Análise de algoritmos

Prof^a. Rose Yuri Shimizu

Roteiro

1. Medir algoritmos
2. Análise de algoritmos
3. Função custo: valores comuns
4. Complexidade

Como medir algoritmos?

Tempo real de máquina como medida?

```
rysh@mundodalua:~/Documents/FGA$ time ./a.out

real	0m1,301s
user	0m1,297s
sys	0m0,004s
```

```
rysh@mundodalua:~/Documents/FGA$ time ./a.out

real	0m18,300s
user	0m1,431s
sys	0m0,000s
```

Máquina com *load* baixo e alto, respectivamente

- real: tempo total para execução (contando todos os processos em execução)
- user: tempo exclusivo do processo executado
- sys: tempo que do sistema dedicado a execução do processo

- Problema: são dependentes de fatores como a linguagem, hardware e/ou processos em execução
- Precisamos de uma medida independente da máquina

Como medir algoritmos?

Contar quantas instruções são executadas?

- **Análise de algoritmos**: analisar a quantidade de recursos que um algoritmo necessita para resolver um problema
- Considerar somente as operações **relevantes**
- Observando a **tendência do comportamento** a medida que a **instância do problema cresce**
  - Sequência de entrada para um problema computacional resolver (calcular/soluções) ou decidir (encontrar uma solução verdadeira) ou verificar (se uma solução é válida)
- Fazendo uma estimativa dos custos das operações
  - Definindo a **custo dos algoritmos**
  - E a **complexidade assintótica**

Roteiro

1. Medir algoritmos
2. Análise de algoritmos
3. Função custo: valores comuns
4. Complexidade

Análise de algoritmos

Função de Custo *f(n)*

- $f(n)$: valor que mensura os recursos necessários para um algoritmo processar a entrada n
- Conta-se as operações mais relevantes e o custo do processamento da instância do problema
- Entradas possíveis:
  - Problemas em ordenação de vetores: tamanho do vetor
  - Problemas em matrizes: linhas e colunas
  - Problemas de pesquisa em memória: número de registros
  - Problemas de grafos ("objetos/vértices que relacionam entre si"): quantidade vértices
  - etc.

Análise de algoritmos

Cenários (dependentes do tamanho da intância)

- **Melhor caso**: menor custo
- **Caso médio**: média dos custos
- **Pior caso**: maior custo

Exemplo: busca sequencial em vetor

```c

int v* = {23, 22, 98, 49, 21, 5, 3, 456, 16, 83, 50, 97};

```

1. **Melhor caso**: `procura(23, v);`
2. **Pior caso**: `procura(97, v);`
3. **Caso médio**: `procura(21, v);`

Análise de algoritmos

Exemplo: busca sequencial em vetor

- **Melhor caso** $f(n) = 1$: procurado é o primeiro consultado
- **Pior caso** $f(n) = n$: procurado é o último consultado
- **Caso médio** $f(n) = (n+1)/2$: examina cerca de metade dos registros
  - p_i a probabilidade de encontrar o elemento na posição i
  - Todos tem o mesmo $p_i = 1/n$, $1 \leq i \leq n$
  - f(n) = soma do número de comparações x probabilidade

$$\begin{aligned} f(n) &= 1(\frac{1}{n}) + 2(\frac{1}{n}) + \ldots + n(\frac{1}{n}) \\ &= \frac{1}{n}(1 + 2 + \ldots + n) \\ &= \frac{1}{n}(\frac{n(n+1)}{2}) = \frac{(n+1)}{2} \end{aligned}$$

- Custos referem-se, em geral, ao consumo de tempo no pior caso.

Roteiro

1. Medir algoritmos
2. Análise de algoritmos
3. Função custo: valores comuns
4. Complexidade

Constante: $f(n) = c$

- Valor constante: $c > 0$
- Independem do tamanho de n
- As instruções são executadas um número fixo de vezes
  - Atribuições, operações aritmética
  - Comando de decisão, comparações
  - Acessos a memória
- $f(n) \approx 6$:

```
int operacao(int n, int a, char op) {
    int r = 0; //atribuição

    if(op==‘+’) r = n+a; //comparação
    else if(op==‘-’) r = n-a; //comparação
    else if(op==‘*’) r = n*a; //comparação +
                              //aritmética + atribuição

    return r;
}
```

Constante: $f(n) = c$

- Listas estáticas (array):
  - Acesso aleatório e direto pelo índice/posição
  - Inserção
- Listas encadeadas:
  - Remoção de um nó específico em lista duplamente encadeadas
  - Inserção após um nó específico
  - Inserção antes um nó específico em lista duplamente encadeadas

```c
int busca(int n, int *v, int x) {
    int i;
    for(i=0; i<n && v[i]!=x; i++);

    return i;
}
```

