CI165 - Análise de algoritmos
(rascunho alterado constantemente)

André Guedes
Departamento de Informática
UFPR

31 de agosto de 2017

Sumário

Aula 1

Apresentação do Curso

Incluir:

- Fortemente polinomial
- Quase polinomial
- Quase linear
- Hierarquia polinomial
- Amortizada

Objetivos: Apresentar um conjunto de técnicas de projeto e análise de algoritmos. A comparação de alternativas é sempre feita utilizando-se técnicas de análise de algoritmos. Ao final do curso o aluno deverá ser capaz de lidar com classes específicas de problemas e suas soluções eficientes, dominando as principais técnicas utilizadas para projetar e analisar algoritmos e sabendo decidir o que pode e o que não pode ser resolvido eficientemente pelo computador

1. avaliação: 2 provas de pesos iguais
2. programa
   (a) Introdução à análise de complexidade de algoritmos. Medidas de complexidade de algoritmos.
   (b) Análise assintótica de limites de complexidade. Crescimento de funções e comportamento assintótico.
   (c) Corretude de algoritmos.

(d) Classes de problemas computacionais.

(e) Notações e classes de complexidade padrão.

(f) Técnicas de análise de algoritmos.

(g) Técnicas de projeto de algoritmos.

(h) Tratabilidade e tópicos atuais sobre tratamento de problemas difíceis.

3. bibliografia

Aula 2

Problemas computacionais e algoritmos

Problema computacional;
Instância, saída, tamanho da instância;
Algoritmo;
Modelo RAM;
Contagem de passos;
Comparação de algoritmos.

Aula 3

Notação assintótica - $\mathcal{O}$

Definição 1. *Dada uma função $g : \mathbb{R} \to \mathbb{R}$, defina o conjunto de funções*

$$\mathcal{O}(g(n)) = \{f : \mathbb{R} \to \mathbb{R} \mid \exists c > 0 \in \mathbb{R}, n_0 \in \mathbb{N}, |f(n)| \leq c|g(n)|, \text{ para todo } n \geq n_0\}.$$

O fato de uma função $f : \mathbb{R} \to \mathbb{R}$ estar em $\mathcal{O}(g(n))$ é denotado por $f(n) \in \mathcal{O}(g(n))$; $f(n) = \mathcal{O}(g(n))$; $f(n) \simeq \mathcal{O}(g(n))$ ou $f(n)$ é $\mathcal{O}(g(n))$.

Se $f(n) \in \mathcal{O}(g(n))$ dizemos que $g(n)$ é um limitante assintótico superior para $f(n)$. Ou seja, quando n cresce, $g(n)$ cresce mais ou igual que $f(n)$.

Considere $f, g, h : \mathbb{R} \to \mathbb{R}$ funções, $n, k, \ell \in \mathbb{N}$ e $\alpha, \beta \in \mathbb{R}$.

Exemplos

1. $n \in \mathcal{O}(n)$, com $c = 1$ e $n_0 = 0$;
2. $f(n) \in \mathcal{O}(f(n))$, com $c = 1$ e $n_0 = 0$;
3. $n \in \mathcal{O}(n^2)$, com $c = 1$ e $n_0 = 1$;
4. $n^k \in \mathcal{O}(n^{k+1})$, com $c = 1$ e $n_0 = 1$;
5. $n^k \in \mathcal{O}(n^\ell)$, para $\ell \geq k$, com $c = 1$ e $n_0 = 1$;
6. $\alpha n^k \in \mathcal{O}(n^\ell)$, para $\ell \geq k$, com $c = \alpha$ e $n_0 = 0$;
7. $\alpha n^k + \beta \in \mathcal{O}(n^\ell)$, para $\ell \geq k$, com $c = \alpha + \beta$ e $n_0 = 1$;
8. $g(n) + f(n) \in \mathcal{O}(g(n))$, para $f(n) \in \mathcal{O}(g(n))$;
9. se $f(n) \in \mathcal{O}(h(n))$ e $g(n) \in \mathcal{O}(h(n))$, então $f(n) + g(n) \in \mathcal{O}(h(n))$;

10. (transitividade) se $f(n) \in \mathcal{O}(g(n))$ e $g(n) \in \mathcal{O}(h(n))$, então $f(n) \in \mathcal{O}(h(n))$; ou seja, se $f(n) \in \mathcal{O}(g(n))$ então $\mathcal{O}(f(n)) \subseteq \mathcal{O}(g(n))$;

11. (polinômios) $\sum_{i=0}^{k} \alpha_i n^i \in \mathcal{O}(n^k)$.

12. $\log_b n \in \mathcal{O}(\log_a n)$, para $a, b > 1$
(lembre que $\log_b n = (\log_b a)(\log_a n)$);

13. $\log_b n \in \mathcal{O}(n)$, para $b > 1$;

14. $\mathcal{O}(1) \subseteq \mathcal{O}(\log n) \subseteq \mathcal{O}(n) \subseteq \mathcal{O}(n^2) \subseteq \cdots \subseteq \mathcal{O}(n^k)$, para $k \geq 2$;

15. (soma) se $f_1(n) \in \mathcal{O}(f(n))$ e $g_1(n) \in \mathcal{O}(g(n))$ então $f_1(n) + g_1(n) \in \mathcal{O}(f(n) + g(n))$;

16. (muliplicação) se $f_1(n) \in \mathcal{O}(f(n))$ e $g_1(n) \in \mathcal{O}(g(n))$ então $f_1(n)g_1(n) \in \mathcal{O}(f(n)g(n))$;

Aula 4

Notação assintótica - Ω e Θ

Definição 2 (Definição de Knuth (1976))**.** *Dada uma função $g : \mathbb{R} \to \mathbb{R}$, defina o conjunto de funções*

$$\Omega(g(n)) = \{f : \mathbb{R} \to \mathbb{R} \mid \exists c > 0 \in \mathbb{R}, n_0 \in \mathbb{N}, |f(n)| \geq c|g(n)|, \text{ para todo } n \geq n_0\}.$$

O fato de uma função $f : \mathbb{R} \to \mathbb{R}$ estar em $\Omega(g(n))$ é denotado por $f(n) \in \Omega(g(n))$; $f(n) = \Omega(g(n))$; $f(n) \simeq \Omega(g(n))$ ou $f(n)$ é $\Omega(g(n))$.

