Algoritmos e Estrutura de Dados I

Michel Pires da Silva
michel@cefetmg.br

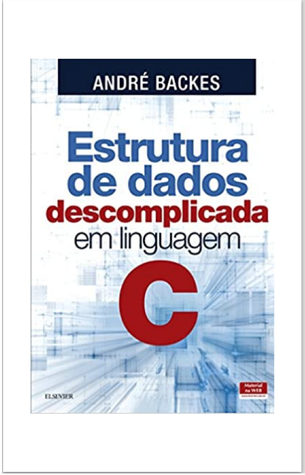
Departamento de Computação
DECOM-DV

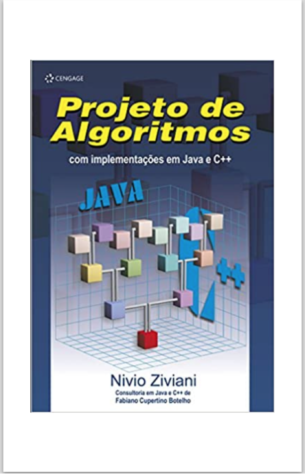
Centro Federal de Educação Tecnológica de Minas Gerais
CEFET-MG

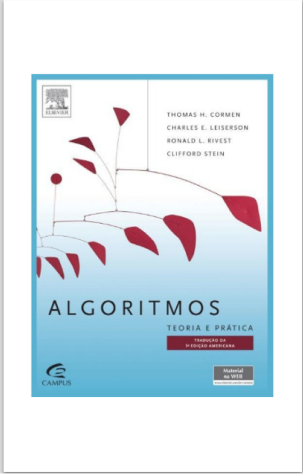
Sumário

Referências

Livros utilizados como base para a preparação das aulas.







Ídolos . . . Até agora :)

Primeiros Conceitos

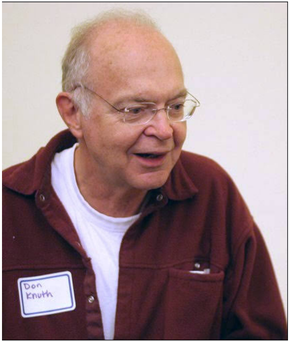
Adu Ja'Far Mohammed Ibn Musa al-khowarizmi (780-850), astrônomo e matemático árebe da cidade de Khowarizmi. Escreveu livros de matemática, astronomia e geografia. A álgebra foi introduzida na Europa ocidental por meio de seus trabalhos. A palavra algoritmo surgiu de uma adaptação latina que autores europeus fizeram com o seu nome.

Primeiros conceitos

Edsger Wybe Dijkstra foi um cientista da computação holandês conhecido por suas contribuições nas mais diversas areas de desenvolvimento de algoritmos. Recebeu o Prêmio Turing em 1972.

Segundo Dijkstra, um algoritmo corresponde a uma **descrição de um padrão** de comportamento, expresso em termos de um **conjunto finito de ações**.

Primeiros conceitos



Donald Knuth, cientista da computação americano, autor do conjunto de obras *The Art of Computer Programming* e criador da ferramenta de edição $\LaTeX\,2_\varepsilon$. Também ganhador do prêmio Turing[a].

Segundo Knuth, as ferramentas de edição de texto são antiquadas, o que torna a produção de grandes obras onerosa.

[a] https://amturing.acm.org/

Atenção: Criem uma conta no overleaf (`https://https://pt.overleaf.com/`), os trabalhos deverão ser realizados utilizando esse editor, de preferência.

Primeiros Conceitos

Estrutura de dados e algoritmos estão intimamente relacionados

- Não se estuda estrutura de dados sem considerar os algoritmos que estão relacionados a cada estrutura

Dica ...

Ao escolher um algoritmo para resolver um determinado problema, observe se a estrutura de dados associada ao mesmo é adequada para solucionar o problema com eficiência.

Cada estrutura de dados contém um conjunto específico de operações. Logo, a escolha da melhor estrutura está diretamente relacionada ao tipo de operação que você está pretendendo executar.

- Podemos resumir a arte de programar em: Escolha de uma estrutura de dados eficiênte e na construção do algoritmo para executar as operações relacionadas a essa estrutura.

Vejamos um exemplo

100 milhões
(entrada)

Nome	Idade	Altura	Peso
Pedro	15	1.78	47
Ana	21	1.69	58
Artur	19	1.82	78
⋮	⋮	⋮	⋮
Amanda	35	1.59	62

Observe a figura acima, note que temos algumas entradas de dados por linha (i.e. uma tupla).

- A partir desse modelo de referência, como você pensaria em representar os dados sob um vetor de entrada.

