - \{m\})$

Prova da correção do algoritmo:

- escolha gulosa *(greedy-choice property)*:

  se $w_m = \min\{w_1, \ldots, w_n\}$ e $w_m \leq W$
  então m pertence a alguma solução ótima

- subestrutura ótima *(optimal substructure property)*:

  se K é solução ótima da instância (n, W) e $n \in K$
  então $K - \{n\}$ é solução ótima para $(n - 1, W - w_n)$

A prova das propriedades é muito fácil!

Versão melhor de DISQUETE-REC:

supõe $w_1 \geq \cdots \geq w_n$

DISQUETE-REC (w, n, W)
se $n = 0$ ou $w_n > W$
    então devolva 0
    senão devolva $1 +$ DISQUETE-REC $(w, n - 1, W - w_n)$

Consumo de tempo:

- tamanho de uma instância: n
- recorrência: $T(n) = T(n-1) + \mathrm{O}(1)$
- solução: $T(n) = \mathrm{O}(n)$

Versão iterativa, supondo $w_1 \geq \cdots \geq w_n$:

```
DISQUETE-IT (w, n, W)
k ← 0
para j ← n decrescendo até 1 faça
    se w_j > W
        então devolva k
        senão k ← k + 1
              W ← W − w_j
devolva k
```

Consumo de tempo: $O(n)$

Programação dinâmica consumiria $\Theta(n^2)$

Aula 13

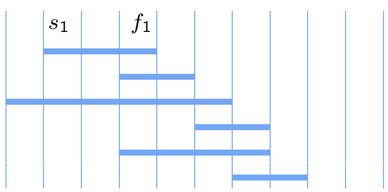
Algoritmos gulosos: problema dos intervalos disjuntos

CLRS seções 16.1, 16.2 e 16.3

Problema: coleção disjunta máxima de intervalos

Problema: Dados intervalos $[s_1, f_1), \ldots, [s_n, f_n)$ encontrar uma coleção máxima de intervalos disjuntos dois a dois

Instância:



Solução (subconjunto de $\{1, \ldots, n\}$):

Intervalos i e j são disjuntos se $f_i \leq s_j$ ou $f_j \leq s_i$

Outra instância:

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

- um candidato a solução: $\{3, 9, 11\}$
- uma solução ótima: $\{1, 4, 8, 11\}$

A estrutura "gulosa" do problema

Propriedade da escolha gulosa:

- se f_m é mínimo então m está em alguma solução ótima

Propriedade da subestrutura ótima:

- se A é solução ótima e $m \in A$ é tal que f_m é mínimo
  então $A - \{m\}$ é solução ótima de $\{j : s_j \geq f_m\}$
  (todos os intervalos "posteriores" a f_m)

Propriedade mais geral da subestrutura ótima:

- se A é solução ótima e $A \ni m$
  então $A - \{m\}$ é solução ótima de $\{i : f_i \leq s_m\} \cup \{j : s_j \geq f_m\}$
  (intervalos "anteriores" a s_m ou "posteriores" a f_m)

Prove as propriedades!

Algoritmo guloso, versão recursiva:

dado $I \subseteq \{1, \dots, n\}$
o algoritmo devolve subconjunto máximo A de I tal que
os intervalos $[s_a, f_a)$, $a \in A$, são disjuntos dois a dois

```
INTERV-DISJ-REC (s, f, I)   ▷ supõe I ≠ ∅
se |I| = 1
    então devolva I
    senão escolha m em I tal que f_m = min {f_i : i ∈ I}
            J ← {j ∈ I : s_j ≥ f_m}
            A ← INTERV-DISJ-REC (s, f, J)
            devolva {m} ∪ A
```

prova da correção: escolha gulosa + subestrutura ótima

Versão recursiva mais eficiente:

- não precisa administrar o conjunto I explicitamente
- supõe $f_1 \leq \cdots \leq f_n$

Algoritmo devolve uma subcol disjunta máx de $\{\, [s_k, f_k) : s_k \geq f_h \,\}$

```
INTERV-DISJ-REC(s, f, h)
m ← h + 1
enquanto m ≤ n e s_m < f_h
    faça m ← m + 1
se m > n
    então devolva ∅
    senão devolva {m} ∪ INTERV-DISJ-REC (s, f, m)
```

Para resolver problema original
adote $f_0 := -\infty$ e diga INTERV-DISJ-REC $(s, f, 0)$

Algoritmo guloso, versão iterativa:

supõe $f_1 \leq \cdots \leq f_n$

```
INTERVALOS-DISJUNTOS (s, f, n)   ▷ supõe n ≥ 1
A ← {1}
i ← 1
para m ← 2 até n faça
    se s_m ≥ f_i
        então A ← A ∪ {m}
                i ← m
devolva A
```

Nome do algoritmo no CLRS: Greedy-Activity-Selector
A organização do algoritmo é semelhante à do PARTICIONE

Consumo de tempo:

- tamanho de instância: n
- consumo de tempo: $\Theta(n)$

Minha primeira versão de INTERVALOS-DISJUNTOS era correta e eficiente, mas deselegante:

```
INTERVALOS-DISJUNTOS-FEIO (s, f, n)
A ← ∅
i ← 1
enquanto i ≤ n faça
    A ← A ∪ {i}
    m ← i + 1
    enquanto m ≤ n e s_m < f_i faça
        m ← m + 1
    i ← m
devolva A
```

Mostre que consumo de tempo é $O(n)$
(apesar dos dois loops)

Aula 14

Algoritmos gulosos: códigos de Huffman

CLRS seção 16.3

Códigos de Huffman

Trata de compressão de dados e combina várias idéias/técnicas/estruturas:

- gula
- fila de prioridades
- árvores binárias

- texto = seqüência de caracteres
- cada caracter representado por uma seq de bits
- código de um caracter = seq de bits
- número de bits por caracter pode ser variável
- exemplo: `a` = `0`, `b` = `101`, etc.
- código livre de prefixos
- fácil codificação/decodificação: `01010101101` = `ababb`

Dado um arquivo (= seqüência de caracteres) $\langle c_1, \ldots, c_n \rangle$, com $c_i \in C$ para todo i

Cada c em C ocorre $f[c]$ vezes no arquivo

Problema:
Encontrar uma codificação de caracteres em seqs de bits que minimize o comprimento do arquivo codificado

Exemplo:

caracter c	a	b	c	d	e	f
freqüência $f[c]$	45	13	12	16	9	5
código de comprimento fixo ..	000	001	010	011	100	101
código de comprimento variável	0	101	100	111	1101	1100

se texto original tem 100 caracteres, texto codificado terá

- 300 bits se código de comprimento fixo
- 224 bits se código de comprimento variável

- tabela de código é representada por árvore binária
- árvore binária cheia: cada nó tem 0 ou 2 filhos
- caracteres são as folhas
- "0" se desce para esquerda, "1" se desce para direita
- $d(c)$:= profundidade de folha c na árvore
- $d(c)$ = número de bits de caracter c
- custo da árvore := $\sum_c f[c]\, d(c)$
- texto codificado terá $\sum_c f[c]\, d(c)$ bits

Problema reformulado: encontrar árvore de custo mínimo

exemplos:

```
            .

       0/      \1
       /        \
      /          \
     .            .
    / \          /
   /   \        /
  .     .      .
 / \   / \    / \
a   b c   d  e   f
```

```
     .
    / \
   /   \
  a     .
       / \
      /   \
     .     .
    / \   / \
   c   b .   d
        / \
       f   e
```

Algoritmo guloso:

Recebe conjunto C de caracteres e vetor de freqüências f
constroi árvore do código e devolve raiz da árvore

```
HUFFMAN (C, f)
0  Q ← C
1  para i ← 1 até |C| − 1 faça
2      x ← EXTRACT-MIN (Q)
3      y ← EXTRACT-MIN (Q)
4      aloque novo nó z
5      esq[z] ← x
6      dir[z] ← y
7      chave[z] ← chave[x] + chave[y]
8      INSERT (Q, z)
9  devolva EXTRACT-MIN (Q)
```