Se $f(n) \in \Omega(g(n))$ dizemos que $g(n)$ é um limitante assintótico inferior para $f(n)$. Ou seja, quando n cresce, $g(n)$ cresce menos ou igual que $f(n)$.

Exemplos

1. $n \in \Omega(n)$;
2. $f(n) \in \Omega(f(n))$;
3. $n^2 \in \Omega(n)$;
4. $n^{k+1} \in \Omega(n^k)$;
5. $n^k \in \Omega(n^\ell)$, para $\ell \leq k$;
6. $\alpha n^k \in \Omega(n^\ell)$, para $\ell \leq k$;
7. $\alpha n^k + \beta \in \Omega(n^\ell)$, para $\ell \leq k$;
8. (transitividade) se $f(n) \in \Omega(g(n))$ e $g(n) \in \Omega(h(n))$ então $f(n) \in \Omega(h(n))$;
9. (inclusão) se $f(n) \in \Omega(g(n))$ então $\Omega(f(n)) \subseteq \Omega(g(n))$;

10. (polinômios) $\sum_{i=0}^{k} \alpha_i n^i \in \Omega(n^k)$;

11. $\log_b n \in \Omega(\log_a n)$, para $a, b > 1$;

12. $n \in \Omega(\log_b n)$, para $b > 1$;

13. $\Omega(n^k) \subseteq \cdots \subseteq \Omega(n^2) \subseteq \Omega(n) \subseteq \Omega(\log_b n) \subseteq \Omega(1)$, para $b > 1$ e $k \geq 2$;

14. (muliplicação) se $f_1(n) \in \Omega(f(n))$ e $g_1(n) \in \Omega(g(n))$ então $f_1(n)g_1(n) \in \Omega(f(n)g(n))$.

Definição 3. *Dada uma função $g : \mathbb{R} \to \mathbb{R}$, defina o conjunto de funções*

$$\Theta(g(n)) = \mathcal{O}(g(n)) \cap \Omega(g(n)).$$

O fato de uma função $f : \mathbb{R} \to \mathbb{R}$ estar em $\Theta(g(n))$ é denotado por $f(n) \in \Theta(g(n))$; $f(n) = \Theta(g(n))$; $f(n) \simeq \Theta(g(n))$ ou $f(n)$ é $\Theta(g(n))$.

Se $f(n) \in \Theta(g(n))$ dizemos que $g(n)$ é um limitante assintótico justo para $f(n)$. Ou seja, quando n cresce, $g(n)$ cresce da mesma forma que $f(n)$.

1. $f(n) \in \Theta(f(n))$;

2. (simetria) se $f(n) \in \Theta(g(n))$ então $g(n) \in \Theta(f(n))$;

3. $\sum_{i=1}^{n} i \in \Theta(n^2)$;

Aula 5

Notação assintótica (cont)

Considere duas funções $f, g : \mathbb{R} \to \mathbb{R}$.

Definição 4. *As funções f e g são assintoticamente equivalentes se $f \in \Theta(g)$. Além disso:*

1. *f é assintoticamente constante se f é assintoticamente equivalente a uma função constante, ou seja, $f \in \Theta(1)$;*
2. *f é assintoticamente linear se f é assintoticamente equivalente a uma função linear, ou seja, $f \in \Theta(n)$;*
3. *f é assintoticamente polinomial se f é assintoticamente equivalente a um polinômio, ou seja, $f \in \Theta(n^k)$, para algum $k \in \mathbb{N}$;*
4. *f é assintoticamente logarítmica se f é assintoticamente equivalente a uma função logarítmica, ou seja, $f \in \Theta(\log n)$;*
5. *f é assintoticamente exponencial se f é assintoticamente equivalente a uma função exponencial, ou seja, $f \in \Theta(b^n)$, para algum $b > 1$;*

Definição 5. *A função f é* assintoticamente menor *que g se a razão f/g for assintoticamente nula, isto é, se*

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0.$$

Notação 1. *O conjunto das funções* assintoticamente menores *que g é denotado por $o(g)$, isto é,*

$$o(g(n)) = \{f : \mathbb{R} \to \mathbb{R} \mid \lim_{n\to\infty} \frac{g(n)}{f(n)} = 0\}.$$

Se $f(n) \in o(g(n))$ dizemos também que $f(n) \ll g(n)$.

Definição 6. *As funções f e g são aproximadamente iguais se*

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1.$$

Denotamos o fato de que f e g são aproximadamente iguais por

$$f(n) \approx g(n).$$

1. $n^2 + n \in \Theta(n^2)$;
2. $10 - \frac{1}{n} \in \Theta(1)$;
3. $\sum_{i=1}^{n} i \in \Theta(n^2)$;
4. Se $f(n) \in \Theta(n)$, então $\sum_{i=1}^{n} f(i) \in \Theta(n^2)$.

Definição 7. *A função g é* assintoticamente maior *que f se f for assintoticamente menor que g.*

Definição 8. *Uma função é* sub–[linear/polinomial/exponencial/logaritmica] *se é assintoticamente menor que uma função assintoticamente linear/polinomial/exponencial/logaritmica.*

Definição 9. *Uma função é* super–[linear/polinomial/exponencial/logaritmica] *se é assintoticamente maior que uma função assintoticamente linear/polinomial/exponencial/logaritmica.*

Teorema 1. *Toda função assintoticamente exponencial é superpolinomial.*

Demonstração. Exercício. □

Teorema 2. *$n!$ é superexponencial.*

Demonstração. Exercício. □

Teorema 3. *$n \lg n$ é subquadrática.*

Demonstração. Exercício. □

Aula 6

Classes de Problemas Computacionais

Classes $\mathbb{P}$ e $\mathcal{NP}$, reduções, classes $\mathcal{NP}$–Completo e $\mathcal{NP}$–Difícil. Hierarquia de classes.