Consideração: Por questões didáticas, leve em conta que nosso vetor, se bem pensado, caberá completamente em memória primária.

Tipos de Dados

Podemos dividir os tipos de dados em duas classes distintas:

- ***Tipos simples de dados:*** São grupos de valores individuais, tais como, *int*, *boolean*, *char* e *float*, encontrados no C.
- ***Tipos estruturados de dados:*** Comumente, definido como uma coleção de valores simples ou um agregado de valores de tipos distintos. Um exemplo seria o tipo *struct* do C.

Tipo	Bytes	Faixa / Intervalo
short int signed int signed short int	2	-32.768 a 32.767
unsigned short int	2	0 a 65.535
int long int signed long int	4	-2.147.483.648 a 2.147.483.647
unsigned int unsigned long int	4	0 a 4.294.967.295
char signed char	1	-128 a 127
unsigned char	1	0 a 255
bool	1	true ou false
float	4	$1.2e^{-38}$ a $3.4e^{+38}$
double	8	$2.2e^{-308}$ a $1.8e^{+308}$

Tipo Abstrato de Dados

Em computação podemos utilizar um dos tipos apresentados para criar o conceito de Tipo Abstrato de Dados ou TAD's

- Modelo matemático acompanhado das operações definidas para ele.
- Um exemplo seria o conjunto dos inteiros acompanhados pelas operações de adição, subtração, multiplicação e divisão.

Observação . . .

TAD's são utilizados extensivamente como base para a construção de algoritmos

Tipo Abstrato de Dados

Situação: Considere uma aplicação que utilize uma lista de inteiros. Poderíamos definir o TAD *Lista* com as seguintes operações:

- Faça a *Lista* vazia
- Obtenha o primeiro elemento da *Lista*. Se estiver vazia, então, retorne *nill ou null*.
- Insira um elemento na *Lista*

Cada operação do tipo abstrato de dados é implementada como um procedimento na linguagem de programação escolhida.

Observações ...

- Qualquer alteração na implementação do TAD fica restrita a parte **encapsulada** sem causar impactos a outras partes do código;
- Cada **conjunto diferente de operações** define um **TAD diferente**, mesmo que ambos atuem sob um **mesmo modelo matemático**

Tipo Abstrato de Dados

Considerando a linguagem C, o TAD pode ser definido através de uma diretriz chamada *struct*.

Vamos relembrar: Elabore uma solução baseada em uma TAD capaz de manipular 20 elementos do tipo tupla (Nome, Idade e E-Mail). Nessa solução precisamos de funções para inserção, remoção e pesquisa. As inserções são em modo sequencial a partir do primeiro espaço considerado livre (i.e., espaços demarcados com -1). As remoções indexadas, onde o usuário fornece um índice e o valor -1 é atribuido ao espaço. Por fim, as pesquisas são realizadas em modo sequencial a partir da primeira entrada válida.

Observação

- Se alterarmos nossa TAD de inteiros para ponto flutuante haveria custo para as operações realizadas?
- Quanto custa a inserção, remoção e presquisa de nossa solução?

Análise Assintótica

Análise Assintótica: Grosseiramente, trata-se de um processo que observa a capacidade / computabilidade dos algoritmos, definindo para estes um custo de execução.

Desafio ...

Sabemos que, dado uma entrada com K números de uma distribuição sequencial (por exemplo, $\langle 1, \ldots, 10 \rangle$), é possível encontrar algoritmos que os ordene tanto a um custo de n^2 quando de n.

Desafio: Dado uma entrada de números randomicos, não sequenciais, que não se repetem é possível pesquisar um elemento a um custo próximo ou menor que n?

Análise Assintótica

O projeto de algoritmos é influenciado diretamente pelo comportamento das computações envolvidas na solução de um problema.

Medidas importantes que observamos:

- **Tempo:** Utilizado para identificar o custo computacional do algoritmo em relação ao seu tempo de execução
- **Espaço:** Utilizado para identificar o custo computacional com relação a carga que o algoritmo impõe junto aos recursos do sistema, tais como, memória.

Observação ...

Podemos utilizar a **análise assintótica** para avaliar *um algoritmo em particular* ou *uma classe de algoritmos*

Análise Assintótica

A *análise de um algoritmo em particular* visa encontrar:

- qual o custo de utilizar um dado algoritmo para resolver um problema específico?
- Quais características devem ser investigadas?

Avaliações

- Avaliação do número de vezes que cada parte do algoritmo será executada
- Estudo da quantidade de memória necessária durante sua execução

Análise Assintótica

A *análise de uma classe de algoritmos* visa encontrar:

- Qual é o algoritmo de menor custo possível para resolver um problema em particular?