Linear: $f(n) = a * n + b$

- Função polinomial de primeiro grau: $a > 0$ e b custo das constantes
- Realiza-se um pequeno trabalho sobre cada elemento da entrada
- n processamentos de custo constante
- Cresce a uma taxa constante
- n entradas, n saídas
- Anel ou laço
  - (Tempos comandos internos + avaliação da condição) x número de iterações

```
int pesquisa(int x, int n, int v[]) {
    for (int i=0; i<n && v[i]!=x; i=i+1);
    return i;
}

int soma (long n) {
    int r = 0;
    while (n > 0) {
        r += n%10;
        n/=10;
    }
}
```

Linear: $f(n) = a * n + b$

- Listas estáticas (array):
  - Buscas/remoções no meio da lista
- Listas encadeadas:
  - Busca sequencial
  - Remoção de um nó específico em listas simplesmente encadeadas
  - Inserção antes de um nó específico em listas simplesmente encadeadas

```
int busca(int n, int *v, int x) {
    int i;
    for(i=0; i<n && v[i]!=x; i++);

    return i;
}

no *buscar(cabeca *lista, Item x) {
    no *a = NULL;
    for(a=lista->prox; a && a->info!=x; a=a->prox);
    return a;
}
```

Linear: $f(n) = a * n + b$

Recorrências:

- Resolver uma recorrência: encontrar uma "fórmula fechada" que calcule diretamente a função
- Sem subexpressões da forma: $+ ... +$ ou contendo $\sum$ ou Π
- Resultando em combinações de:
  - Funções polinomiais: $f(x) = a_n * x^n + a_{n-1} * x^{n-1} + ... + a_1 * x + a_0$
    - n: número inteiro positivo ou nulo
    - x: variável
    - a's: coeficientes
  - Funções exponenciais: $f(x) = b^x$
  - Funções logarítmicas: $f(x) = log_a x$

Linear: $f(n) = a * n + b$

```
//fatorial iterativo
//f(n) ≈ c*n, sendo c uma constante
int fatorial1(int n) {
    int fat = 1;

    while(n > 0) //n..0
        fat *= n--; //constante → n vezes

    return fat;
}

//fatorial recursivo: fórmula fechada?
int fatorial2(int n)
{
    if(n==1 || n==0) return 1;
    return n * fatorial2(n-1);
}
```

Linear: $f(n) = a * n + b$

```
//fatorial recursivo
int fatorial2(int n)
{
    if(n==1 || n==0) return 1; //constante
    return n * fatorial2(n-1); //custo p/ entrada n-1
}
```

$$
\begin{aligned}
f(n) &= c + f(n-1) \\
&= c + c + f(n-2) \\
&= c + c + c + f(n-3) \\
&\ldots \\
&= c * i + f(n-i) \\
&= c * n + f(1), i = n-1 \\
&= c * (n-1) + c = c * n
\end{aligned}
$$

Quadrática: $f(n) = a * n^2 + b * n + c$

- Função polinomial do segundo grau
- Para cada n, processa-se cerca de n itens
- n processamentos de custo linear
- Se n dobra, o tempo quadruplica
- Valor depende de uma constante exponencial fixa (2)
- Caracteriza-se por aninhamento de iterações

```
//versao quadradica != ordenacao seja quadratica (algoritmo
     de quicksort)
void ordenacao(int v[], int n)
{
    for(int i=1; i<n; i++) //a partir do segundo elemento
    {
        for(int j=i; j>0 && v[j]<v[j-1]; j--) //comparar
    com os antecessores
        {
            troca(v[j], v[j-1]); //posicionar
        }
    }
}
```

Quadrática: $f(n) = a * n^2 + b * n + c$

- Versão recursiva

```c
void ordenacao(int v[], int n)
{
    if(n<=1) return;
    ordenacao(v, n-1); //f(n-1)

    for(int j=n-1; j>0 && v[j]<v[j-1]; j--) //n-1
        troca(v[j], v[j-1]);
}
```

- Fórmula fechada para encontrar custo:

Quadrática: $f(n) = a * n^2 + b * n + c$

$$
\begin{aligned}
f(n) &\approx f(n-1) + (n-1) + c \\
&\approx f(n-2) + (n-2) + c + (n-1) + c \\
&\approx f(n-2) + (n-2) + (n-1) + 2 * c \\
&\approx f(n-3) + (n-3) + (n-2) + (n-1) + 3 * c \\
&\approx f(n-i) + (n-i) + ... + (n-2) + (n-1) + i * c : n-i = 1, i = n-1 \\
&... \\
&\approx f(1) + (n-(n-1)) + ... + (n-1) + (n-1) * c \\
&\approx c + 1 + 2 + ... + (n-1) + (n-1) * c \\
&\approx n * c + \frac{(1 + (n-1)) * (n-1)}{2} \\
&\approx n * c + \frac{n^2}{2} + \frac{n}{2} \\
&\approx \frac{n^2}{2} + \frac{n}{2} + n * c
\end{aligned}
$$

Cúbica: $f(n) = a * n^3 + b * n^2 + c * n + d$

```c
//versao cubica != multiplicacao seja cubica (algoritmo de Strassen)
//se n cresce, a entrada multiplica
void multiplica_matrizes(int A[3][3], int B[3][3], int C[3][3]) { //nxn
    for (int i = 0; i < 3; i++) {         //n
        for (int j = 0; j < 3; j++) {     //n
            C[i][j] = 0;
            for (int k = 0; k < 3; k++) { //n
                C[i][j] += A[i][k] * B[k][j]; //$c_{i,j} = \sum a_{i,k} * b_{k,j}$
            }
        }
    }

    imprime_matriz(C);
}
```

Custo?

```c
void imprime_vetor(int v[], int n) {
    for (int i = 0; i < n; i+=2) {
        for (int j = 0; j<2 && i+j<n; j++) {
            printf("%d\n", v[i+j]);
        }
    }
}

void imprime(int n) {
    for (int i = 0; i < 1000; i++) {
        printf("%d\n", i+n);
    }
}

no *abrir(no *lista) {
    if(lista==NULL) return lista;
    return abrir(lista->prox);
}
```

Custo?

```c
int abrir_mapa(struct area campo[20][20], int l, int c) {
  campo[l][c].visivel = 1;

  if(campo[l][c].item!=0) return campo[l][c].item;

  for(int i=l-1; i<=l+1; i++) {   //3
    for(int j=c-1; j<=c+1; j++) { //3
      if(i>=0 && i<20 && j>=0 && j<20 &&
          campo[i][j].visivel==0) { //restringe as chamadas
                                    // cada posição é ``aberta''
                                    // 1 única vez
            abrir_mapa(campo, i, j);
      }
    }
  }

  return 0;
}
```

Exponencial: $f(n) = K^n$

- Problemas resolvidos por força bruta
  - Procurar a solução verificando as combinações das possibilidades
  - As chamadas recursivas aumentam múltiplas vezes por chamada
  - Alcance "lento" da condição de parada, gerando muitas chamadas recursivas
  - n múltiplicações sucessivas de uma base (número fixo de chamadas recursivas)
- Quando n é 20, o tempo é cerca de 1 milhão
- Exemplo: enumerar as linhas de uma tabela verdade 2^3

a	b	c
V	V	V
V	V	F
V	F	V
V	F	F
F	V	V
F	V	F
F	F	V
F	F	F

Exponencial: $f(n) = K^n$ - tabela verdade

```c
char T[2] = {'V', 'F'};
        //1o processa 2 instruções
        for(int c=0; c<2; c++){
            printf("%c %c %c\n", T[a], T[b], T[c]);
        }
```

Exponencial: $f(n) = K^n$ - tabela verdade

```c
char T[2] = {'V', 'F'};




    //2o processa 2^2 instruções
    for(int b=0; b<2; b++){

        //1o processa 2 instruções
        for(int c=0; c<2; c++){
            printf("%c %c %c\n", T[a], T[b], T[c]);
        }
    }

```

Exponencial: $f(n) = K^n$ - tabela verdade

```c
char T[2] = {'V', 'F'};

//3o processa 2^2^2 instruções
for(int a=0; a<2; a++){

    //2o processa 2^2 instruções
    for(int b=0; b<2; b++){

        //1o processa 2 instruções
        for(int c=0; c<2; c++){
            printf("%c %c %c\n", T[a], T[b], T[c]);
        }
    }
}
```

Exponencial: $f(n) = K^n$

```
int p(int n)
{
    if (n == 0) return 0;
    else if (n == 1) return 1;

    return p(n - 1) + p(n - 2);
}
```

```
Quantidade de chamadas
  2^0                   ____________ p(n) ___________
                       /                             \
  2^1              __ p(n-1) __                    __ p(n-2) __
                  /            \                  /            \
  2^2         p(n-2)          p(n-3)          p(n-3)          p(n-4)
              /    \          /    \          /    \          /    \
  2^3     p(n-3) p(n-4)  p(n-4) p(n-5)    p(n-4) p(n-5)  p(n-5) p(n-6)
           ...    ...     ...    ...       ...    ...     ...    ...
            |      |       |      |         |      |       |      |
  2^(n-1) p(1|0) p(1|0)  p(1|0) p(1|0) ... p(1|0) p(1|0)  p(1|0) p(1|0)
```