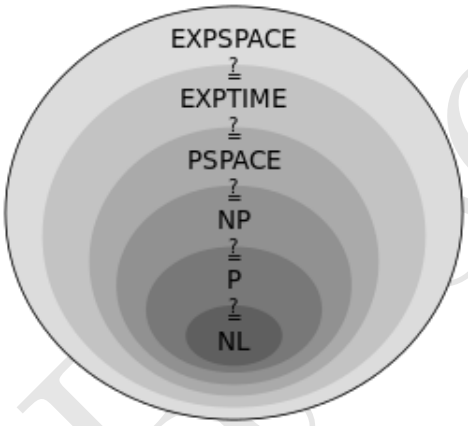

Figura 6.1: Hierarquia de Classes (https://en.wikipedia.org/wiki/Computational_complexity_theory)

Aula 7

Análise de algoritmos

Um problema computacional é caracterizado pela descrição do

1. conjunto de possíveis entradas,
2. conjunto de possíveis saídas para cada possível entrada.

Notação 2. *Se C é um conjunto, 2^C denota o* conjunto dos subconjuntos *ou* conjunto das partes *de C, isto é,*

$$2^C = \{S \mid S \subseteq C\}.$$

Definição 10. *Um* problema computacional *é uma tripla $P = (\mathcal{I}(P), \mathcal{S}(P), f_P)$ onde*

$\mathcal{I}(P)$ *é um conjunto, chamado de* conjunto das instâncias do problema P.

$\mathcal{S}(P)$ *é um conjunto, chamado de* conjunto das respostas (ou soluções) do problema P

f_P *é uma função $\mathcal{I}(P) \to 2^{\mathcal{S}(P)}$, que associa a cada instância $I \in \mathcal{I}(P)$ o* conjunto de soluções $f_P(I) \subseteq \mathcal{S}(P)$ da instância I do problema P.

Um algoritmo executa um certo conjunto de passos. A complexidade de tempo de um algoritmo é a quantidade de passos que ele executa. A complexidade de tempo é uma função da entrada.

Para simplificar agrupamos entradas com o mesmo tamanho. Mas a complexidade deixa de ser uma função, pois para entradas de mesmo tamanho o algoritmo pode usar quantidades diferentes de passos. Para resolver isso usamos o pior caso, o melhor caso e o caso médio.

Seja A um algoritmo que roda com entradas de um conjunto I. Considere que uma entrada $X \in I$ tem tamanho $|X|$. Se a complexidade de tempo de A é dada pela função $T_A : I \to \mathbb{R}$, então:

(pior caso) $T_A^+(n) = max\{T_A(X) \mid X \in I, |X| = n\}$;

(melhor caso) $T_A^-(n) = min\{T_A(X) \mid X \in I, |X| = n\}$;

(caso médio) $T_A^m(n) = \frac{\sum_{X \in I, |X|=n} T_A(X)}{|\{X \in I || X| = n\}|}$, caso o número de instâncias de tamanho n seja finito. Mais especificamente, $T_A^m(n)$ é a esperança do custo para as instâncias de tamanho n.

Um algoritmo A é dito ser $\mathcal{O}(g(n))$ se $T_A^+(n) \in \mathcal{O}(g(n))$; ser $\Omega(g(n))$ se $T_A^-(n) \in \Omega(g(n))$; e ser $\Theta(g(n))$ se é $\mathcal{O}(g(n))$ e $\Omega(g(n))$.

Perceba que, se um algoritmo A é $\Theta(g(n))$ então $T_A^+(n) \in \Theta(g(n))$, $T_A^-(n) \in \Theta(g(n))$ e $T_A^m(n) \in \Theta(g(n))$. Além disso, $T_A^-(n) \in \Theta(T_A^+(n))$ e $T_A^+(n) \in \Theta(T_A^-(n))$.

Note que isso se trata de comportamento assintótico do número de passos de um algoritmos para o seu universo de instâncias. Se uma certa instância X está fixada, o número de passos (como estamos tratando de um modelo computacional determinístico) é constante, $T_A(X)$, e portanto não é significativo dizer que $T_A(X) \in \mathcal{O}(g(n))$ (ou Ω ou Θ).

7.1 Analisando um Algoritmo

Notação 3. *Dados $a, b \in \mathbb{Z}$, $[a..b]$ denota o conjunto dos inteiros entre a e b (inclusives), isto é,*

$$[a..b] = \{z \in \mathbb{Z} \mid a \leq z \leq b\}.$$

Busca em Vetor Ordenado (BVO)

Entrada: Dois inteiros a e b, um vetor v satisfazendo $v[i] \leq v[i+1]$ para todo $a \leq i < b$, e um valor x.

Saída: um inteiro $i \in [a..b]$ satisfazendo $v[i] = x$ ou `não` caso não exista tal inteiro.

Uma instância do BVO é uma quádrupla (a, b, v, x) onde

a é um inteiro,

b é um inteiro,

v é um vetor de valores cujos índices incluem $[a..b]$ e

x é um valor.

O conjunto $\mathcal{I}(\mathsf{BVO})$ das instâncias de BVO é o conjunto de todas as quádruplas (a, b, v, x) como acima.

Dada uma instância $I = (a, b, v, x)$ do BVO, s é uma solução da instância I se

- $s \in [a..b]$ e $v[s] = x$ ou,
- $s = \ \mathsf{não}\ $ e $v[i] \neq x$ para todo $i \in [a..b]$.

Portanto, o conjunto $\mathcal{S}(\mathsf{BVO})$ das soluções do BVO é o conjunto $\mathbb{Z} \cup \{\ \mathsf{não}\ \}$. Para cada instância $I = (a, b, v, x) \in \mathcal{I}(\mathsf{BVO})$ temos um conjunto das soluções da instância I, dado por

$$f_{\mathsf{BVO}}(a, b, v, x) = \begin{cases} \{\ \mathsf{não}\ \}, & \text{se } v[i] \neq x \text{ para todo } i \in [a..b] \\ \{i \in [a..b] \mid v[i] = x\}, & \text{caso contrário.} \end{cases}$$

Portanto, $\mathsf{BVO} = (\mathcal{I}(\mathsf{BVO}), \mathcal{S}(\mathsf{BVO}), f_{\mathsf{BVO}})$, onde $\mathcal{I}(\mathsf{BVO})$, $\mathcal{S}(\mathsf{BVO})$ e f_{BVO} são os definidos acima.