Para responder a pergunta ...

- Toda uma família de algoritmos é investigada
- É identificado o algoritmo que apresenta o melhor desempenho (i.e. tempo / espaço) para o conjunto de dados utilizado
- É colocado limites para a complexidade computacional dos algoritmos pertencentes a classe avaliada.

Análise Assintótica

Sobre o custo dos algoritmos:

- **Determinar o menor custo** possível para resolver problemas de uma dada classe temos a **medida da dificuldade** inerente para resolver o problema
- Quando o custo de um algoritmo é igual ao **menor custo possível** para o conjunto de dados avaliado, o algoritmo é considerado **ótimo**
- Se a mesma medida de custo é aplicada em diferentes algoritmos então é possível compara-los e escolher o mais adequado.

Não podemos obter custo de máquinas reais ...

- Os resultados ficam dependentes do compilador. Um exemplo seria os compiladores que executam com eficiência *loop unrolling*
- Os resultados dependeriam muito do *hardware* e/ou modelo de execução
- Quando grandes quantidades de memória forem utilizadas, medir o tempo depende do aspecto espacial

Análise Assintótica

Solução para a avaliação: Elaboração de modelos matemáticos sob uma **máquina hipotética**

- Nesse modelo é preciso especificar um conjunto de operações e seus custos de execução
- É viável desconsiderar algumas operações e levar em conta somente as mais significantes
  - Um exemplo está nos algoritmos de ordenação. Nesses, é possível levar em consideração o número de comparações como métrica de avaliação, desconsiderando as demais ações realizadas.

Nesse formato geramos o que chamamos de *função de complexidade*

Análise Assintótica

Função de complexidade: Tem por objetivo medir o custo de execução de um algoritmo. Esta função usa como representação a letra f

- $f(n)$ é a medida do tempo necessário para executar um algoritmo sob um problema de tamanho n

Observações ...

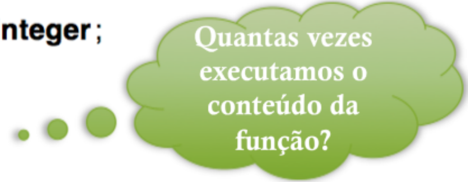
- A função de *complexidade de tempo* $f(n)$ mede o tempo necessário para executar um algoritmo. Não representa necessariamente tempo, mas sim, o número de vezes que uma operação relevante é executa.
- A função de *complexidade de espaço* $f(n)$ mede a memória necessária para executar um dado algoritmo

Análise Assintótica

Vejamos um exemplo

- Considere o algoritmo abaixo que tenta encontrar o maior elemento de um vetor de inteiros $A[1 \ldots n]$, $n \geq 1$

```
function Max (var A: Vetor): integer;
var i, Temp: integer;
begin
  Temp := A[1];
  for i := 2 to n do if Temp < A[i] then Temp := A[i];
  Max := Temp;
end;
```



Análise Assintótica

Solução: Seja f uma função de complexidade tal que $f(n)$ é o número de comparações entre os elementos no conjunto A. Logo, se A contiver n elementos:

- $f(n) = (n - 1)$, para $n > 0$

Pergunta?

Esse algoritmo pode ser considerado **ótimo** ?

Análise Assintótica

Observação

Observe que a medida de custo está diretamente ligada ao tamanho do conjunto de dados

- É comum considerar o tempo de execução de um programa como uma função do tamanho da entrada
- O custo computacional também é afetado, em muitos casos, pela disposição dos dados no conjunto
- No exemplo anterior a função **Max** tem custo uniforme sobre todos os problemas de tamanho n.

Pergunta ?

É possivel encontrar o maior elemento com custo menor do que foi encontrado ?

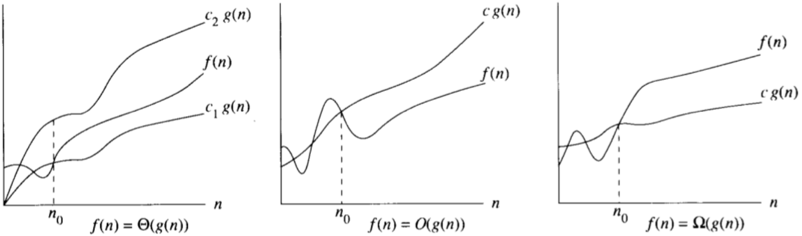
Análise Assintótica

Possíveis avaliações aplicáveis aos algoritmos

- **Melhor caso** $O(g(n))$**:** Menor tempo de execução sob **todas as entradas** de tamanho *n*
- **Pior caso** $\Omega(g(n))$**:** Maior tempo de execução sob **todas as entradas** de tamnho *n*
- **Caso médio (ou caso esperado)** $\theta(g(n))$**:** Média dos tempos de execução de **todas as entradas** de tamanho *n*. Aqui é comum o uso de uma *distribuição de probabilidade*