Exponencial: $f(n) = K^n$

```
//"fibonacci" modificado - custo piorado
int p(int n)
{
    if (n == 0 || n==1) return n;

    return p(n - 1) + p(n - 1);
}
```

$$
\begin{aligned}
f(n) &\approx 2 * f(n-1) \\
&\approx 2 * (2 * f(n-2)) \\
&\approx 2^2 * f(n-2) \\
&\approx 2^2 * (2 * f(n-3)) \\
&\approx 2^3 * f(n-3) \\
&\approx 2^i * f(n-i), i = n-1 \\
&\approx 2^{n-1} * f(1)
\end{aligned}
$$

Fatorial: $f(n) = n!$

- Problemas resolvidos por força bruta
  - Procurar a solução verificando a combinação de todas as possibilidades
  - Combinatória: custo fatorial
  - Cada chamada executa múltiplas (linear) chamadas recursivas
  - Alcance "lento" da condição de parada, gerando muitas chamadas recursivas
- Problema do caixeiro viajante: procura um circuito que possua a menor distância, começando em qualquer cidade, entre várias, visitando cada cidade precisamente uma vez e regressando à cidade inicial
- Para resolver:
  - Exatos: basicamente, analisam todas as alternativas possíveis (força bruta);
    - Existe o algoritmo de Held-Karp com custo exponencial ($n^2 * 2^n$)
  - Heurísticos: através de alguma heurística (estratégia) obtém-se soluções aproximadas
- Observação: é um problema verificável em tempo polinomial; dado uma sequência de cidades, verificar se passa por todas as cidades uma única vez.

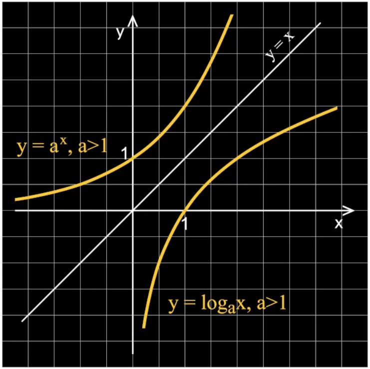
Fatorial: $f(n) = n!$

```c
void anagram(char str[], int k) {
    char tmp;
    int i, len = strlen(str);
    if(k == len) printf("%s\n", str);
    else {
        //cada chamada realiza: múltiplas chamadas recursivas
        for(i = k; i < len; i++) {
            swap_char(str, k, i); //1 troca
            anagram(str, k + 1);   //próximas permutas
            swap_char(str, i, k); //volta ao original
        }
    }
}
void subsets(int v[], int i, int n, int sub[], int fim) {
  for(int j = 0; j < fim; j++) printf("%d ", sub[j]);
  printf("\n");

  for(; i < n; i++) { //n
    sub[fim] = v[i];
    subsets(v, i+1, n, sub, fim+1); //n-1
  }
}
//f(n) = n*f(n-1);
//f(n) = n*(n-1)*(n-2)*...*1;
```

Logarítmico: $f(n) = \log n$

- Função logarítmica é a inversa da função exponencial



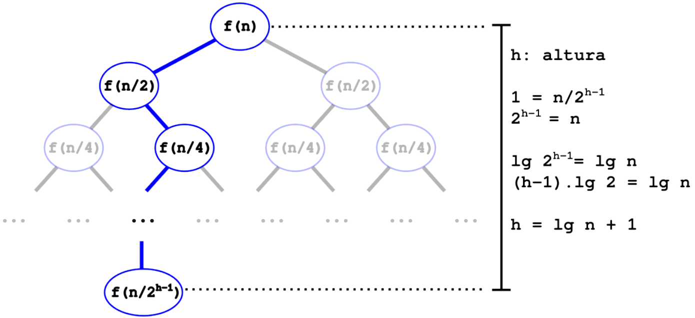
Logarítmico: $f(n) = \log n$

- O crescimento do custo fica um pouco mais lento a medida que n cresce
- Tempo típico de algoritmos que vai diminuindo a instância do problema a cada passo
  - Restrige o problema, em instâncias significativamente menores do problema
  - A solução do problema concentra-se na solução de sub-problema
- Não importa a base de log pois a grandeza do resultado não tem alterações significativas
  - Sendo $n = 1000$, $\log_2 n \approx 10$ ($\log_{10} n = 3$)
  - Sendo $n = 1000000$, $\log_2 n \approx 20$ ($\log_{10} n = 6$)