Q é fila de prioridades

EXTRACT-MIN retira de Q um nó q com $chave[q]$ mínima

Detalhamento da linha 0 de HUFFMAN:

```
INICIALIZA-ÁRVORE (C)
para cada c em C faça
    aloque novo nó z
    esq[z] ← dir[z] ← NIL
    chave[z] ← f[c]
    INSERT (Q, z)
```


Desempenho de HUFFMAN:

- instância: (C, f)
- tamanho de instância: $n := |C|$
- inicialização da árvore: n vezes INSERT
- $n - 1$ vezes: EXTRACT-MIN, EXTRACT-MIN, INSERT
- cada EXTRACT-MIN e cada INSERT consome $O(\lg n)$

total: $O(n \lg n)$

Prova da correção de HUFFMAN:

Propriedade da escolha gulosa: se x e y minimizam f
então existe árvore ótima em que x e y são folhas-irmãs

Prova:

- sejam a e b são folhas-irmãs de profundidade máxima
- suponha $f[x] \le f[y]$ e $f[a] \le f[b]$
- troque x com a:

$$
\begin{aligned}
\text{custo}(T) - \text{custo}(T') &= \sum_c f[c]d(c) - \sum_c f[c]d'(c) \\
&= f[x]d(x) + f[a]d(a) \;-\; f[x]d'(x) - f[a]d'(a) \\
&= f[x]d(x) + f[a]d(a) \;-\; f[x]d(a) - f[a]d(x) \\
&= (f[a] - f[x])\,(d(a) - d(x)) \\
&\ge 0
\end{aligned}
$$

- agora troque y com b

Propriedade da subestrutura ótima:

suponha que x e y minimizam f
e T é uma árvore ótima que tem x e y como folhas-irmãs;
seja z o pai de x, y em T
e defina $C' := C - \{x, y\} \cup \{z\}$ e $f'[z] := f[x] + f[y]$;
então $T' := T - \{x, y\} + z$ é árvore ótima para (C', f')

Prova:

- custo(T) = custo(T') $+ f[x] + f[y]$
  ...

Aula 15

Union/Find: conjuntos disjuntos dinâmicos

CLRS seções 21.1, 21.2 e 21.3

ADT de conjuntos disjuntos dinâmicos

ADT = abstract data type = estrutura de dados abstrata
= conjunto de coisas mais repertório de operações sobre as coisas

Nossas coisas: conjuntos mutuamente disjuntos

Exemplo:

	$\{a\}$ $\{b\}$ $\{c\}$ $\{d\}$ $\{e\}$
une b, d	
	$\{a\}$ $\{b, d\}$ $\{c\}$ $\{e\}$
une c, e	
	$\{a\}$ $\{b, d\}$ $\{e, c\}$
une c, d	
	$\{a\}$ $\{b, d, c, e\}$
une e, d	
	$\{a\}$ $\{b, d, c, e\}$

Cada objeto (como b, por exemplo) pertence a um único conjunto

Operações de nossa ADT:

FINDSET (x)	qual dos conjuntos contém x?
UNION (x, y)	unir o conjunto que contém x ao que contém y
MAKESET (x)	construir conjunto $\{x\}$

Como dar "nomes" aos conjuntos?

- por exemplo, que nome dar ao conjunto $\{b, d\}$?
- resposta: cada conjunto tem um "representante"

Seqüência de operações MAKESET, UNION, FINDSET:

$$\underbrace{\underbrace{\text{M}\quad \text{M}\quad \text{M}}_{n}\quad \text{U}\quad \text{F}\quad \text{U}\quad \text{U}\quad \text{F}\quad \text{U}\quad \text{F}\quad \text{F}\quad \text{F}\quad \text{U}\quad \text{F}}_{m}$$

Diversas estruturas de dados poderiam ser usadas:

- vetor indexado pelos objetos
- uma lista ligada por conjunto
- etc.
- vamos usar *disjoint-set forest*

Estrutura de dados *disjoint-set forest*:

- cada nó x tem um pai $pai[x]$
- isso define um conjunto de árvores
- cada conjunto de objetos é uma das árvores
- cada árvore tem uma raiz r definida por $pai[r] = r$
- a raiz de uma árvore é o representante do conjunto

Nossa meta: tempo total O(m)
= tempo amortizado O(1) por operação

As árvores são diferentes das que vimos até agora:

- não têm ponteiros para filhos
- um nó pode ter qualquer número de filhos

Implementações básicas

FINDSET devolve a raiz da árvore que contém x

FINDSET-ITER (x)
enquanto $pai[x] \neq x$
    faça $x \leftarrow pai[x]$
devolva x

versão recursiva:

FINDSET-REC (x)
se $pai[x] = x$
    então devolva x
    senão devolva FINDSET-REC ($pai[x]$)

UNION (x, y)
$x' \leftarrow$ FINDSET (x)
$y' \leftarrow$ FINDSET (y)
$pai[x'] \leftarrow y'$

MAKESET (x)
$pai[x] \leftarrow x$

Exemplo:

para $i \leftarrow 1$ até 16
    faça MAKESET (x_i)
para $i \leftarrow 1$ até 15 em passos de 2
    faça UNION (x_i, x_{i+1})
para $i \leftarrow 1$ até 13 em passos de 4
    faça UNION (x_i, x_{i+2})
UNION (x_1, x_5)
UNION (x_{11}, x_{13})
UNION (x_1, x_{10})
FINDSET (x_2)
FINDSET (x_9)

Exemplo:

para $i \leftarrow 1$ até 9
    faça MAKESET (x_i)
UNION (x_1, x_2)
UNION (x_3, x_4)
UNION (x_4, x_5)
UNION (x_2, x_4)
UNION (x_6, x_7)
UNION (x_8, x_9)
UNION (x_8, x_6)
UNION (x_5, x_7)

Consumo de tempo:

MAKESET	O(1)
UNION	O(n)
FINDSET	O(n)

$$\underbrace{\underbrace{\mathrm{M}\ \ \mathrm{M}\ \ \mathrm{M}}_{n}\ \ \mathrm{U}\ \ \mathrm{F}\ \ \mathrm{U}\ \ \mathrm{U}\ \ \mathrm{F}\ \ \mathrm{U}\ \ \mathrm{F}\ \ \mathrm{F}\ \ \mathrm{F}\ \ \mathrm{U}\ \ \mathrm{F}}_{m}$$

Tempo total: $n\,\mathrm{O}(1) + m\,\mathrm{O}(n) + m\,\mathrm{O}(n) = \mathrm{O}(mn)$

Ainda está longe da meta de O(m) ...

Melhoramento 1: union by rank

MAKESETUR (x)
$pai[x] \leftarrow x$
$rank[x] \leftarrow 0$

UNIONUR (x, y) ▷ supõe x e y em árvores diferentes
$x' \leftarrow$ FINDSET (x)
$y' \leftarrow$ FINDSET (y)
se $rank[x'] = rank[y']$
    então $pai[y'] \leftarrow x'$
        $rank[x'] \leftarrow rank[x'] + 1$
    senão se $rank[x'] > rank[y']$
        então $pai[y'] \leftarrow x'$
        senão $pai[x'] \leftarrow y'$

- Exercício: $rank[x]$ é a altura do nó x

  (altura = mais longo caminho de x até uma folha)

- Fato importante: $rank[x] \le \lg n_x$,
  sendo n_x o número de nós na árvore que contém x

  (n_x não é o número de nós da subárvore com raiz x)

- Conclusão: todo caminho de um nó até uma raiz tem comprimento $\le \lg n$

Exemplo:

para $i \leftarrow 1$ até 9
    faça MAKESETUR (x_i)
UNIONUR (x_2, x_1)
UNIONUR (x_4, x_3)
UNIONUR (x_5, x_4)
UNIONUR (x_4, x_2)
UNIONUR (x_7, x_6)
UNIONUR (x_9, x_8)
UNIONUR (x_6, x_8)
UNIONUR (x_7, x_5)

Consumo de tempo da versão com *union by rank*:

MAKESETUR	O(1)
UNIONUR	O($\lg n$)
FINDSET	O($\lg n$)

$$\underbrace{\underbrace{\text{M M M}}_{n}\ \text{U F U U F U F F F U F}}_{m}$$

Tempo total: O($m \lg n$)

Ainda está longe de O(m) . . .