Análise Assintótica

Comportamento das notações O, Ω e Θ



Notações

- $O(g(n))$ = {Há constantes positivas c e n_0 tal que $0 \leq f(n) \leq cg(n)$ para todo $n \geq n_0$ }
- $\Omega(g(n))$ = {Há constantes positivas c e n_0 tal que $0 \leq cg(n) \leq f(n)$ para todo $n \geq n_0$}
- $\Theta(g(n))$ = {Há constantes positivas c_1, c_2 e n_0 tal que $c_1 g(n) \leq f(n) \leq c_2 g(n)$ para todo $n \geq n_0$}

Análise Assintótica - Classes de Problemas

- $f(n) = O(1)$: Esses algoritmos são ditos de **complexidade constante**
  - Utilização do algoritmo independente do tamanho de n
  - As instruções do algoritmo são executadas um número fixo de vezes

- $f(n) = O(logn)$: Esses algoritmos são ditos de **complexidade logarítmica**
  - Típico em algoritmos que transformam o problema em outros menores
  - Pode considerar o tempo de execução como menor do que uma constante grande
  - Quando n é mil, $log_2 n = 10$, quando n é 1 milhão, $log_2 n = 20$. Assim para dobrar o valor do *logn* precisamos considerar o quadrado de n

Análise Assintótica - Classes de Problemas

- $f(n) = O(n)$: Esses algoritmos são ditos de **complexidade linear**
  - Em geral, apenas um pequeno trabalho é realizado sob cada elemento da entrada
  - É a melhor situação para um algoritmo que tenha que processar/produzir n elementos de entrada/saída
- $f(n) = O(nlogn)$: Esses custo é típico em algoritmos que quebram o problema em outros menores, resolvem cada um deles independentemente e reunem as soluções depois
  - Quando n é 1 milhão, $nlog_2n$ é cerca de 20 milhões
  - Quando n é 2 milhões $nlog_2n$ é cerca de 42 milhões, pouco mais que o dobro

Análise Assintótica - Classes de Problemas

- $f(n) = O(n^2)$: Esses algoritmos são ditos de **complexidade quadrática**
  - Ocorrem quando os itens de dados são processados aos pares, muitas vezes em um laço dentro do outro
  - Sempre que n dobra, o tempo de execução é multiplicado por 4
  - Algoritmos úteis para resolver problemas de tamanhos relativamente pequenos

- $f(n) = O(n^3)$: Esses algoritmos são ditos de **complexidade cúbica**
  - Úteis apenas para resolver pequenos problemas
  - Quando n é 100, o número de operações é da ordem e 1 milhão
  - Sempre que n dobra, o tempo de execução fica multiplicado por 8

Análise Assintótica - Classes de Problemas

- $O(2^n)$: Esses algoritmos são ditos de **complexidade exponencial**
  - Geralmente não são úteis sob o ponto de vista prático
  - Ocorrem na solução de problemas quando se usa **força bruta** para resolvê-los
  - Quando n é 20, o tempo de execução é certa de 1 milhão. Quando n dobra, o tempo fica elevado ao quadrado
- $f(n) = O(n!)$: Esses algoritmos são ditos de **complexidade exponencial** apesar de $O(n!)$ ter comportamento muito pior do que $O(2^n)$
  - Ocorrem quando se usa **força bruta** na solução do problema
  - Quando n é 20 tem-se 2432902008176640000, um número de 19 dígitos

Análise Assintótica

Um exemplo: Considere que precisamos acessar um dado **registro** em um conjunto de dados distribuído de forma aleatória.

- Cada registro contém uma **chave** única que é utilizada para recuperar registros do arquivo

Pergunta ?

Qual o custo computacional para se executar tal tarefa no melhor, pior e caso médio ?

Análise Assintótica

Possível solução: Algoritmo mais simples é o que faz a **pesquisa sequencial**

- Seja f uma função de complexidade tal que $f(n)$ é o número de registros consultados no arquivo. Logo:
  - **Melhor caso:** $f(n) = 1 \rightarrow O(1)$;
  - **Pior caso:** $f(n) = n + 1 \rightarrow \Omega(n) = n + 1$ para representar registro não encontrado
  - **Caso médio:** $f(n) = \frac{(n+1)}{2} \rightarrow \Theta(n) = \frac{(n+1)}{2}$

Análise Assintótica

Caso 1: Considere a situação de encontrar o maior e o menor elemento de um vetor de inteiros $A[1 \dots n]$, $n \geq 1$

```
procedure MaxMin1 (var A: Vetor; var Max, Min: integer);
var i : integer;
begin
  Max := A[1];  Min := A[1];
  for i := 2 to n do
    begin
    if A[i] > Max then Max := A[i];
    if A[i] < Min then Min := A[i];
    end;
end;
```

Análise Assintótica

Seja $f(n)$ o número de comparações entre os elementos de A para a obtenção do maior e menor elementos, se A possui n elementos então:

Solução ...