Exemplo 1.

$$v[0..9] = (9, 7, 2, 4, 4, 8, 10, 5, 3, 1).$$

$$I = (2, 6, v, 4) \in \mathcal{I}(\mathit{BVO})$$

é uma instância do BVO.

$$f_{\mathit{BVO}}(2, 6, v, 4) = \{3, 4\} \subseteq \mathcal{S}(\mathit{BVO})$$

é o conjunto de soluções da instância $I = (2, 6, v, 4)$ *do* BVO.

$$I' = (4, 6, v, 2) \in \mathcal{I}(\mathit{BVO})$$

é outra instância do BVO.

$$f_{\mathit{BVO}}(4, 6, v, 2) = \{\ \mathit{não}\ \} \subseteq \mathcal{S}(\mathit{BVO})$$

é o conjunto de soluções da instância $I' = (4, 6, v, 2)$ *do* BVO.

7.1.1 Algoritmos

Algoritmo 1: $A(a, b, v, x)$

```
i ← a
Enquanto i ≤ b
    Se v[i] = x
        Devolva i
    i ← i + 1
Devolva não
```

O Algoritmo A é um *algoritmo para o BVO* porque para toda instância (a, b, v, x) do BVO, $A(a, b, v, x)$ é uma solução dessa instância.

Definição 11. *Seja P um problema computacional. Um* algoritmo A para o problema P *é uma função computável $A\colon \mathcal{I}(P) \to \mathcal{S}(P)$ satisfazendo*

$$A(I) \in f_P(I),$$

para toda instância $I \in \mathcal{I}(P)$.

Algoritmo 2: $B(a, b, v, x)$

```
i ← a
Enquanto i ≤ b e v[i] ≤ x
    Se v[i] = x
        Devolva i
    i ← i + 1
Devolva não
```

O Algoritmo B é outro *algoritmo para o BVO* pelas mesmas razões.

Qual dos dois é melhor?

"Melhor"? Em que sentido?

Qual precisa de menos recursos computacionais:

- tempo de execução
- memória
- banda de rede
- etc.

O "melhor" algoritmo num sentido pode ser o pior em outro.

Com relação ao nosso exemplo do BVO, vamos nos concentrar na medida de tempo de execução.

7.2 Medida de Desempenho de Algoritmos

Vamos denotar por

t_a**:** tempo de processamento para executar uma atribuição

t_{ci}**:** tempo de processamento para executar uma comparação entre índices de v

t_{cv}**:** tempo de processamento para executar uma comparação entre elementos de v

t_r**:** tempo de processamento para o término da execução (inclusive transmissão do resultado)

t_s**:** tempo de processamento para efetuar uma soma

e daí, vamos denotar por

$T_A(a, b, v, x)$**:** tempo de processamento na execução de $A(a, b, v, x)$

$T_B(a, b, v, x)$**:** tempo de processamento na execução de $B(a, b, v, x)$

Do exame direto do algoritmo A temos

$$T_A(a, b, v, x) = t_a + (e_A(a, b, v, x) - 1)(t_{ci} + t_{cv} + t_s + t_a) + t_r$$

onde $e_A(a, b, v, x)$ indica o número de vezes que o laço do "enquanto" é repetido na execução de $A(a, b, v, x)$.

Para simplificar, vamos fazer

$$\begin{aligned} c_1 &= t_a + t_r \\ c_2 &= t_{ci} + t_{cv} + t_a + t_s \end{aligned}$$

de forma que

$$T_A(a, b, v, x) = c_1 + (e_A(a, b, v, x) - 1)c_2 = (c_1 - c_2) + c_2 e_A(a, b, v, x).$$

Fazendo

$$c_3 = c_1 - c_2 = t_a + t_r - t_{ci} - t_{cv} - t_a - t_s = t_r - (t_s + t_{ci} + t_{cv}),$$

fica

$$T_A(a, b, v, x) = c_3 + c_2 e_A(a, b, v, x).$$

Fazendo a mesma conta para o Algoritmo B, temos

$$\begin{aligned} T_B(a,b,v,x) &= t_a + (e_B(a,b,v,x) - 1)(t_{ci} + t_{cv} + t_{cv} + t_s + t_a) + t_r \\ &= c_1 + (c_2 + t_{cv})e_B(a,b,v,x) - (c_2 + t_{cv}) \\ &= c_3 + t_{cv} + (c_2 + t_{cv})e_B(a,b,v,x). \end{aligned}$$

Fazendo

$$\begin{aligned} c'_2 &= c_2 + t_{cv} \\ c'_3 &= c_3 + t_{cv}, \end{aligned}$$

fica

$$T_B(a,b,v,x) = c'_3 + c'_2 e_B(a,b,v,x).$$

Em resumo, temos

$$\begin{aligned} T_A(a,b,v,x) &= c_3 + c_2 e_A(a,b,v,x), \\ T_B(a,b,v,x) &= c'_3 + c'_2 e_B(a,b,v,x). \end{aligned}$$

No estudo analítico acima, c_2, c_3, c'_2 e c'_3 são valores (medidas de tempo) que só dependem da implementação (arquitetura do computador, sistema operacional, qualidade do compilador etc). Não dizem nada a respeito dos Algoritmos A e B.

Por outro lado, observando as expressões acima, podemos ver que os elementos significativos na expressão do tempo de execução dos algoritmos A e B são os valores de $e_A(a,b,v,x)$ e $e_A(a,b,v,x)$, que são *grandezas adimensionais* e que só dizem respeito aos algoritmos A e B e aos valores de a, b, v e x.

Aula 8

Pior caso, Melhor caso e Caso médio do BVO

Assuma que $|(a, b, v, x)| = n$ fixo.

8.1 Pior caso

Para que instâncias $T_A(a, b, v, x)$ tem o pior custo? E $T_B(a, b, v, x)$?

...

8.2 Melhor caso

Para que instâncias $T_A(a, b, v, x)$ tem o melhor custo? E $T_B(a, b, v, x)$?

...

8.3 Caso médio

Quais os possíveis valores de $T_A(a, b, v, x)$? E de $T_B(a, b, v, x)$?
$T_A(a, b, v, x) \in [1..n + 1]$
$T_B(a, b, v, x) \in [1..n + 1]$

...