Melhoramento 2: union by rank + path compression

▷ A função devolve a raiz da árvore que contém x
▷ depois de fazer com que x e seus ancestrais se tornem filhos da raiz

FINDSETPC (x)
se $x \ne pai[x]$
    então $pai[x] \leftarrow$ FINDSETPC $(pai[x])$
devolva $pai[x]$

Código de UNIONURPC:
troque FINDSET por FINDSETPC em UNIONUR

Outra maneira de escrever o código:

```
FINDSETPC (x)
se x = pai[x]
    então devolva x
    senão pai[x] ← FINDSETPC (pai[x])
          devolva pai[x]
```


Exemplo:

```
0 para i ← 1 até 9
1     faça MAKESETUR (x_i)
2 UNIONURPC (x_2, x_1)
3 UNIONURPC (x_4, x_3)
4 UNIONURPC (x_5, x_4)
5 UNIONURPC (x_4, x_2)
6 UNIONURPC (x_7, x_6)
7 UNIONURPC (x_9, x_8)
8 UNIONURPC (x_6, x_8)
9 UNIONURPC (x_7, x_5)
```

repita com UNIONURPC (x_5, x_7) no lugar da linha 9

Consumo de tempo com *union by rank* + *path compression*:

MAKESETUR, UNIONURPC, FINDSETPC

$$\underbrace{\underbrace{\mathrm{M}\ \ \mathrm{M}\ \ \mathrm{M}}_{n}\ \ \mathrm{U}\ \ \mathrm{F}\ \ \mathrm{U}\ \ \mathrm{U}\ \ \mathrm{F}\ \ \mathrm{U}\ \ \mathrm{F}\ \ \mathrm{F}\ \ \mathrm{F}\ \ \mathrm{U}\ \ \mathrm{F}}_{m}$$

- tempo total: $\mathrm{O}(m \lg^* n)$
- prova é difícil
- custo amortizado de uma operação: $\mathrm{O}(\lg^* n)$
- tudo se passa como se todo caminho tivesse comprm $\leq \lg^* n$

$\lg^* n$ é o único número natural k tal que $T(k-1) < n \leq T(k)$
sendo $T(0) = 1$ e $T(k) = 2^{T(k-1)}$ para todo k

Exemplo: $T(4) = 2^{2^{2^2}} = 2^{(2^{(2^2)})}$

k	$T(k)$
0	$2^0 = 1$
1	$2^1 = 2$
2	$2^2 = 4$
3	$2^4 = 16$
4	$2^{16} = 65536$
5	2^{65536}

observe que $\lg T(k) = T(k-1)$ para $k = 1, 2, 3, \ldots$

portanto $\lg^* n$ é o menor k tal que $\underbrace{\lg \lg \ldots \lg}_{k} n \leq 1$

n	$\lg^* n$
1	0
2	1
3	1
4	2
5	2
$\vdots$	$\vdots$
15	2
16	3
$\vdots$	$\vdots$
65535	3
65536	4
$\vdots$	$\vdots$
$\underbrace{1000000000000000000\cdots000000000000}_{80}$	4
$\vdots$	$\vdots$

- $\lg^* n$ cresce tão devagar que é quase $O(1)$
- quase atingimos nossa meta de $O(m)$
- CLRS usa função $\alpha(n)$ que cresce ainda mais devagar que $\lg^* n$

Aula 16a

Grafos e árvores

CLRS cap. 23

Grafos

Um **grafo** é um par (V, E) de conjuntos tal que $E \subseteq V^{(2)}$

- $V^{(2)}$ é o conj dos pares não-ordenados de elementos de V
- $V^{(2)}$ é o conj dos pares $\{u, v\}$ com $u \in V$, $v \in V$, $u \neq v$
- $0 \leq |E| \leq \frac{1}{2}|V|\,(|V| - 1) < |V|^2$
- os elementos de V são chamados **vértices**
  os elementos de E são chamados **arestas**
- notação simplificada para aresta: (u, v) ou uv
- vértices u e v são **vizinhos** ou **adjacentes** se uv é aresta

Exemplo de grafo:

- $V = \{a, b, c, d, e, f, g, h, i\}$
- $E = \{\{a, b\}, \{b, c\}, \{c, d\}, \{d, e\}, \ldots, \{i, c\}\}$
- notação simplificada:

  $E = \{ab, bc, cd, de, ef, fg, gh, ha, bh, cf, ci, df, ig, ih, ic\}$

Conexão e componentes:

- um grafo é **conexo** se, para cada par x, y em V, existe caminho de x a y
- fato: se (V, E) é conexo então $|E| \geq |V| - 1$

  a recíproca é verdadeira?
- um **componente** de um grafo é um "pedaço" conexo maximal do grafo
- G é conexo sse G tem apenas 1 componente

Grafos: estrutura recursiva

Dentro de todo grafo há outros grafos

Subgrafo de $G = (V, E)$:

- $X \subseteq V$ e $F \subseteq X^{(2)}$
- se $F \subseteq E$ então (X, F) é **subgrafo** de G
- exemplo: um componente
- exemplo: mapa das ruas de Campinas é subgrafo do mapa completo (ruas e estradas) do estado SP

Contração de $G = (V, E)$:

- partição $\mathcal{X}$ de V e subconjunto $\mathcal{F}$ de $\mathcal{X}^{(2)}$
- suponha $(X, E \cap X^{(2)})$ conexo para cada X em $\mathcal{X}$
- suponha que $\{X, Y\}$ em $\mathcal{F}$ sse existe $\{x, y\}$ em E tal que $x \in X$ e $y \in Y$
- então $(\mathcal{X}, \mathcal{F})$ é **contração** de G
- exemplo: a coleção dos componentes de G (cada componente é um vértice da contração)
- exemplo: mapa das estradas do estado de SP é contração do mapa completo (ruas e estradas) do estado
- contração é um tipo especial de um máinor ($minor$, em inglês)

Exemplo importante de contração: o grafo G/uv

- $uv \in E$
- $\mathcal{X} \; := \; \{\{u,v\}\} \; \cup \; \{\{x\} : x \in V - \{u,v\}\}$
- $\mathcal{F}$ induzido por E da maneira óbvia
- $(\mathcal{X}, \mathcal{F})$ é uma contração de G
- notação: G/uv

veja CLRS seção B.4, p.1084

Cálculo de componentes

Problema: Encontrar o número de componentes de um grafo G

```
COMPONENTS-REC (G)
V ← V[G]
E ← E[G]
se E = ∅
    então devolva |V|
seja uv um elemento de E
devolva COMPONENTS-REC (G/uv)
```

Como implementar "G/uv" de maneira eficiente?