- $f(n) = 2(n-1)$, para $n > 0$ no melhor caso, pior caso e caso médio.

O resultado de $f(n)$, nesse caso, está diretamente relacionado a quantas vezes cada *if* é executado dentro do laço de repetição

É possível fazer o algoritmo melhorar sua performance e com isso reduzir sua complexidade ?

Análise Assintótica

Caso 2: Vejamos agora se o algoritmo melhorou sua performance

```
procedure MaxMin2 (var A: Vetor; var Max, Min: integer);
var i: integer;
begin
  Max := A[1];  Min := A[1];
  for i := 2 to n do
     if A[i] > Max
     then Max := A[i]
     else if A[i] < Min then Min := A[i];
end;
```

Análise Assintótica

Avaliando o caso 2 nós temos os seguintes custos computacionais:

- **Melhor caso** $O(n)$**:** Ocorre quando todos os elementos estão ordenados de forma crescente. Isso fará com que somente o *if* seja avaliado gerando um custo de $f(n) = (n-1)$
- **Pior caso** $\Omega(n)$**:** Ocorre quando os elementos estão ordenados de forma decrescente. Isso faz com que o *else* seja avaliado constantemente após o *if* gerando um custo de $f(n) = 2(n-1)$
- **Caso médio:** Considera-se que o conjunto de dados está disposto de tal forma que haverá (n - 1) passadas no *if* e $\frac{(n-1)}{2}$ no *else*. Logo, $f(n) = (n-1) + \frac{(n-1)}{2} = \frac{3n}{2} - \frac{3}{2}$

Análise Assintótica

Tarefa

- Como você melhoraria o **MinMax2** e qual seria o custo assintótico que iria obter com as modificações, se isso é possível.
- Tente pensar em um algoritmo que consiga apresentar melhores resultados que o **MinMax2** e discuta quais pontos foram relevantes para a construção de tal solução. Ou prove, se possível, que **MinMax2** é o caso ótimo para resolver esse problema computacional.

Análise Assintótica

Caso 3: Uma possível solução de melhoria para o **MinMax2**

```
procedure MaxMin3 (var A: Vetor; var Max, Min: integer);
    var i, FimDoAnel: integer;
begin
    if (n mod 2) > 0 then begin A[n+1] := A[n];   FimDoAnel := n; end
    else  FimDoAnel := n – 1;

    if ( A[1] > A[2] ) then begin Max := A[1]; Min := A[2]; end
    else begin Max := A[2]; Min := A[1]; end;

    i:=3;
    while ( i <= FimDoAnel ) do begin
        if (A[i] > A[i+1]) then begin
            if (A[i] > Max) then Max := A[i];
            if ( A[i+1] < Min ) then Min := A[i+1];
        end
        else begin
            if ( A[i+1] > Max ) then Max := A[i+1];
            if A[i] < Min then Min := A[i];
        end;
        i := i + 2;
    end;
end;
```

Qual a complexidade desse algoritmo?

Análise Assintótica

Avaliando o caso 3 nós temos os seguintes custos computacionais:

- Os elementos contídos em A são avaliados aos pares, logo tem-se um custo total de avaliação de $\frac{n}{2}$
- O maior elemento é obtido pela primeira parte do laço a um custo de $\frac{n}{2} - 1$ comparações
- O menor elemento é obtido pela segunda parte do algoritmo a um custo de $\frac{n}{2} - 1$ comparações

Para essa implementação o custo total é $f(n) = \frac{n}{2} + \frac{(n-2)}{2} + \frac{(n-2)}{2} = \frac{3n}{2} - 2$, para $n > 0$. Isso no melhor, pior e caso médio

Análise Assintótica

A seguir tem-se o quadro comparativo dentre os algoritmos apesentados

Os três algoritmos	$f(n)$		
	Melhor caso	Pior caso	Caso médio
MaxMin1	$2(n-1)$	$2(n-1)$	$2(n-1)$
MaxMin2	$n-1$	$2(n-1)$	$3n/2-3/2$
MaxMin3	$3n/2-2$	$3n/2-2$	$3n/2-2$

Análise Assintótica

Existe a possibilidade de melhorar ainda mais o algoritmo ?