Aula 9

Máximo de vetor MV

Máximo de Vetor (MV)

Entrada: Dois inteiros a e b, e um vetor v indexado por $[a..b]$.

Saída: Um inteiro $m \in [a..b]$ satisfazendo $v[m] \geq v[i]$ para todo $i \in [a..b]$.

Uma instância do MV é uma tríplа (a, b, v) onde

a é um inteiro,

b é um inteiro e

v é um vetor de valores cujos índices incluem $[a..b]$.

O conjunto $\mathcal{I}(\mathsf{MV})$ é o conjunto de todas as tríplas (a, b, v) como acima. O conjunto $\mathcal{S}(\mathsf{MV})$ é o conjunto $\mathbb{Z}$. Para cada instância $I = (a, b, v) \in \mathcal{I}(\mathsf{MV})$ temos um conjunto das soluções da instância I, dado por

$$f_{\mathsf{MV}}(a, b, v) = \{m \in [a..b] \mid v[m] \geq v[i], \text{ para todo } i \in [a..b]\}.$$

Algoritmo 3: $MV(a, b, v)$

```
m ← a
Para i ← a + 1 até b
    Se v[i] > v[m]
        m ← i
Devolva m
```

O tamanho da entrada é $n = b - a + 1$, que é o número de elementos do vetor v.

Usando as mesmas constantes do problema BVO, temos:

$$T_{MV}(a,b,v) = (t_a + t_a + t_s + t_r) + e_{MV}(a,b,v)(t_a + t_{cv} + t_s + t_a)$$

onde $e_{MV}(a,b,v)$ indica o número de vezes que o laço do "for" é repetido na execução de $MV(a,b,v)$.

Mas $e_{MV}(a,b,v)$ é sempre igual a $b-a = n-1$, e portanto temos que

$$T_{MV}(a,b,v) = (t_a + t_a + t_s + t_r) + (t_a + t_{cv} + t_s + t_a)(n-1)$$

Para simplificar, vamos fazer

$$\begin{aligned} c_1 &= t_a + t_a + t_s + t_r \\ c_2 &= t_a + t_{cv} + t_s + t_a \end{aligned}$$

de forma que

$$T_{MV}(a,b,v) = c_1 + c_2(n-1) = (c_1 - c_2) + c_2 n = c_2 n + c_3$$

se fizermos $c_3 = c_1 - c_2$.

Analisando pior caso, melhor caso e caso médio temos todos são iguais, ou seja:

$$T_{MV}(n) = T^+_{MV}(n) = T^-_{MV}(n) = T^m_{MV}(n) = c_2 n + c_3.$$

E, portanto, $T_{MV}(n) \in \Theta(n)$.

Algoritmo 4: $Mult(x,y)$

```
z ← 0
Enquanto y > 0
    Se y é ímpar
        z ← z + x
    x ← 2 * x
    y ← ⌊y/2⌋
Devolva z
```

Aula 10

Algoritmos com laços e somatórios

A fazer...

Aula 11

Algoritmos recursivos

Algoritmos recursivos não são algoritmos que usam recursão, são algoritmos que são desenvolvidos baseados em resolver uma instância usando a resposta de instâncias "menores" até um limite mínimo, a "base".

Exemplos:

- MergeSort
- Fibonacci

Algoritmo 5: $q(x)$

Se $x = 1$ *ou* $x = 2$
    Faz algo $\Theta(1)$
Senão
    $q(x-1)$
    Para $i \leftarrow 1$ *até* x
        Faz outro algo $\Theta(1)$

$$T_q(x) = \begin{cases} \Theta(1), & \text{se } x \in \{1,2\} \\ T_q(x-1) + x\Theta(1), & \text{caso contrário.} \end{cases}$$

$$
\begin{aligned}
T_q(x) &= T_q(x-1) + \Theta(x) \\
&= T_q(x-2) + \Theta(x-1) + \Theta(x) \\
&= T_q(x-3) + \Theta(x-2) + \Theta(x-1) + \Theta(x) \\
&= T_q(x-3) + \sum_{i=0}^{2} \Theta(x-i) \\
&= \cdots \\
&= T_q(x-u) + \sum_{i=0}^{u-1} \Theta(x-i),
\end{aligned}
$$

onde

$$u = \min\{k \mid x - k \leq 2\} = x - 2.$$

Ou seja,

$$
\begin{aligned}
T_q(x) &= T_q(x-u) + \sum_{i=0}^{u-1} \Theta(x-i) \\
&= T_q(2) + \sum_{i=0}^{x-3} \Theta(x-i) \\
&= \Theta(1) + \sum_{i=0}^{x-3} \Theta(x-i) \\
&= \Theta(1) + \Theta(x^2) \\
&= \Theta(x^2).
\end{aligned}
$$

Como o tamanho da entrada é $n = \lfloor \log x \rfloor + 1$, temos que

$$
\begin{aligned}
n &= \lfloor \log x \rfloor + 1 \\
n - 1 &= \lfloor \log x \rfloor \\
n - 1 \leq \log x &< n \\
2^{n-1} \leq 2^{\log x} &< 2^n \\
2^{n-1} \leq x &< 2^n.
\end{aligned}
$$

Logo,

$$T_q(n) \in \Omega(2^{2(n-1)}) = \Omega(2^{2n}/4)$$

e

$$T_q(n) \in \mathcal{O}(2^{2n})$$

Ou seja,

$$T_q(n) \in \Theta(2^{2n}).$$

Algoritmo 6: $h(x)$

```
Se x ≤ 1
    Faz algo Θ(1)
Senão
    Se x = 2
        Faz outro algo Θ(1)
    Senão
        Para i ← 1 até x
            h(x − 1)
            Faz algo diferente Θ(1)
```

Algoritmo 7: $m(x, y)$

```
Se y = 0
    Devolva 0
Senão
    Se y é ímpar
        Devolva m(2x, ⌊y/2⌋) + x
    Senão
        Devolva m(2x, ⌊y/2⌋)
```

Aula 12

Mais exemplos de recursivos

Algoritmo 8: $M(a, b, v)$

Se $a = b$
  Devolva $v[a]$
Senão
  $m \leftarrow \left\lfloor \frac{a+b}{2} \right\rfloor$
  $r_1 \leftarrow M(a, m, v)$
  $r_2 \leftarrow M(m+1, b, v)$
  $r \leftarrow \max\{r_1, r_2\}$
  Devolva r

Não depende de v, só de a e b. Ou seja, só depende do tamanho da entrada $n = b - a + 1$.