Resposta: estrutura "floresta de conjuntos disjuntos"
cada árvore da estrutura é um bloco da partição da V

Versão iterativa (convém escrever "(V, E)" no lugar de "G"):

```
COMPONENTS (V, E)
para cada v em V faça
    MAKESET (v)
c ← |V|
para cada (u, v) em E faça
    se FINDSET (u) ≠ FINDSET (v)
        então UNION (u, v)
              c ← c − 1
devolva c
```

Use MAKESETUR, UNIONURPC e FINDSETPC

- tamanho de instância: $(|V|, |E|)$
- temos V operações MAKESET seguidas de E operações UNION e FINDSET
- consumo de tempo: $\mathrm{O}((|V| + |E|) \lg^* |V|)$
- para simplificar, escreva $\mathrm{O}((V + E) \lg^* V)$
- se $V = \mathrm{O}(E)$ então consumo será $\mathrm{O}(E \lg^* V)$

Aula 16b

Árvores geradoras de peso mínimo e o algoritmo de Kruskal

CLRS cap. 23

Árvores

Uma árvore é um tipo especial de grafo:

- uma **árvore** é um grafo conexo (V, A) tal que $|A| = |V| - 1$
- árvores são "minimamente conexas"
- árvores não têm os conceitos "raiz", "pai", "filho"
- (V, A) é árvore <u>sse</u> (V, A) é conexo e não tem ciclos

Árvore geradora de um grafo $G = (V, E)$:

- uma **árvore geradora** de G é uma árvore (V, A) que é subgrafo de G
- definição alternativa: uma **árvore geradora** de G é um *subconjunto* A de E tal que (V, A) é árvore
- G tem uma árvore geradora <u>sse</u> G é conexo

Problema da árvore geradora mínima

Problema: Dado grafo (V, E) com pesos nas arestas encontrar uma árvore geradora de peso mínimo

- em inglês: *minimum spanning tree* ou *MST*
- cada aresta uv tem um peso $w(uv)$ (número real)
  é claro que $w(vu) = w(uv)$
- o peso de $A \subseteq E$ é $w(A) := \sum_{uv \in A} w(uv)$
- uma árvore geradora (V, A) é **ótima** se tem peso mínimo
- aplicação: pavimentação barata de rede de estradas

Exercício: Discuta o seguinte algoritmo guloso:

$A \leftarrow \emptyset$
para cada v em V faça
    $F \leftarrow \{e : e = xv \in E\}$
    seja f uma aresta de peso mínimo em F
    $A \leftarrow A \cup \{f\}$
devolva A

Algoritmo de Kruskal: avô de todos os algoritmos gulosos

MST-KRUSKAL-REC (G, w) $\triangleright$ supõe G conexo
1 se $E[G] = \emptyset$
2   então devolva $\emptyset$ $\triangleright$ G tem 1 só vértice
3 escolha uv em $E[G]$ que tenha w mínimo
4 $w' \leftarrow$ CONTRAI-PESOS (G, uv, w)
5 $A \leftarrow$ MST-KRUSKAL-REC $(G/uv, w')$
6 devolva $\{uv\} \cup A$

Na linha 4, CONTRAI-PESOS devolve w' definida assim:
para toda aresta XY de G/uv
$w'(XY) = \min\{w(xy) : xy \in E[G], x \in X, y \in Y\}$

Minha discussão do algoritmo de Kruskal é um pouco diferente da do CLRS

Prova de que o algoritmo está correto:

- propriedade da escolha gulosa:
  se aresta uv minimiza w
  então uv pertence a alguma árvore ótima
- propriedade da subestrutura ótima:
  se T é árvore ótima de G e uv é aresta de T
  então T/uv é árvore ótima de G/uv

Como implementar G/uv e w' de maneira eficiente?
Resposta: floresta de conjuntos disjuntos

MST-KRUSKAL (V, E, w) $\triangleright$ supõe (V, E) conexo
$A \leftarrow \emptyset$
para cada v em V
    faça MAKESET (v)
coloque E em ordem crescente de w
para cada $uv \in E$ em ordem crescente de w
    faça se FINDSET $(u) \neq$ FINDSET (v)
            então $A \leftarrow A \cup \{uv\}$
                UNION (u, v)
devolva A

- use MAKESETUR, UNIONURPC, FINDSETPC
- que acontece se grafo não for conexo?

Consumo de tempo do algoritmo MST-KRUSKAL:

- linha por linha:

linha	consumo
2–3	O(V)
4	O(E lg E)
5	O(E)
6	O(E lg* V)
7	O(E)
8	O(E lg* V)

- como $|V| - 1 \leq |E|$, consumo é O(E lg E)
- como $|E| < |V|^2$, consumo é O(E lg V)

Conclusão: poderia usar FINDSET no lugar de FINDSETPC

Aulas 17–18

Árvore geradora de peso mínimo: algoritmo de Prim

CLRS cap. 23

Problema da árvore geradora mínima

Problema: Dado grafo (V, E) com pesos nas arestas encontrar uma árvore geradora de peso mínimo

Versão simplificada: encontrar o peso de uma árvore geradora de peso mínimo

Para descrever algoritmo de Prim precisamos de dois conceitos:

- um **corte** é um par $(S, \overline{S})$ de subconjuntos de V sendo $\overline{S} := V - S$
- uma aresta uv **cruza** um corte $(S, \overline{S})$ se $u \in S$ e $v \in \overline{S}$ ou vice-versa

Exercício: Discuta o seguinte algoritmo guloso abaixo

O algoritmo supõe que $E =: \{e_1, \ldots, e_m\}$ e $w(e_1) \leq \cdots \leq w(e_m)$

```
A ← ∅
seja r um vértice qualquer
S ← {r}
para i ← 1 até m faça
    se e_i cruza o corte (S, S̄)
        então A ← A ∪ {e_i}
              seja u_i a ponta de e_i em S̄
              S ← S ∪ {u_i}
devolva A
```

Idéia do algoritmo de Prim:

- no começo de cada iteração, temos uma árvore (S, A) que é subgrafo de (V, E)
- cada iteração escolhe uma aresta de peso mínimo dentre as que cruzam o corte $(S, \overline{S})$

Prova de que o algoritmo está correto:

- propriedade da escolha gulosa:
  se aresta uv tem peso mínimo dentre as que cruzam um corte
  então uv pertence a alguma árvore ótima
- propriedade da subestrutura ótima:
  se T é árvore ótima de G e uv é aresta de T
  então T/uv é árvore ótima de G/uv

Rascunho 1 do algoritmo de Prim

```
RASCUNHO-1 (V, E, w)     ▷ supõe grafo conexo
1  escolha qualquer r em V
2  S ← {r}
3  peso ← 0
4  enquanto S ≠ V faça
5      min ← ∞
6      para cada e em E faça
7          se e cruza (S, S̄) e min > w(e)
8              então min ← w(e)
9                    seja u* a ponta de e em S̄
10     S ← S ∪ {u*}
11     peso ← peso + min
12 devolva peso
```


- bloco de linhas 5–9:
  escolhe uma aresta de peso mínimo
  dentre as que cruzam corte $(S, \overline{S})$
- prova da correção do algoritmo:
  no início de cada iteração,
  o conjunto de arestas (implicitamente) escolhidas
  é parte de alguma árvore geradora ótima

Consumo de tempo do RASCUNHO-1:

linha	consumo
1–3	O(1)
4–5	O(V)
6–10	O(VE)
11	O(V)
12	O(1)

- consumo total: O(VE)
- consumo muito alto...

Rascunho 2 do algoritmo de Prim

Novidades:

- para simplificar, vamos usar matriz simétrica W no lugar de w

$$W[u,v] \;:=\; \begin{cases} w(uv) & \text{se } uv \text{ é aresta} \\ \infty & \text{caso contrário} \end{cases}$$

- é como se o grafo fosse completo

- cada vértice v tem uma $\mathit{chave}[v]$ igual ao peso de uma aresta mínima dentre as que ligam v a S

RASCUNHO-2 (V, W, n) ▷ supõe grafo conexo

```
1  escolha qualquer r em V
2  S ← {r}
3  para cada u em V − {r} faça
4      chave[u] ← W[r, u]
5  peso ← 0
6  enquanto S ≠ V faça
7      min ← ∞
8      para cada u em V − S faça
9          se min > chave[u]
10             então min ← chave[u]
11                   u* ← u
12     S ← S ∪ {u*}
13     peso ← peso + chave[u*]
14     para cada v em V − S faça
15         se chave[v] > W[v, u*]
16             então chave[v] ← W[v, u*]
17 devolva peso
```


Observações:

- no começo de cada iteração
  $\mathit{chave}[u] = \min\{W[s,u] : s \in S\}$
  para cada u em $V - S$

- no começo da linha 12
  $\mathit{min} = \min\{W[s,u] : s \in S \text{ e } u \in V - S\}$

Consumo de tempo do RASCUNHO-2:

linha	consumo
1–5	$O(V)$
6–7	$O(V)$
8–11	$O(V^2)$
12–13	$O(V)$
14–16	$O(V^2)$

- consumo total: $O(V^2)$

- melhor que $O(VE)$, mas ainda alto...