Logarítmico: $f(n) = \log n$

```
//vetor ordenado
int pesquisa (int x, int v[], int esq, int dir) {
    int meio = (esq + dir)/2;
    if (v[meio] == x) return meio;
    if (esq >= dir) return -1;
    else if (v[meio] < x)
        return pesquisa(x, v, meio+1, dir);
    else
        return pesquisa(x, v, esq, meio-1);
}
```

Logarítmico: $f(n) = \log n$



Logarítmico: $f(n) = \log n$

```
//vetor ordenado
int pesquisa (int x, int v[], int esq, int dir){
    int meio = (esq + dir)/2;
    if (v[meio] == x) return meio;
    if (esq >= dir) return -1;
    else if (v[meio] < x)
        return pesquisa(x, v, meio+1, dir);
    else
        return pesquisa(x, v, esq, meio-1);
}
```

$$
\begin{aligned}
f(n) &= f(n/2) + 1 \\
&= f(n/4) + 2 \\
&= f(n/8) + 3 \\
&= f(n/2^k) + k, 2^k = n : log_2 2^k = log_2 n : klog_2 2 = log_2 n : k = log_2 n \\
&= f(1) + log_2 n
\end{aligned}
$$

Linearítmica: $f(n) = n \log n$

- Divisão e conquista: problema quebrando em **problemas menores**, resolvendo cada um deles independentemente e **depois juntando as soluções**, localmente resolvidos, gerando um nova solução
- Dividir n em partes aproximadamente iguais
- Cada sub-problema resolvido em tempo linear

```
void intercala (int p, int q, int r, int v[]) {...}
void mergesort (int p, int r, int v[]){
    if (p < r-1) {
        int q = (p + r)/2;
        mergesort (p, q, v);
        mergesort (q, r, v);
        intercala (p, q, r, v);
    }
}
```

Linearítmica: $f(n) = n \log n$

$$
\begin{aligned}
f(n) &= f(n/2) + f(n/2) + n = 2 * f(n/2) + n \\
&= 2 * (2 * f(n/4) + n/2) + n = 4 * f(n/4) + 2 * n/2 + n \\
&= 4 * (2 * f(n/8) + n/4) + 2 * n = 8 * f(n/8) + 4 * n/4 + 2 * n \\
&= 8 * f(n/8) + 3 * n \\
&= \dots \\
&= 2^i * f(n/2^i) + i * n : i = log_2 n \\
&= n * f(n/n) + log_2 n * n = n * f(1) + n * log_2 n
\end{aligned}
$$

Roteiro

1 Medir algoritmos

2 Análise de algoritmos

3 Função custo: valores comuns

4 Complexidade

Análise Assintótica de algoritmos

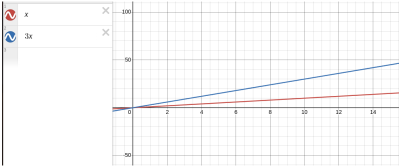
- Complexidade: aborda problemas computacionais e não instâncias particulares do problema
- Teoria da Complexidade está interessada em como os tempos de execução de algoritmos crescem com um aumento no tamanho das instâncias
- É uma medição formal (matematicamente consistente) de se **calcular aproximadamente** a eficiência de algoritmos
- Descreve a tendência de **crescimento das funções custo**
- E como representamos esse comportamento assintótico?

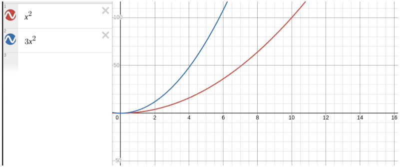
Notação O

- Para representar a relação assintótica surgiram diversas notações, sendo a **notação O** bastante utilizada
- A ideia das notações é achar uma **função** $g(n)$ que represente algum **limite** de $f(n)$
- Compara-se a **tendência de crescimento** de $f(n)$ e $g(n)$
- Quando $f(n) = O(g(n))$, temos
  - Informalmente: $f(n)$ cresce, no máximo, tão rapidamente quanto $g(n)$
  - $g(n)$ é o **limite superior** para a taxa de crescimento de $f(n)$
  - Diz-se que $g(n)$ **domina assintoticamente** $f(n)$
- Exemplo: busca sequencial
  - $O(n)$, custo cresce, no máximo, conforme n cresce