Sejam n_1 e n_2 os tamanhos de cada um dos subvetores, $[a..m]$ e $[m+1..b]$.

Então,

$$\begin{aligned} n_1 &= \left\lfloor \frac{a+b}{2} \right\rfloor - a + 1 \\ &= \left\lfloor \frac{a+b}{2} - a + 1 \right\rfloor \\ &= \left\lfloor \frac{a+b-2a+2}{2} \right\rfloor \\ &= \left\lfloor \frac{b-a+2}{2} \right\rfloor \\ n_1 &= \left\lfloor \frac{n+1}{2} \right\rfloor \end{aligned}$$

e

$$\begin{aligned} n_2 &= b - (\left\lfloor \frac{a+b}{2} \right\rfloor + 1) + 1 \\ &= b - \left\lfloor \frac{a+b}{2} \right\rfloor - 1 + 1 \\ &= b + \left\lceil -\frac{a+b}{2} \right\rceil \\ &= \left\lceil b - \frac{a+b}{2} \right\rceil \\ &= \left\lceil \frac{2b-a-b}{2} \right\rceil \\ &= \left\lceil \frac{b-a}{2} \right\rceil \\ n_2 &= \left\lceil \frac{n-1}{2} \right\rceil \end{aligned}$$

$$T_M(n) = \begin{cases} \Theta(1), & \text{se } n \leq 1 \\ T_M(\lfloor \frac{n+1}{2} \rfloor) + T_M(\lceil \frac{n-1}{2} \rceil) + \Theta(1), & \text{caso contrário.} \end{cases}$$

Como $T_M(n)$ é não decrescente e $\lfloor (n+1)/2 \rfloor \geq \lceil (n-1)/2 \rceil$, então $T_M(\lfloor (n+1)/2 \rfloor) \geq T_M(\lceil (n-1)/2 \rceil)$. Portanto,

$$T_M(n) \leq \begin{cases} \Theta(1), & \text{se } n \leq 1 \\ 2T_M(\lfloor \frac{n+1}{2} \rfloor) + \Theta(1), & \text{caso contrário.} \end{cases}$$

Ou

$$T_M(n) \leq f(n) = \begin{cases} c_1, & \text{se } n \leq 1 \\ 2f(\lfloor \frac{n+1}{2} \rfloor) + c_2, & \text{caso contrário.} \end{cases}$$

$$
\begin{aligned}
f(n) &= 2f(\left\lfloor \frac{n+1}{2} \right\rfloor) + c_2 \\
&= 2(2f(\left\lfloor \frac{\frac{n+1}{2}+1}{2} \right\rfloor) + c_2) + c_2 \\
&= 2^2 f(\left\lfloor \frac{n+1+2}{2^2} \right\rfloor) + 2c_2 + c_2 \\
&= 2^2(2f(\left\lfloor \frac{\frac{n+1+2}{2^2}+1}{2} \right\rfloor) + c_2) + 2c_2 + c_2 \\
&= 2^3 f(\left\lfloor \frac{n+1+2+4}{2^3} \right\rfloor) + 2^2 c_2 + 2c_2 + c_2 \\
&= 2^3 f(\left\lfloor \frac{n+1+2+4}{2^3} \right\rfloor) + 2^2 c_2 + 2c_2 + c_2 \\
&= 2^u f(\left\lfloor \frac{n+\sum_{i=0}^{u-1} 2^i}{2^u} \right\rfloor) + c_2 \sum_{i=0}^{u-1} 2^i \\
&= 2^u f(\left\lfloor \frac{n+2^u-1}{2^u} \right\rfloor) + c_2(2^u - 1)
\end{aligned}
$$

para $u = \min\left\{k \in \mathbb{N} \mid \left\lfloor \frac{n+2^k-1}{2^k} \right\rfloor \leq 1\right\}$, ou seja,

$$
\begin{aligned}
\left\lfloor \frac{n+2^k-1}{2^k} \right\rfloor &\leq 1 \\
\frac{n+2^k-1}{2^k} &< 2 \\
(n+2^k-1) &< 2^{k+1} \\
(n+2^k-1) &< 2^{k+1} \\
n-1 &< 2^{k+1} - 2^k \\
n-1 &< 2^k \\
\log(n-1) &< k.
\end{aligned}
$$

Portanto $u = \lfloor \log(n-1) \rfloor + 1$, para $n > 1$.

Logo, $f(n) = c_1 2^{\lfloor \log(n-1) \rfloor + 1} + c_2(2^{\lfloor \log(n-1) \rfloor + 1} - 1)$, para $n \geq 2$ e $f(1) = c_1$.

Como $\log(n-1) - 1 < \lfloor \log(n-1) \rfloor \leq \log(n-1)$, temos que

$$
\begin{aligned}
2^{\log(n-1)-1+1} &< 2^{\lfloor \log(n-1) \rfloor + 1} &\leq 2^{\log(n-1)+1} \\
n-1 &< 2^{\lfloor \log(n-1) \rfloor + 1} &\leq 2(n-1).
\end{aligned}
$$

Portanto, para $n \geq 2$,

$$\begin{aligned} f(n) &\leq c_1 2(n-1) + c_2(2(n-1)-1) \\ f(n) &\leq 2c_1 n - 2c_1 + 2c_2 n - 3c_2 \\ f(n) &\leq 2(c_1+c_2)n - 2c_1 - 3c_2 \end{aligned}$$

Fazendo $c = 2(c_1 + c_2)$ temos $f(n) \leq cn - c - c_2$. Logo, $T_M(n) \in \mathcal{O}(n)$.

Alternativamente podemos provar que $T_M(n) \in \Theta(n)$.