- lembrete: $|V| - 1 \leq |E| < |V|^2$

Rascunho 3 do algoritmo de Prim

Novidades:

- listas de adjacência no lugar da matriz W
- $Adj[u]$ é a lista dos vértices vizinhos a u
- $V - S$ é armazenado numa fila de prioridades Q com prioridades dadas por $chave$
- Q pode ser implementada como um min-heap (mas há outras possibilidades)


```
RASCUNHO-3 (V, Adj, w)     ▷ supõe grafo conexo
 1  para cada u em V faça
 2       chave[u] ← ∞
 3  escolha qualquer r em V
 4  para cada u em Adj[r] faça
 5       chave[u] ← w(ru)
 6  peso ← 0
 7  Q ← FILA-PRIORIDADES (V − {r}, chave)
 8  enquanto Q ≠ ∅ faça
 9       u ← EXTRAI-MIN (Q)
10       peso ← peso + chave[u]
11       para cada v em Adj[u] faça
12            se v ∈ Q e chave[v] > w(vu)
13                 então DECREMENTE-CHAVE (Q, chave, v, w(vu))
14  devolva peso
```


Detalhes:

- FILA-PRIORIDADES $(U, chave)$ constrói fila de prioridades com conjunto de elementos U e prioridades dadas por $chave$ (por exemplo, um min-heap)
- DECREMENTE-CHAVE $(Q, chave, v, w(vu))$ faz $chave[v] \leftarrow w(vu)$ e faz os ajustes necessários na estrutura de Q

Consumo de tempo do RASCUNHO-3 supondo que fila de prioridades é um min-heap:

- consumo na linha 7: $\mathrm{O}(V)$
- consumo na linha 9: $\mathrm{O}(V \lg V)$
- consumo nas linhas 11–12: $\mathrm{O}(E)$ pois $\sum_v |Adj[v]| = 2|E|$ (método "agregado" de análise)
- consumo na linha 13: $\mathrm{O}(E \lg V)$
- consumo nas demais linhas: $\mathrm{O}(V)$
- total: $\mathrm{O}(V \lg V + E \lg V)$
- como $|E| \geq |V| - 1$, total é $\mathrm{O}(E \lg V)$

Rascunho 4 do algoritmo de Prim

Novidade:

- árvore inicial *não tem vértices*
- no rascunho anterior árvore inicial era $(\{r\}, \emptyset)$


```
RASCUNHO-4 (V, Adj, w)    ▷ supõe grafo conexo
para cada u em V faça
    chave[u] ← ∞
escolha qualquer r em V
chave[r] ← 0
peso ← 0
Q ← FILA-PRIORIDADES (V, chave)
enquanto Q ≠ ∅ faça
    u ← EXTRAI-MIN (Q)
    peso ← peso + chave[u]
    para cada v em Adj[u] faça
        se v ∈ Q e chave[v] > w(vu)
            então DECREMENTE-CHAVE (Q, chave, v, w(vu))
devolva peso
```

Consumo de tempo: $O(E \lg V)$

assintoticamente, o tempo necessário par ordenar as E arestas

Algoritmo de Prim: versão final

Novidades:

- devolve uma árvore geradora ótima
- poderia devolver conjunto A de arestas de árvore ótima, mas é melhor devolver um vetor de predecessores π


```
MST-PRIM (V, Adj, w)    ▷ supõe grafo conexo
para cada vértice u faça
    chave[u] ← ∞
    π[u] ← NIL
escolha qualquer r em V
chave[r] ← 0
Q ← FILA-PRIORIDADES (V, chave)
enquanto Q ≠ ∅ faça
    u ← EXTRAI-MIN (Q)
    para cada v em Adj[u] faça
        se v ∈ Q e chave[v] > w(uv)
            então DECREMENTE-CHAVE (Q, chave, v, w(vu))
                    π[v] ← u
devolva π
```

Consumo de tempo: $O(E \lg V)$

Qual das 5 versões do algoritmo é mais eficiente?

- precisamos comparar $O(VE)$ com $O(V^2)$ com $O(E \lg V)$
- qual a melhor?
- depende da relação entre E e V

Lições que desenvolvimento do algoritmo de Prim ensina:

- a idéia do algoritmo é simples
  mas a implementação eficiente não é trivial
- tamanho de instâncias é medido por *dois* números (E e V)
  que não são independentes ($V - 1 \leq E < V^2$)
- uma implementação pode ser melhor quando E próximo de V^2
  outra implementação pode ser melhor quando E longe de V^2
- boa implementação depende de
  boa estrutura de dados para grafos

Estruturas de dados para grafos

Representação de grafos:

- podemos supor $V = \{1, 2, 3, \ldots, n\}$
- matriz de adjacências: $M[u, v]$ vale 1 ou 0
- listas de adjacência: lista encadeada $Adj[u]$ para cada $u \in V$

Estrutura de dados para árvore geradora:

- adota uma raiz para a árvore
- cada nó u exceto a raiz terá um pai $\pi[u]$
- o conjunto de arestas da árvore será $\{\, (\pi[u], u) : v \in V - \{r\} \,\}$

Aula 19a

Ordenação em tempo linear

CLRS sec. 8.2 8.3

Ordenação por contagem

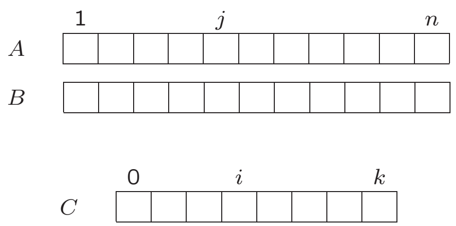
Problema: Encontrar uma permutação crescente de $A[1\,..\,n]$ sabendo que cada $A[j]$ está em $\{0,\ldots,k\}$

Exemplo:

2	2	1	0	1	2	2	1	2	0	0	0	1	2	2	1

Algoritmo rearranja $A[1\,..\,n]$ em ordem crescente supondo que $A[j] \in \{0,\ldots,k\}$ para cada j:

```
COUNTING-SORT (A, n, k)
1   para i ← 0 até k
2       faça C[i] ← 0
3   para j ← 1 até n
4       faça C[A[j]] ← C[A[j]] + 1
5   para i ← 1 até k
6       faça C[i] ← C[i] + C[i − 1]
7   para j ← n decrescendo até 1
8       faça B[C[A[j]]] ← A[j]
9            C[A[j]] ← C[A[j]] − 1
10  para j ← 1 até n
11      faça A[j] ← B[j]
```

Entre linhas 4 e 5, $C[i] := |\{m : A[m] = i\}|$

Entre linhas 6 e 7, $C[i] := |\{m : A[m] \le i\}|$

Invariante: antes de cada execução da linha 8, para cada i,

$$C[i] \;:=\; |\{m : A[m] < i\}| \;+\; |\{m : m \le j \text{ e } A[m] = i\}|$$

Consumo de tempo do COUNTING-SORT:

linha	consumo na linha
1–2	$\Theta(k)$
3–4	$\Theta(n)$
5–6	$\Theta(k)$
7–9	$\Theta(n)$
10–11	$\Theta(n)$