Notação O

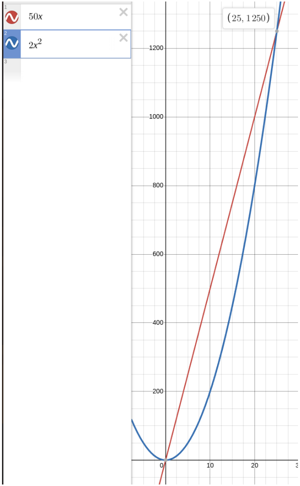
- Supondo em um programa
  - Com custo de $f(n) = 4n^2 + 4n + 1$
  - A medida que n aumenta, o termo quadrático começa a dominar
  - Para n muito grandes, diminui-se o impacto da constante multiplicativa do termo quadrático e dos outros termos
  - Assim, temos que $f(n) = O(n^2)$
- A dominação assintótica revela a **equivalência** entre os algoritmos
  - Sendo F e G algoritmos da mesma classe
  - Com $f(n) = 3.g(n)$, mesmo F sendo 3 vezes mais “custosa” que G
  - Possuem a mesma complexidade $O(f(n)) = O(g(n))$
  - **Mesma tendência de crescimento**; mesmo comportamento





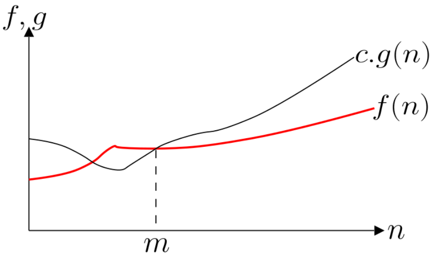
Notação O

- Outro aspecto a ser considerado é o **tamanho do problema** a ser executado
  - **Uma complexidade $O(n)$ em geral representa um programa mais eficiente que um $O(n^2)$**
  - Porém dependendo do valor de n, um algoritmo $O(n^2)$ poder ser mais indicado do que o $O(n)$
  - Por exemplo, com $f(n) = 50.n$ e $g(n) = 2.n^2$
    - ★ Problemas com $n < 25$
    - ★ $g(n) = 2.n^2$ é mais eficiente do que um $f(n) = 50.n$



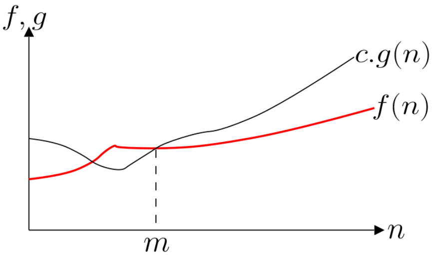
Notação O

- Formalmente, define-se:
  - Uma função $f(n) = O(g(n))$
  - Se $f(n) \leq c.g(n)$, para algum c positivo e para todo n suficiente grande, ou seja,



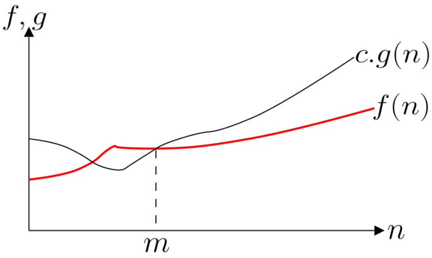
Notação O

- Formalmente, define-se:
  - Uma função $f(n) = O(g(n))$
  - Se $f(n) \leq c.g(n)$, para algum c positivo e para todo n suficiente grande, ou seja,
    - Existem duas constantes positivas c e m
    - Tais que, $f(n) \leq c.g(n)$
    - Para todo $n \geq m$ (ponto inicial do tendência para o comportamento)



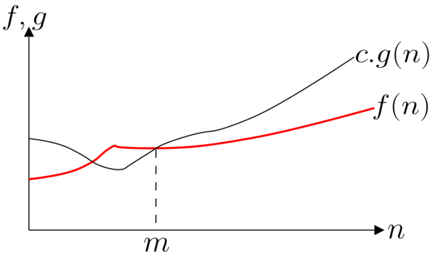
Notação O

- Formalmente, define-se:
  - Uma função $f(n) = O(g(n))$
  - Se $f(n) \leq c.g(n)$, para algum c positivo e para todo n suficiente grande, ou seja,
    - ★ Existem duas constantes positivas c e m
    - ★ Tais que, $f(n) \leq c.g(n)$
    - ★ Para todo $n \geq m$ (ponto inicial do tendência para o comportamento)