Como sabemos quando é hora de parar de tentar ?

- **Técnica:** Utilização de um *oráculo*
- Dado um modelo de computação que expresse o comportamento do algoritmo, o oráculo informa o resultado de cada passo possível (no caso, o resultado de cada comparação)
- O oráculo preocupa-se em fazer com que o algoritmo trabalhe o máximo possível, escolhendo como resultado da próxima comparação aquele passo que gere maior volume de trabalho

Análise Assintótica

Teorema

Qualquer algoritmo que encontre o maior e o menor elemento de um conjunto com n elementos não ordenados para $n > 0$ faz pelo menos $\frac{3n}{2} - 2$ comparações.

- **Prova:** A técnica descreve o comportamento do algoritmo por meio de um conjunto (A, B, C, D) para demostrar os possíveis estados da execução
  - **A:** Representa os elementos que nunca foram comparados
  - **B:** Representa os elementos que nunca perderam em comparações realizadas
  - **C:** Representa os elementos que foram perdedores e nunca venceram em comparações realizadas
  - **D:** Representa os elementos que foram vencedores e perdedores em comparações realizadas

Análise Assintótica

O estado inicial do algoritmo será $(n, 0, 0, 0)$ e deve terminar em $(0, 1, 1, n-2)$

Os possíveis estados que o oráculo pode assumir é:

- Dois elementos de A são comparados $(A-2, B+1, C+1, D)$
- Um elemento de A é comparado com B ou C, logo tem-se $(A-1, B+1, C, D)$, $(A-1, B, C+1, D)$ ou $(A-1, B, C, D+1)$
- Dois elementos de B, para $B \geq 2$ são comparados, logo tem-se $(A, B-1, C, D+1)$
- Dois elementos de C, para $C \geq 2$ são comparados, logo tem-se $(A, B, C-1, D+1)$

Análise Assintótica

O caminho mais rápido para levar A até zero requer $\frac{n}{2}$ mudanças de estado, terminando como $(0, \frac{n}{2}, \frac{n}{2}, 0)$

- Para reduzir B a um único elemento são necessários $\frac{n}{2} - 1$ comparações
- Para reduzir C a um único elemento são necessários $\frac{n}{2} - 1$ comparações
- Logo, para obter o estado $(0, 1, 1, n-2)$ a partir do estado $(n, 0, 0, 0)$ são necessários $\frac{n}{2} + \frac{(n-2)}{2} + \frac{(n-2)}{2} = \frac{3n}{2} - 2$ comparações. Assim, o algoritmo MaxMin3 é **ótimo**

Vamos Praticar . . .

1) - Mostre uma função em C/C++ que consiga trabalhar com a sequência abaixo[1]:

$$1, -\frac{1}{3}, \frac{1}{5}, -\frac{1}{7}, \frac{1}{9}, -\frac{1}{11}, \ldots$$

2) - Mostre se as correlações abaixo são verdadeiras:

- $3n^3 + 2n^2 + n + 1 = \mathcal{O}(n^3)$
- $7n^2 = \mathcal{O}(n)$
- $2^{n+2} = \mathcal{O}(2^n)$
- $2^{2n} = \mathcal{O}(2^n)$
- $5n^2 + 7n = \Theta(n^2)$
- $6n^3 + 5n^2 = \Omega(n^2)$
- $9n^3 + 3n = \Omega(n)$
- $6n^3 \neq \Theta(n^2)$

[1] Uma analogia: População celular e bactérias

Mais sobre limites

Além das notações $\mathcal{O}$, Ω temos em literatura definições para o, ω.

Observação

Notação pouco utilizada mas contemplada em literatura, sendo importânte saber seus conceitos e diferenças em relação às notações $\mathcal{O}$, Ω.

As notações o, ω correspondem a casos mais específicos de representação de custo, ou seja, são definições mais firmes.

O pequeno *o*

- O limite assintótico superior definido pela notação $\mathcal{O}$ pode ser assintoticamente firme ou não, ou seja, a condição para uma função $f(n)$ pertença a $g(n)$ é dado por comparações $\geq$ e $\leq$

- **Exemplo**: O limite $2n^2 = \mathcal{O}(n^2)$ é assintóticamente firme, mas o limite $2n = \mathcal{O}(n^2)$ não é.

- Formalmente, a notação *o* é definida como:

$f(n) = o(g(n))$, para qualquer $c > 0$ e $n_0 \mid 0 \leq f(n) < cg(n) \ \forall \ n \geq n_0$

- **Exemplo**: $2n = o(n^2)$ mas $2n^2 \neq o(n^2)$

Em *o*, a função $f(n)$ apresenta crescimento muito menor que $g(n)$

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

Notação ω

- Por analogia, a notação ω está para Ω tal como o está para $\mathcal{O}$.