Consumo total: $\Theta(n+k)$

- se k é constante (não depende de n) então consumo é $\Theta(n)$
- se $k = \mathrm{O}(n)$ então consumo é $\Theta(n)$

A propósito: COUNTING-SORT é *estável*

Ordenação digital (radix sort)

Problema: Rearranjar $A[1\,..\,n]$ em ordem crescente sabendo que cada $A[j]$ tem d dígitos decimais

$$
\begin{aligned}
A[j] &= a_d \cdots a_2 a_1 \\
&= a_d\,10^{d-1} + \cdots + a_2\,10^1 + a_1\,10^0
\end{aligned}
$$

Exemplo:

329	457	657	839	436	720	355

Exemplo:

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

Algoritmo rearranja $A[1\,..\,n]$ em ordem crescente:

```
RADIX-SORT (A, n, d)
1  para i ← 1 até d faça      ▷ 1 até d e não o contrário!
2      ordene A[1..n] usando apenas dígito i como chave
```

É essencial que a ordenação na linha 2 seja estável

Consumo de tempo:

- se usar COUNTING-SORT, tempo será $\Theta(d\,n)$

Aula 19b

Limite inferior (lower bound) para o problema da ordenação

CLRS sec. 8.1

Ordenação: limite inferior

Problema: Rearranjar $A[1 \ldots n]$ em ordem crescente

- existem algoritmos $\mathrm{O}(n \lg n)$
- existe algoritmo assintoticamente melhor?
- depende
- todo algoritmo baseado em comparações é $\Omega(n \lg n)$
- prova?
- qualquer algoritmo de comparações é uma "árvore de decisão"

Exemplo:

```
INSERTIONSORT (A, n)
para j ← 2 até n faça
    chave ← A[j]
    i ← j − 1
    enquanto i ≥ 1 e A[i] > chave faça
        A[i + 1] ← A[i]
        i ← i − 1
    A[i + 1] ← chave
```

- árvore de decisão para $A = [a, b, c]$, elementos distintos
- "$a : b$" representa comparação entre a e b
- "bac" significa que $b < a < c$
- qquer execução do algoritmo é caminho da raiz até folha

Árvore de decisão:

$a : b$

$\leq$ $>$

$b : c$ $a : c$

/ \ / \

abc $a : c$ bac $b : c$

/ \ / \

acb cab bca cba

Árvore de decisão para um algoritmo que ordena $A[1 \mathinner{..} n]$:

- cada folha é uma permutação de $1, \ldots, n$
- todas as $n!$ permutações devem estar presentes
- quantas de comparações, no pior caso?
- resposta: altura, h, da árvore
- conclusão: devemos ter $2^h \geq n!$
- $(n!)^2 = \prod_{i=0}^{n-1} (i+1)(n-i) \geq \prod_{i=0}^{n-1} n = n^n$
- $n! \geq n^{n/2}$
- $h \geq \lg(n!) \geq \lg(n^{n/2}) \geq \frac{1}{2} n \lg n$

Conclusão:
Todo algoritmo de ordenação baseado em comparações faz

$$\Omega(n \lg n)$$

comparações no pior caso

Exercício: Qual o invariante na linha 8 do RADIX-SORT?

Exercício: Suponha que todos os elementos do vetor $A[1 \,..\, n]$ pertencem ao conjunto $\{1, 2, \ldots, n^3\}$. Escreva um algoritmo que rearranje $A[1 \,..\, n]$ o vetor em ordem crescente em tempo $\mathrm{O}(n)$.

Exercício: Faça árvore decisão para algoritmo SELSORT aplicado a $A[1 \,..\, 4]$.

Exercício: Mostre que qualquer algoritmo baseado em comparações necessita de pelo menos $n-1$ comparações para decidir se n números dados são todos iguais.

Exercício: Dadas seqüências estritamente crescentes $\langle a_1, \ldots, a_n \rangle$ e $\langle b_1, \ldots, b_n \rangle$, queremos ordenar a seqüência $\langle a_1, \ldots, a_n, b_1, \ldots, b_n \rangle$. Mostre que $2n-1$ comparações são necessárias, no pior caso, para resolver o problema.

Aula 20

Complexidade de problemas: introdução informal às classes P e NP

CLRS cap. 34

Introdução à complexidade de problemas

Até aqui, tratamos de problemas no varejo

Agora vamos trabalhar no atacado

Alguns exemplos de problemas computacionais

- Raiz quadrada:
  dado um inteiro positivo n, encontrar um inteiro positivo x
  tal que $x^2 = n$ ou constatar que tal x não existe

- Equação do segundo grau:
  dados inteiros a, b, c, encontrar um inteiro x tal que
  $ax^2 + bx + c = 0$ ou constatar que tal x não existe

- Equação diofantina:
  dada equação polinomial com número arbitrário de variáveis
  (como $x^3yz + 2y^4z^2 - 7xy^5z = 6$, por exemplo)
  encontrar valores das variáveis que satisfaçam a equação
  ou constatar que tais valores não existem

- Fatoração:
  dado um número natural $n > 1$, encontrar naturais $p > 1$ e $q > 1$
  tais que $n = p \times q$ ou constatar que tal par não existe

- MDC:
  encontrar o maior divisor comum de inteiros positivos p e q

- Fator comum grande:
  dados inteiros positivos p, q e k, encontrar
  divisor comum d de p e q tal que $d \geq k$
  ou constatar que tal divisor não existe

- Subseqüência crescente máxima:
  encontrar uma subseqüência crescente máxima
  de uma seqüência dada de inteiros positivos

- Subseqüência crescente longa:
  dado inteiro positivo k e seqüência de inteiros positivos,
  encontrar uma subseqüência crescente de comprimento $\geq k$
  ou constatar que uma tal subseq não existe

- Intervalos disjuntos:
  encontrar uma subcoleção disjunta máxima
  de uma coleção de intervalos dada

- Disquete:
  dados inteiros positivos $w_1, \ldots, w_n$ e W, encontrar
  subconj máximo J de $\{1, \ldots, n\}$ tal que $\sum_{j \in J} w_j \leq W$

- Subset sum:
  dados inteiros positivos $w_1, \ldots, w_n$ e W, encontrar
  $J \subseteq \{1, \ldots, n\}$ tal que $\sum_{j \in J} w_j = W$

- Árvore geradora de peso mínimo:
  dado um grafo com peso nas arestas,
  encontrar uma árvore geradora de peso mínimo
  ou constatar que o grafo não tem árvore geradora

- Árvore geradora de peso máximo: ...

- Caminho mínimo:
  dados vértices u e v de um grafo,
  encontrar um caminho de comprimento mínimo de u a v
  ou constatar que não existe caminho de u a v

- Caminho curto:
  dados vértices u e v de um grafo e um número k,
  encontrar um caminho de u a v de comprimento $\leq k$
  ou constatar que tal caminho não existe

- Caminho máximo: ...

- Ciclo mínimo:
  dado um grafo, encontrar ciclo com número mínimo de arestas
  ou constatar que o grafo não tem ciclo

- Ciclo longo:
  dado um grafo e um número k,
  encontrar ciclo simples de comprimento $\geq k$
  ou constatar que tal ciclo não existe

- Ciclo hamiltoniano:
  dado um grafo, encontrar um ciclo simples
  que passe por todos os vértices
  ou constatar que tal ciclo não existe

- Emparelhamento que cobre um lado de grafo bipartido:
  dado um grafo com bipartição (U, V),
  encontrar um emparelhamento que cubra U
  ou constatar que tal emparelhamento não existe

Observações sobre a lista de problemas:

- vários tipos de problemas:
  de busca, de maximização, de minimização, etc.