Primeiramente vamos provar que $T_M(n) \in \mathcal{O}(n)$. Lembrando que $T_M(n) = T_M(\lfloor \frac{n+1}{2} \rfloor) + T_M(\lceil \frac{n-1}{2} \rceil) + \Theta(1)$. Portanto $T_M(n) \leq T_M(\lfloor \frac{n+1}{2} \rfloor) + T_M(\lceil \frac{n-1}{2} \rceil) + c'$, para $n \geq n_0$, para algum $n_0 \in \mathbb{N}$. Assim, vamos provar por indução que $T_M(n) \leq cn - d$, para $n \geq n_0$, para $c > 0$, $d > 0$ e $n_0 \geq 2$ constantes.

Base:

Como $T_M(n_0)$ é um constante (já que n_0 é constante), seja $c_0 = T_M(n_0)$. Logo existem constantes c e d tais que $T_M(n_0) \leq cn_0 - d$.

Hipótese de indução:

Para algum $k \geq n_0$, $T_M(n) \leq cn - d$ para todo $n_0 \leq n \leq k$. Em particular, para $k = n_0$ a hipótese é verdadeira.

Passo de indução:

Suponha $n = k + 1$.

$$\begin{aligned} T_M(n) &= T_M(\lfloor \tfrac{n+1}{2} \rfloor) + T_M(\lceil \tfrac{n-1}{2} \rceil) + \mathcal{O}(1), && \text{(pela recorrência)} \\ T_M(n) &\leq T_M(\lfloor \tfrac{n+1}{2} \rfloor) + T_M(\lceil \tfrac{n-1}{2} \rceil) + c', && \text{(trocando o } \mathcal{O}(1) \text{ por } c') \\ T_M(n) &\leq c\lfloor \tfrac{n+1}{2} \rfloor - d + c\lceil \tfrac{n-1}{2} \rceil - d + c', && \text{(pela HI)} \\ T_M(n) &\leq c(\lfloor \tfrac{n+1}{2} \rfloor + \lceil \tfrac{n-1}{2} \rceil) - 2d + c', \\ T_M(n) &\leq cn - 2d + c', && \text{(já que } \left\lfloor \frac{n+1}{2} \right\rfloor + \left\lceil \frac{n-1}{2} \right\rceil = n) \\ T_M(n) &\leq cn - d - d + c' \leq cn - d. && \text{(assumindo } d > c') \end{aligned}$$

Portanto, $T_M(n) \leq cn - d$ e $T_M(n) \in \mathcal{O}(n)$.

Falta provar que $T_M(n) \in \Omega(n)$.

Aula 13

Dividir e conquistar

Teorema 4 (Master Theorem)**.** *Seja* $T(n) = aT\frac{n}{b} + f(n)$*, onde* $a \geq 1$*,* $b > 1$ *e* $f(n)$ *é uma função assintoticamente positiva.*

- *Se* $f(n) \in \mathcal{O}(n^c)$*, onde* $c < \log_b a$*, então* $T(n) \in \Theta(n^{\log_b a})$*;*
- *Se* $f(n) \in \Theta(n^{\log_b a} \log^k n)$ *com* $k \geq 0$*, então* $T(n) \in \Theta(n^{\log_b a} \log^{k+1} n)$*;*
- *Se* $f(n) \in \Omega(n^c)$*, onde* $c > \log_b a$ *e* $f(n)$ *satisfaz a condição de regularidade, ou seja,* $af(n/b) \leq kf(n)$ *para alguma constante* $k < 1$ *e* n *suficientemente grande, então* $T(n) \in \Theta(f(n))$*.*

Aula 14

Revisão

Aula 15

1^a prova

Aula 16

Correção da 1^a prova

Aula 17

Entrega de provas

Entregar as provas corrigidas e fazer uma introdução ao tema da aula seguinte.

Aula 18

Técnicas de algoritmos

Explicação sobre espaço de busca.

Busca exaustiva, backtracking, branch & bound, dividir e conquistar, guloso, programação dinâmica, ...

Enumeração, ...

Aula 19

Técnicas de algoritmos - a árvore de busca

Backtracking, branch & bound, guloso.
Exemplos:

- encontrar subconjunto mínimo (ou máximo) com propriedade P;
- clique máxima.

Aula 20

Técnicas de algoritmos - Backtracking e Guloso

20.1 Problema da Mochila (Knapsack)

Seja o seguinte problema: Um ladrão tem uma mochila com capacidade W (peso) e entra em um lugar para roubar. Neste lugar existem n ítens distintos, cada um com um peso, w_i, e um valor, v_i, com $1 \leq i \leq n$. O ladrão quer encher a sua mochila de forma que a soma dos valores seja máxima. Ou seja,

$$\sum_{i=1}^{n} v_i x_i \text{ , sujeito a } \sum_{i=1}^{n} w_i x_i \leq W \text{ onde } x_i \geq 0. \tag{20.1}$$

As variáveis x_i representam a quantidade de cada ítem que o ladrão colocou na mochila.

O Problema da Mochila tem várias versões. Vamos falar da versão 0/1, que é a versão onde só existe uma unidade de cada ítem. Assim, podemos usar um conjunto de variáveis $x_i \in \{0, 1\}$. A equação a ser maximizada pode ser escrita como:

$$\sum_{i=1}^{n} v_i x_i \text{ , sujeito a } \sum_{i=1}^{n} w_i x_i \leq W \text{ onde } x_i \in \{0, 1\}. \tag{20.2}$$

20.1.1 Como resolver?

- Busca exaustiva - Enumeração

- Backtracking
- Guloso? Aproximação?
- Programação Dinâmica

20.1.2 A recorrência que gera a árvore

Seja $M(I, W)$ o valor da mochila ótima para o conjunto de ítens I e capacidade W.

$$M(I,W) = \begin{cases} 0, & \text{se } W = 0 \text{ ou } I = \emptyset \\ M(I \setminus \{i\}, W), & \text{se } w_i > W \\ \max\{M(I \setminus \{i\}, W), M(I \setminus \{i\}, W - w_i) + v_i\}, & \text{c.c.} \end{cases} \tag{20.3}$$

onde $i \in I$ é um ítem qualquer.

Se consireramos os ítens com índices de 1 a n, e escolhemos o i do fim para o começo, o conjunto $I \setminus \{i\}$ pode ser descrito sempre pelo índice do último ítem.

$$M(n,W) = \begin{cases} 0, & \text{se } W = 0 \text{ ou } n = 0 \\ M(n-1, W), & \text{se } w_n > W \\ \max\{M(n-1, W), M(n-1, W - w_n) + v_n\}, & \text{c.c.} \end{cases} \tag{20.4}$$

Qual o tamanho da árvore? No pior caso é uma árvore binária completa de altura n, portanto tem tamanho $2^n - 1$.