- Formalmente, ω é definido como:

$f(n) = \omega(g(n))$, para qualquer $c > 0$ e $n_0 \mid 0 \leq cg(n) < f(n)\ \forall$ $n \geq n_0$

- **Exemplo**: $\frac{n^2}{2} = \omega(n)$, mas $\frac{n^2}{2} \neq \omega(n^2)$

- Em ω, a função $f(n)$ apresenta crescimento muito maior que $g(n)$

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

Hierarquia de funções

Ao avaliarmos um conjunto de algoritmios, podemos estabelecer uma hierarquia de tempo e espaço para ordena-los a partir de seus custos assintóticos.

$$1 \prec loglogn \prec logn \prec n^{\varepsilon} \prec n^{c} \prec n^{logn} \prec c^{n} \prec n^{n} \prec c^{c^{n}}$$

Para todo ε e c arbitrários com $0 < \varepsilon < 1 < c$

Funções Recursivas

Seja a seguinte função para calcular o fatorial de *n*.

```
function fat (var n : integer) : integer;
begin
   if ((n == 0) || (n == 1)) then
      fat := 1;
   else
      fat := n * fat(n -1);
end;
```

$$T(n) = \begin{cases} d & n = 1 \\ c + T(n-1) & n > 1 \end{cases}$$

Essa equação diz que, quando $n = 1$ o custo para executar *fat* é igual a d e para valores de n maiores que 1, o custo para executar *fat* é c mais o custo para executar $T(n-1)$.

Resolvendo a equação de recorrência

A equação de recorrência de *fat* pode ser expressa da seguinte forma:

$$
\begin{aligned}
T(n) &= c + T(n-1) \\
&= c + (c + T(n-2)) \\
&= c + c(c + T(n-3)) \\
\vdots &= \vdots \\
&= c + c + \cdots + (c + T(1)) \\
&= \underbrace{c + c + \cdots + c}_{n-1} + d
\end{aligned}
$$

Em cada passo, o valor de T é substituído pela sua definição, utilizando-se para isso um método de expansão. A última equação mostra que depois da expansão existem $n-1$ c's, correspondentes aos valores de 2 a n. Dessa forma, a recorrência pode ser expressa como:

$$T(n) = c(n-1) + d = \mathcal{O}(n)$$

Somatórios Úteis

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$\sum_{i=0}^{k} 2^k = 2^{k+1} - 1$$

$$\sum_{i=0}^{k} a^i = \frac{a^{k+1} - 1}{a - 1} \rightarrow a \neq 1$$

$$\sum_{i=0}^{k} \frac{1}{2^i} = 2 - \frac{1}{2^k}$$

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

Teorema Mestre

Algumas vezes o custo computacional é definido por uma equação de recorrência, tal como:

$$T(n) = aT(\frac{n}{b}) + f(n)$$

onde, $a \geq 1$ e $b > 1$ são constantes e $f(n)$ é uma função assintóticamente positiva que podem ser resolvidas utilizando um modelo matemático chamado de Teorema Mestre. Esse teorema pode ser limitado assintóticamente da seguinte forma:

1. Se $f(n) = O(n^{log_b a - \varepsilon})$ para alguma constante $\varepsilon > 0$, então $T(n) = \Theta(n^{log_b a})$
2. Se $f(n) = \Theta(n^{log_b a})$, então $T(n) = \Theta(n^{log_b a} logn)$
3. Se $f(n) = \Omega(n^{log_b a + \varepsilon})$ para alguma constante $\varepsilon > 0$ e se $af\left(\frac{n}{b}\right) \leq cf(n)$ para alguma constante $c < 1$ e para n suficientemente grande, então $T(n) = \Theta(f(n))$

Teorema Mestre

Comentários:

- Nos três casos estamos comparando $f(n)$ com a função de recorrência no formato $n^{log_b a}$. Logo, a solução da recorrência é dada pela maior das duas funções.

Observações:

- No primeiro caso, a função $n^{log_b a}$ é maior e a solução de $T(n) = \Theta(n^{log_b a})$
- No terceiro caso, a função $f(n)$ é maior e a solução para a recorrência é $T(n) = \Theta(f(n))$
- No segundo caso, as duas funções apresentam o mesmo tamanho. Logo, a solução fica multiplicada por um fator logarítimo na forma $T(n) = \Theta(n^{log_b a} logn) = \Theta(f(n) logn)$

Teorema Mestre

O teorema **não** cobre todas as possibilidades de recorrência para $f(n)$:

- Entre os casos 1 e 3 existem funções $f(n)$ que são menores que $n^{log_b a}$ mas não são polinomialmente menores.
- Entre os casos 2 e 3 existem funções $f(n)$ que são maiores que $n^{log_b a}$ mas não são polinomialmente maiores.