- vários dos problemas envolvem grafos

- em geral, nem todas as instâncias têm solução

- aparências enganam:
  problemas com enunciado semelhante
  podem ser muito diferentes

- alguns problemas são "casos particulares" de outros

Problemas polinomiais e a classe P

- algoritmo resolve problema se, para qualquer instância,
  dá resposta certa
  ou constata que a instância não tem solução

- algoritmo é polinomial se
  existe j tal que consumo de tempo $O(N^j)$
  sendo N o tamanho da instância

- problema é polinomial se
  existe algoritmo polinomial para o problema

- classe **P**: todos os problemas polinomiais

- dê um exemplo de problema fora de **P**

  compare com a prova do *lower bound* $\Omega(n \lg n)$ da ordenação

Complexidade de problemas

- questão: "quão fácil é o problema A?"
  resposta: existe algoritmo polinomial para A

- questão: "quão difícil é o problema A?"
  resposta: não existe algoritmo polinomial para A

- "prova de existência" versus "prova de não-existência"

- para a maioria dos problemas
  não sei fazer nem uma coisa nem outra
  não sei o problema está em **P** ou fora de **P**

- que fazer?

Complexidade relativa de problemas

- conceito de complexidade relativa:
  problema B mais difícil que problema A
  problema A mais fácil que problema B
- exemplo: problema A é subproblema de B
- em geral: redução polinomial entre problemas (veja adiante)
- antes de tratar de reduções,
  preciso restringir a classe de problemas

  não dá pra tratar de *todo e qualquer* problema...

Problemas "razoáveis" e a classe NP

- um problema é "razoável" se
  é fácil reconhecer uma solução do problema
  quando se está diante de uma

  para toda instância I do problema
  é possível verificar, em tempo polinomial,
  se uma suposta solução da instância I é uma solução de I
- a maioria dos problemas da lista acima é "razoável"
- problema "razoável" $\cong$ pertence à classe **NP**
- não confunda "NP" com "não-polinomial"
- daqui em diante, só trataremos de problemas em **NP**

A questão "P = NP?"

- evidente: $\mathbf{P} \subseteq \mathbf{NP}$
- ninguém mostrou ainda que $\mathbf{P} \neq \mathbf{NP}$
- será que $\mathbf{P} = \mathbf{NP}$?

Apêndice: certificados de inexistência de solução

- se um problema não tem solução
  como "certificar" isso?
- certificado de inexistência de solução
- diferença entre "certificar" e "resolver"
- exemplos:
  raiz quadrada, eq do segundo grau, col disj de intervalos,
  caminho de u a v, ciclo hamiltoniano, etc.
- certificados de maximalidade
- exemplos: MDC, etc.

Aulas 21–22

Complexidade computacional:

problemas de decisão,

as classes NP e NPC

CLRS cap. 34

P, NP e NP-completo

- antes de definir P e NP
  é preciso padronizar o tipo de problemas
- padronização: só problemas de decisão
- problema de decisão: toda instância tem resposta SIM ou NÃO
- SIM / NÃO $\cong$ tem solução / não tem solução
- instância positiva ... SIM
  instância negativa ... NÃO
- a teoria só trata de problemas de decisão

Exemplos de problemas de decisão

- EQ-SEGUNDO-GRAU:
  dados inteiros a, b, c, decidir se existe um inteiro x
  tal que $ax^2 + bx + c = 0$
- NÚMERO-COMPOSTO:
  dado um número natural $n > 1$,
  decidir se existem naturais $p > 1$ e $q > 1$ tais que $n = p \times q$
- NÚMERO-PRIMO:
  decidir se um dado natural $n > 1$ é primo

Note os problemas complementares

Mais exemplos de problemas de decisão

- INTERV-DISJS:
  dada coleção de intervalos e natural k,
  decidir se existe subcoleção disjunta com $\geq k$ intervalos
- SUBSEQ-CRESC-LONGA:
  dado inteiro positivo k e seqüência de inteiros positivos,
  decidir se alguma subseqüência crescente tem comprimento $\geq k$
- SUBSET-SUM:
  dados inteiros positivos $w_1, \ldots, w_n$ e W,
  decidir se existe $J \subseteq \{1, \ldots, n\}$ tal que $\sum_{j \in J} w_j = W$
- EMPARELH-BIPARTIDO:
  dado um grafo com bipartição (U, V),
  decidir se existe um emparelhamento que cobre U

Mais exemplos de problemas de decisão

- HAM-CYCLE:
  decidir se um grafo dado tem um ciclo hamiltoniano
- LONG-CYCLE:
  dado um grafo G e um natural k,
  decidir se G tem um ciclo simples de comprimento $\geq k$
- CLIQUE:
  dado um grafo G e um natural k,
  decidir se G tem uma clique com $\geq k$ vértices
- INDEPENDENT-SET:
  dado um grafo G e um natural k,
  decidir se G tem um conjunto independente com $\geq k$ vértices

Mais exemplos de problemas de decisão

- CLIQUE-COVER:
  dado um grafo G e um natural k,
  decidir se os vertices de G podem se cobertos por $\leq k$ cliques

Problemas de decisão não são tão "fracos" assim:

- suponha que HAM-CYCLE (G) tem resposta SIM
- como encontrar ciclo hamiltoniano num grafo G?
- para cada aresta e
      se HAM-CYCLE $(G - e)$ tem resposta SIM
          então $G \leftarrow G - e$
  G é ciclo hamiltoniano

Mais padronização

- instância: cadeias de caracteres
- tamanho de uma instância: comprimento da cadeia

Conseqüências:

- tamanho de um número W é $\sim \log W$
- o tamanho de um vetor $A[1\,..\,n]$ é $\sim \sum_{i=1}^{n} \log A[i]$

Exemplo:

```
PRIMO (n)
para i ← n − 1 decrescendo até 2 faça
    se i divide n
        então devolva "NÃO"
devolva "SIM"
```

Qual o tamanho de uma instância?
O algoritmo é polinomial?

Classe P

- todos os problemas de decisão
  que podem ser resolvidos por algoritmos polinomiais

Antes de definir a classe NP, preciso do conceito certificado

Certificado de problema de decisão

- certificado: é uma "prova" de que resposta SIM está certa
- certificado substitui o conceito de solução
- tamanho do certificado:
  limitado por polinômio no tamanho da instância
- exemplos de certificados:
  eq do segundo grau
  col disj de intervalos
  caminho de u a v
  ciclo hamiltoniano
  etc.

Algoritmo verificador para um problema de decisão:

- recebe instância I e candidato C a certificado
  e devolve SIM ou NÃO ou não pára
- se instância I é positiva
  então existe C que faz verificador devolver SIM
  em tempo polinomial no tamanho de I
- se instância I é negativa
  então não existe C que faça verificador devolver SIM

Certificado polinomial:
certificado para o qual existe algoritmo verificador

Classe NP

- todos os problemas de decisão
  cujas instâncias positivas têm certificados polinomiais
- grosseiramente: problema NP $\cong$ problema "razoável"

Problemas complementares e certificado de NÃO

- problema complementar: troque SIM por NÃO
- certificado de SIM de um problema $=$
  certificado de NÃO do complementar
- exemplos:
  RAIZ-QUADRADA e seu complemento
  EMPARELH-BIPARTIDO e "conjunto de Hall"
  SUBSEQ-CRESC-LONGA e cobert por subseqs estrit descresc
  INTERV-DISJS e cobertura por cliques de intervalos
  NÚMERO-PRIMO e NÚMERO-COMPOSTO

Classe co-NP

- todos os problemas de decisão
  cujos complementos estão em **NP**
- todos os problemas de decisão
  cujas instâncias negativas têm certificados polinomiais

Relação entre P, NP e co-NP

- fácil: $\mathbf{P} \subseteq \mathbf{NP}$ e $\mathbf{P} \subseteq \mathbf{co\text{-}NP}$
- ninguém sabe ainda: $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$

Reduções polinomiais entre problemas em NP

Redução polinomial de problema X a problema Y:

- algoritmo que transforma instâncias de X em instâncias de Y
- transforma instâncias positivas em positivas e negativas em negativas
- consome tempo polinomial no tamanho das instâncias de X