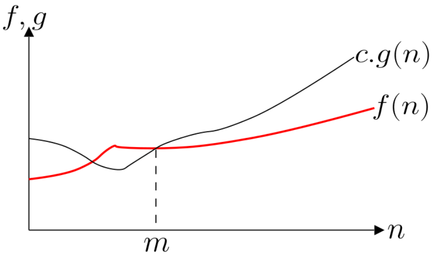
Notação O

- Formalmente, define-se:
  - Uma função $f(n) = O(g(n))$
  - Se $f(n) \leq c.g(n)$, para algum c positivo e para todo n suficiente grande, ou seja,
    - Existem duas constantes positivas c e m
    - Tais que, $f(n) \leq c.g(n)$
    - Para todo $n \geq m$ (ponto inicial do tendência para o comportamento)



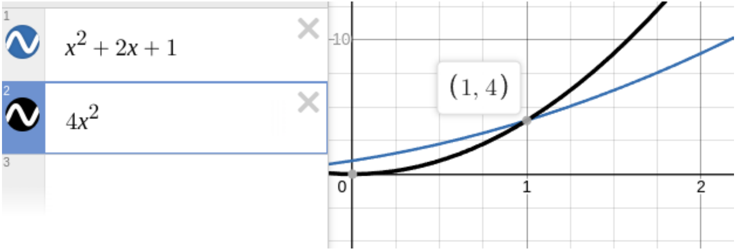
Notação O

- Formalmente, define-se:
  - Uma função $f(n) = O(g(n))$
  - Se $f(n) \leq c.g(n)$, para algum c positivo e para todo n suficiente grande, ou seja,
    - ⋆ Existem duas constantes positivas c e m
    - ⋆ Tais que, $f(n) \leq c.g(n)$
    - ⋆ Para todo $n \geq m$ (ponto inicial do tendência para o comportamento)



Notação O - exemplo

- Com tempo de execução $f(n) = (n+1)^2$
  - Temos que $f(n) = O(n^2)$
  - Existem as constantes $m = 1$ e $c = 4$
  - E para todo $n \geq 1$, temos a relação $n^2 + 2n + 1 \leq 4.n^2$



Notação O - equivalência na tendência ao infinito

- Temos que:
  - $f(n) = O(g(n))$ se $\dfrac{\lim_{n\to\infty} f(n)}{\lim_{n\to\infty} g(n)}$ for constante
  - **Se o fator de proporcionalidade entre $f(n)$ e $g(n)$, for constante**
  - **Se são diretamente proporcionais**
- Demonstração:
  - $f(n) = a_i n^i + a_{i-1} n^{i-1} + ... + a_1 n^1 + a_0 n^0$
  - $f(n) = O(n^i)$

    Como:

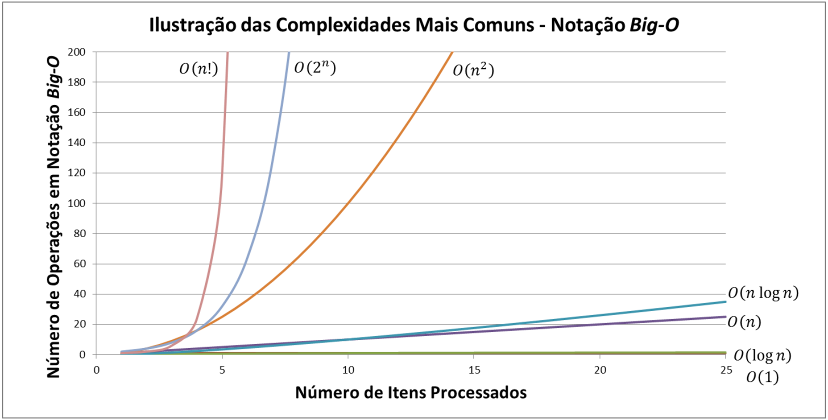
$$
\begin{aligned}
\lim_{n\to\infty} f(n) &= \lim_{n\to\infty} (a_i n^i + a_{i-1} n^{i-1} + ... + a_1 n^1 + a_0 n^0) \\
&= \lim_{n\to\infty} (a_i n^i + a_{i-1} n^i n^{-1} + ... + a_1 n^i n^{-(i-1)} + a_0 n^i n^{-i}) \\
&= \lim_{n\to\infty} n^i (a_i + \frac{a_{i-1}}{n} + ... + \frac{a_1}{n^{i-1}} + \frac{a_0}{n^i}) \\
&= \lim_{n\to\infty} a_i n^i
\end{aligned}
$$

    De forma análoga para $g(n) = n^i$, temos:

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

Notação O - Operações x Entradas

- Complexidades polinomiais são significativamente mais eficientes que as exponenciais
  - Problema bem resolvido: possui uma solução polinomial
  - n^2, n^3, n, $\log n$, $n \log n$