Quando $f(n)$ cai em uma dessas restrições ou a condição de regularidade do caso 3 é falsa, então não se pode aplicar o teorema mestre como solução para a recorrência.

Teorema Mestre

Vamos aplicar o teorema mestre no seguinte exemplo:

$$T(n) = 9T(\frac{n}{3}) + n$$

Definimos as variáveis como:

$$a = 9,\ b = 3 \text{ e } f(n) = n$$

Dessa forma,

$$n^{log_b a} = n^{log_3 9} = \Theta(n^2)$$

Como $f(n) = O(n^{log_3 9 - \varepsilon})$, onde $\varepsilon = 1$, podemos aplicar o caso 1 do teorema e concluir que a solução da recorrência é

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

Teorema Mestre

Vamos praticar um pouco, aplique o teorema mestre nas seguintes equações de recorrência:

- $T(n) = 4T(\frac{n}{2}) + n$
- $T(n) = 4T(\frac{n}{2}) + n^2$
- $T(n) = 4T(\frac{n}{2}) + n^3$
- $T(n) = 2T(\frac{n}{2}) + n$

Teorema Mestre

Desafio:
O tempo de execução de um algoritmo *A* é de:

$$T(n) = 7T(\frac{n}{2}) + n^2$$

Dado um algoritmo A' cuja função de recorrência é dada por:

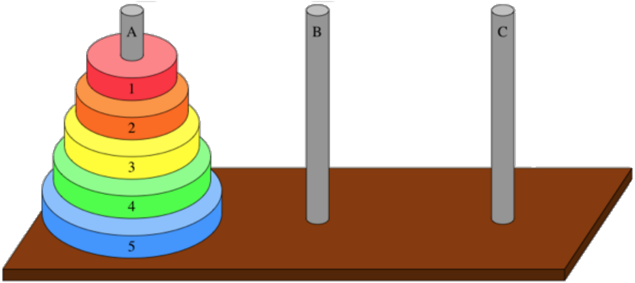
$$T(n) = aT'(\frac{n}{4}) + n^2$$

Pergunda: Qual o maior valor inteiro para *a* que faz de A' assintoticamente mais rápido que A?

Modelagem por recorrência

A Torre de Hanoi foi apresentada em 1883 pelo matemático frances Edouard Lucas.

O jogo é muito simples :), ele começa com um conjunto de discos empilhados em tamanhos decrescente em uma de três varetas.



Modelagem por recorrência

Objetivo: Transferir toda a torre para uma das outras varetas, movendo um disco de cada vez, mas nunca movendo um disco maior sobre um menor.

Solução recorrente:

- Seja $T(n)$ o número mínimo de movimentos para transferir n discos de uma vareta para outra de acordo com as regras definidas;
- Observe que:

  $T(0) = 0$ [nenhum movimento é necessário]
  $T(1) = 1$ [apenas um movimento é necessário]
  $T(2) = 3$ [três movimentos são necessários]
  $T(3) = 7 \ldots$

Modelagem por recorrência

Generalização do problema:

- Para três discos, a solução é transferir os dois discos do topo para a vareta do meio, transferir o terceiro para a última vareta e, finalmente, mover os dois outros discos sobre o topo do terceiro disco.

Para n discos:

1. Transferir os $n - 1$ discos menores para outra vareta requer $T(n-1)$ movimentos;
2. Transferir o maior disco para a última vareta requer 1 movimento;
3. Transferir os $n - 1$ discos menores sob o maior disco requer $T(n-1)$ movimentos;

$$T(0) = 0$$
$$T(n) = 2T(n-1) + 1 \text{ para } n > 0$$
$$T(n) = 2^n - 1 \text{ movimentos}$$

Modelagem por recorrência

Provando que a Torre de Hanoi gasta no mínimo $2^n - 1$ movimentos para executar sua função de movimentação.

Caso base: Para $n = 0$ tempos que $T(0) = 2^0 - 1 = 0$, valor presente na equação detalhada acima

Indução: Considerando nossa recorrência, supomos que a formula fechada é válida para todos os valores até $T(n-1)$, ou seja, $T(n-1) = 2^{n-1} - 1$. Dessa forma, temos:

$$T(n) = 2T(n-1) + 1 = 2(2^{n-1} - 1) + 1 = 2^n - 2 + 1 = 2^n - 1$$