Notação: $X \leq_{\text{P}} Y$

$\langle$algor redução$\rangle$ + $\langle$algor polin para $Y\rangle$ = $\langle$algor polin para $X\rangle$

Redução mostra que

- X é tão fácil quanto Y
- Y é tão difícil quanto X

Exemplos de redução:

- RAIZ-QUADRADA $\leq_{\text{P}}$ EQ-SEGUNDO-GRAU
- INTERV-DISJS $\leq_{\text{P}}$ INDEPENDENT-SET
- HAM-CYCLE $\leq_{\text{P}}$ LONG-CYCLE
- HAM-PATH $\leq_{\text{P}}$ HAM-CYCLE
- HAM-CYCLE $\leq_{\text{P}}$ HAM-PATH
- CLIQUE $\leq_{\text{P}}$ INDEPENDENT-SET
- INDEPENDENT-SET $\leq_{\text{P}}$ CLIQUE
- HAM-CYCLE $\leq_{\text{P}}$ INDEPENDENT-SET

Problemas completos em NP

- problema Y é **NP-completo** se
  1. $Y \in$ NP e
  2. $X \leq_{\text{P}} Y$ para todo X em NP
- NP-completo $\cong$ tão difícil qto qquer problema em NP
- classe **NPC**: todos os problemas NP-completos
- evidente: **P = NP** <u>sse</u> $\mathbf{P} \cap \mathbf{NPC} \neq \emptyset$
- teorema de Cook & Levin: NPC não é vazio

Dado problema X

- eu gostaria de dizer que "$X \in$ P" ou que "$X \notin$ P"
- mas em geral só consigo dizer que "$X \in$ P" ou que "$X \in$ NPC"

Livro de Garey e Johnson

Exemplos de problemas em **NPC**:

- SUBSET-SUM
- MOCHILA
- HAM-CYCLE
- LONG-CYCLE
- LONG-PATH
- CLIQUE
- INDEPENDENT-SET
- CLIQUE-COVER
- etc.

Aula 23

Análise amortizada

Tabelas dinâmicas

CLRS cap. 17

Análise amortizada

Contexto:

- repertório de operações
- seqüência de n operações do repertório
- algumas operações são caras
- mas muitas são baratas e "compensam" as caras
- quero calcular o custo total C da seq de ops

Mais detalhes:

- i-ésima op tem um custo (= consumo de tempo) c_i
- quero calcular custo total $C := \sum_{i=1}^{n} c_i$
- cálculo da soma pode ser difícil
- imagine custo fictício constante $\hat{c}$ por operação tal que $C \leq \sum_{i=1}^{n} \hat{c} = \hat{c} \cdot n$
- $\hat{c}$ é o custo **amortizado** de uma operação

Aqui, "custo" = "tempo"

Análise amortizada: método contábil

Exemplo: Contador binário

A

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

em vermelho: bits que mudam de valor quando somo 1

$k-1$		3	2	1	0

```
INCREMENT (A, k)
  i ← 0
  enquanto i < k e A[i] = 1
      faça A[i] ← 0
              i ← i + 1
  se i < k
      então A[i] ← 1
```

custo = tempo = número de bits alterados $\leq k$

Seqüência de n chamadas de INCREMENT:

$$\underbrace{\text{INCR} \quad \text{INCR} \quad \cdots \quad \text{INCR} \quad \text{INCR} \quad \text{INCR}}_{n}$$

- custo total $\leq nk$
- a estimativa é exagerada
- vou mostrar que custo total é $\leq n$
- vou mostrar que custo amortizado de uma operação é 1

Exemplo com $k = 6$ e $n = 16$:

A

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

- $A[0]$ muda n vezes
- $A[1]$ muda $\lfloor n/2 \rfloor$ vezes
- $A[2]$ muda $\lfloor n/4 \rfloor$ vezes
- $A[3]$ muda $\lfloor n/8 \rfloor$ vezes
- etc.

custo total: $$\sum_{i=0}^{\lfloor \lg n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor \;<\; n \sum_{i=0}^{\infty} \frac{1}{2^i} \;=\; 2n$$

Como podemos calcular esse "$2n$" de maneira mais fácil?

Método contábil de análise:

- $c_i \;=\;$ custo real da i-ésima operação
- $\hat{c}_i \;=\;$ custo amortizado (fictício) da i-ésima operação
- defina $\hat{c}_i$ de modo que $\sum_i c_i \;\leq\; \sum_i \hat{c}_i$

Método contábil no caso do contador binário:

$$\text{INCR}_1 \;\; \text{INCR}_2 \;\; \cdots \;\; \text{INCR}_i \;\; \text{INCR}_{n-1} \;\; \text{INCR}_n$$

- $c_i \;=\;$ número de bits alterados
- $\hat{c}_i \;=\; 2$
- prova de $\sum c_i \leq \sum \hat{c}_i$:
  - pago \$2 por cada chamada de INCR
  - \$1 é gasto pelo único 0→1
  - \$1 fica sentado no bit 1 para pagar 1→0 no futuro
  - em cada instante, há \$1 sentado em cada bit 1
  - "crédito armazenado" nunca é negativo

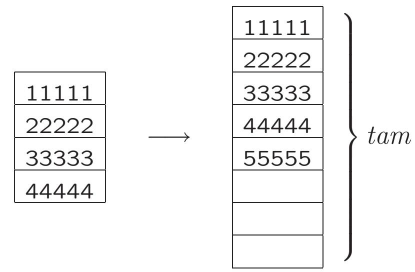
- conclusão: em cada instante, $\sum c_i \leq \sum \hat{c}_i$
- conclusão: $\hat{c}_i := 2$ está correto
- custo real total: $\sum_{i=1}^{n} c_i \;\leq\; \sum_{i=1}^{n} \hat{c}_i \;=\; 2n$

“Amortizado” não é “médio”

- não confunda “custo amortizado” com “custo médio”
- “amortizado” não envolve probabilidades

Exemplo 2: Tabela dinâmica

Tamanho da tabela aumenta (dobra) quando necessário



- $num[T]$ = número de itens
- $tam[T]$ = tamanho da tabela
- valores de tam: $0, 1, 2, 4, 8, 16, \ldots$

começa com $num[T] = tam[T] = 0$

TABLE-INSERT (T, x)

```
se tam[T] = 0
    então aloque tabela[T] com 1 posição
          tam[T] ← 1
se num[T] = tam[T]
    então aloque nova com 2 · tam[T] posições
          insira itens da tabela[T] na nova
          libere tabela[T]
          tabela[T] ← nova
          tam[T] ← 2 tam[T]
insira x na tabela[T]
num[T] ← num[T] + 1
```

custo = número de “insira”

Seqüência de n TABLE-INSERTS

- qual o custo total da seq de operações?
- custo real da i-ésima operação:

$$c_i = \begin{cases} 1 & \text{se há espaço} \\ i & \text{se tabela cheia (ou seja, se } i-1 \text{ é potência de 2)} \end{cases}$$

- custo total da seq de operações: $\sum c_i$
- para simplificar cálculo de $\sum c_i$, invente custo amortizado
- defina custo amortizado de i-ésima op: $\hat{c}_i := 3$
- preciso garantir que $\sum c_i \leq \sum \hat{c}_i$

- prova de $\sum c_i \leq \sum \hat{c}_i$:

  \$1 para para inserir novo item

  \$1 pagto adiantado para realocação do novo item

  \$1 pagto adiantado para realoc de um dos itens antigos

  "crédito acumulado" nunca é negativo

  logo, $\sum_{i=1}^{k} c_i \leq \sum_{i=1}^{k} \hat{c}_i$ para cada k

- conclusão: $\hat{c}_i = 3$ é um bom custo amortizado

- custo real total: $\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i = 3n$

Análise amortizada: método do potencial

Exemplo: Tabela dinâmica com *insert* e *delete*

- método contábil fica difícil

- introduz método da função potencial