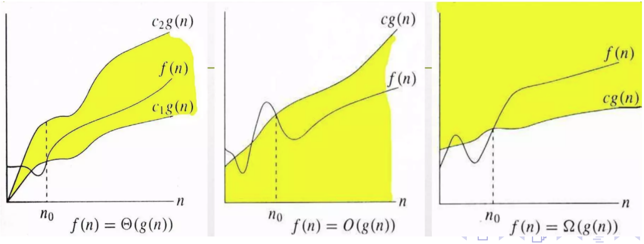


Notação O - Custo das representações das adjacências

- Matriz:
  - matriz[i][j] = 1, diz-se que o vértice i está conectado ao vértice j
  - Acesso aleatório e direto $O(1)$
  - Em grafos esparsos, desperdício de espaço $O(V^2)$;
  - Processar todos os elementos $O(V^2)$
- Lista:
  - Células/nós relação entre dois vértices
  - Em grafos esparsos, custo espacial é menor $O(V + A)$
  - Processar todos elementos $O(V + A)$
  - Se $A \geq V$, como acontece em muitas aplicações, o consumo de tempo é proporcional a A (linear)
  - Em grafos densos, com a maioria dos vértices conectados entre si, a vantagem espacial é menor do que a desvantagem do acesso $O(V + A) = O(V + V * (V - 1)) = O(V^2)$

Outras Notações

- Ω (omega)
  - Representa o **limite inferior** para função $f(n)$
  - $f(n)$ cresce, no mínimo, tão lento quanto $g(n)$
  - $f(n)$ é $\Omega(g(n))$
    - ⋆ Se existirem duas constantes c e m
    - ⋆ Tais que $|f(n)| \geq c.|g(n)|$
    - ⋆ Para todo $n \geq m$
- θ (theta)
  - Função $f(n)$ é **limitada superiormente e inferiormente** à $g(n)$
  - $f(n)$ cresce tão rápido quanto $g(n)$
  - Com uma diferença de apenas uma constante, ou seja
    - ⋆ $0 \leq c1.|g(n)| \leq f(n) \leq c2.|g(n)|$
  - Notação para representar maior precisão



Complexidade algoritmos recursivos

Teorema mestre

- Muitas das recorrências que ocorrem na análise de algoritmos de divisão e conquista têm a seguinte forma (com a, c e k constantes):
  - $T(n) = aT(n/2) + cn^k$
- Para resolver recorrências com a forma acima, podemos utilizar o teorema mestre:
  - $T(n) = aT(n/b) + cn^k$
    - $T(n)$: custo para n entradas
    - a: quantidade de subproblemas
    - n/b: tamanho de cada subproblema
    - $T(n/b)$: custo para cada subproblema
    - cn^k: custo para dividir e combinar os resultados
  - se $\lg a / \lg b > k$, então $T(n) = \theta(n^{\lg a / \lg b})$
  - se $\lg a / \lg b = k$, então $T(n) = \theta(n^k \lg n)$
  - se $\lg a / \lg b < k$, então $T(n) = \theta(n^k)$

Complexidade algoritmos recursivos

- $T(n) = T(n/2) + 1$
  - a = 1, b = 2, c = 1, k = 0
- se $\lg a / \lg b = k$, então $T(n) = \theta(n^k \lg n)$
  - $\lg 1 / \lg 2 = 0/1 = 0 = k$, então $T(n) = \theta(n^0 \lg n) = \theta(\lg n) \rightarrow O(\lg n)$

```
//vetor ordenado
int pesquisa (int x, int v[], int esq, int dir){
    int meio = (esq + dir)/2;
    if (v[meio] == x) return meio;
    if (esq >= dir) return -1;
    else if (v[meio] < x)
        return pesquisa(x, v, meio+1, dir);
    else
        return pesquisa(x, v, esq, meio-1);
}
```

Complexidade algoritmos recursivos

- $T(n) = 2 * T(n/2) + n$
  - a = 2, b = 2, c = 1, k = 1
- se $\lg a / \lg b = k$, então $T(n) = \theta(n^k \lg n)$
  - $\lg 2 / \lg 2 = 1/1 = 1 = k$, então $T(n) = \theta(n^1 \lg n) = \theta(n \lg n) \rightarrow O(n \lg n)$

```c
void intercala (int p, int q, int r, int v[]) {...}
void mergesort (int p, int r, int v[]){
    if (p < r-1) {
        int q = (p + r)/2;
        mergesort (p, q, v);
        mergesort (q, r, v);
        intercala (p, q, r, v);
    }
}
```