20.1.3 Algoritmo de Aproximação Guloso

George Dantzig (1957): ordem não-crescente de valor por unidade de peso (v_i/w_i).

Aula 21

Técnicas de algoritmos - Programação Dinâmica

> *(*) There is an ordering on the subproblems, and a relation that shows how to solve a subproblem given the answers to "smaller" subproblems, that is, subproblems that appear earlier in the ordering.* ([p. 171] Dasgupta, Papadimitriou, and Vazirani. 2006. Algorithms (1 ed.). McGraw-Hill, Inc., New York, NY, USA.)

Caso exemplo: Problema da Mochila.

Subproblemas:

- caso ilimitado - submochilas, com capacidades menores
- caso limitado 0/1 - submochilas e ítens utilizados

21.1 Caso Ilimitado

$$M(w) = \text{maior valor para uma mochila com capacidade } w. \tag{21.1}$$

Se $M(w)$ é ótimo, e na mochila que deu o ótimo tem um ítem i, então $M(w - w_i)$ tem seu ótimo na mochila de $M(w)$ tirando o ítem i, ou seja, $M(w - w_i) = M(w) - v_i$. Variando para cada i, temos:

$$M(w) = \begin{cases} 0, & \text{se } w = 0 \\ \max_{i:w_i \leq w}\{M(w - w_i) + v_i\}, & \text{c.c.} \end{cases} \tag{21.2}$$

Note que os valores que w pode assumir são todos os valores menores ou iguas a W (dependendo dos pesos de cada ítem pode ser menos que isso). Se guardamos os valores de $M(w)$ em um vetor indexado por $[0..W]$ e vamos calculando de 1 até W, temo custo $\mathcal{O}(n)$ para cada posição, e portanto, custo total $\mathcal{O}(nW)$.

21.2 Caso Limitado 0/1

$$M(w, j) = \text{maior valor para uma mochila de capacidade } w \text{ e itens } 1, \ldots, j. \tag{21.3}$$

Queremos então encontrar $M(W, n)$.

$$M(w,j) = \begin{cases} 0, & \text{se } w = 0 \text{ ou } j = 0 \\ \max\{M(w - w_j, j-1) + v_j, M(w, j-1)\}, & \text{c.c.} \end{cases} \tag{21.4}$$

Note que $M(w - w_j, j - 1) + v_j$ representa o caso onde o ítem j está na mochila de $M(w, j)$ e que $M(w, j - 1)$ representa o caso contrário.

É necessário guardar os valores $M(w, j)$ em uma matriz indexada por $[0..W] \times [0..n]$. O custo para preencher cada posição da matriz é $\mathcal{O}(1)$ e o custo total é $\mathcal{O}(nW)$.

Aula 22

Programação dinâmica

escrever as descrições dos problemas.

22.1 Problemas

Multiplicação de matrizes.

Moedas com quaisquer valores.

22.2 Links

1. https://www.topcoder.com/community/data-science/data-science-tutorials/dynamic-programming-from-novice-to-advanced/
2. https://www.cs.berkeley.edu/~vazirani/algorithms/chap6.pdf

Aula 23

Programação dinâmica

escrever as descrições dos problemas.

23.1 Problemas

Distância de edição.

23.2 Links

1. https://www.topcoder.com/community/data-science/data-science-tutorials/dynamic-programming-from-novice-to-advanced/
2. https://www.cs.berkeley.edu/~vazirani/algorithms/chap6.pdf

Aula 24

Branch & Bound

escrever as descrições dos problemas.

Mesmo tipo de recorrências que PD e backtracking.

Corte baseado em limitantes (*bounds*).

Seja um problema de otimização P e uma instância I. Denote a solução ótima de P para a instância I seja $OPT_P(I)$. E seja B uma função limitante para P.

Se P é um problema de **maximização** então $B(I) \geq OPT_P(I)$.

Se P é um problema de **minimização** então $B(I) \leq OPT_P(I)$.

A recorrência gera uma árvore. Cada nó, v, da árvore é um sub-problema, ou seja, tem uma instância I_v associada. A solução final pode usar a solução de um nó com alguma correção. Seja f a função de correção. Um nó v é cortado se o limitante da solução daquele nó (e de seus filhos) não melhorar a melhor solução encotrada até agora.

24.1 Problemas

Clique máxima

24.2 Links

1. http://acervodigital.ufpr.br/bitstream/handle/1884/27588/R%20-%20D%20-%20ZUGE,%20ALEXANDRE%20PRUSCH.PDF

2. http://pubsonline.informs.org/doi/pdf/10.1287/ited.1070.0005

Aula 25

Branch & Bound

escrever as descrições dos problemas.

Mochila

Aula 26

Corretude de Algoritmos

É preciso verificar se um certo algoritmo A de fato resolve o problema P que se pretende resolver.

- Ajuda a descobrir erros
- Melhora a capacidade de desenvolver algoritmos
- Melhora a clareza dos algoritmos desenvolvidos

Como provar a corretude?

- Verificação formal (Djikstra)
- Invariantes de laço
- Indução (recursivos)

Referências:

- Edsger Wybe Dijkstra. 1997. A Discipline of Programming (1st ed.). Prentice Hall PTR, Upper Saddle River, NJ, USA.
- http://www.cs.utexas.edu/~EWD/ewd02xx/EWD249.PDF
- http://cs.engr.uky.edu/~lewis/essays/algorithms/correct/correct1.html

Aula 27

Corretude de Algoritmos

E se o algoritmo é de um formato conhecido?

backtracking: prove que a recorrência resolve o problema.

dividir e conquistar: prove que a recorrência resolve o problema e que os mecanismos de dividir e juntar as partes funcionam.

guloso: prove que a recorrência resolve o problema e que a função que decide o ramo a seguir está correta.

programação dinâmica: prove que a recorrência resolve o problema e que a ordem em que os subproblemas são resolvidos está certa.

branch & bound: prove que a recorrência resolve o problema e que os limitantes estão corretos.