Estruturas de Dados

Noções de complexidade

Projeto de algoritmos

- Nem todos os algoritmos desenvolvidos para resolver um mesmo problema sempre tem o mesmo custo;
  - Alguns algoritmo levam mais tempo que outros;
  - Alguns algoritmos consomem mais memória que outros;
- O projeto de um algoritmo pode afetar no tempo levado para a resolução do problema de maneira mais relevante que a velocidade do computador;

Projeto de algoritmos

- Suponha o seguinte cenário:
  - Computador A que executa 10^{10} instruções por segundo;
  - Computador B que executa 10^7 instruções por segundo;
  - Algoritmo X que executa $2n^2$ instruções para ordenar n elementos;
  - Algoritmo Y que executa 50 n lg n instruções para ordenar n elementos;
- O que terminaria mais rápido?
  - Executar o algoritmo X no computador A ou executar o algoritmo Y no computador B para 10^7 elementos?

Obs: $\lg n = \log_2 n$

Projeto de algoritmos

- Computador A executando algoritmo X:
  - Total de instruções

    $$2 \times (10^7)^2 = 2 \times 10^{14} \text{ instruções}$$

  - Total de tempo:

    $2 \times 10^{14} / 10^{10} = 20000$ segundos (mais de 5 horas e meia)
- Computador B executando algoritmo Y:
  - Total de instruções:

    $$50 \times 10^7 \log_2 10^7 \approx 10^7 \times 23{,}25 \times 50 \approx 1162{,}5 \times 10^7$$

  - Total de tempo:

    $10^7 \times 1162{,}5 / 10^7 = 1162{,}5$ segundos (menos de 20 minutos)

Projeto de algoritmos

- Como a eficiência de um algoritmo é importante para o desempenho obtido na solução do problema, se torna uma área de muita importância a análise de algoritmos;
- Na área de análise de algoritmos existem dois tipos de problemas distintos:
  - Análise de um algoritmo em particular;
    - Qual o custo de usar um dado algoritmo para resolver um problema específico?
  - Análise de uma classe de algoritmos;
    - Qual o algoritmo de menor custo para resolver um problema particular?

Análise de algoritmos

- A análise de algoritmos em particular tem por objetivo fazer uma previsão do quanto é consumido de cada um dos recursos por um determinado algoritmo;
- A análise de um algoritmo pode ser feita de duas maneiras:
  - Por meio do Método Experimental;
  - Por meio do Método Analitico;

Análise de algoritmos

- Método Experimental:
  - Realizar várias implementações completas do algoritmo;
  - Executar uma grande quantidade de testes;
  - Analisar estatisticamente os resultados;
- Existem alguns motivos para não usar o método experimental:
  - Resultados dependem do compilador;
  - Resultados dependem do hardware;
  - Tempo pode ser influenciado pelo uso de memória;

Análise de algoritmos

- Método Analitco:
  - Usar um modelo matemático, baseado em um computador idealizado;
  - Extrair funções que descrevem a complexidade do algoritmo em função do tamanho do problema;
    - Complexidade de tempo;
    - Complexidade de espaço;

Análise de algoritmos

- O que é a eficiência de um algoritmo?
  - Medida Quantitativa;
  - O inverso da quantidade de recursos requeridos para o seu funcionamento
- Por que medir a eficiência de um algoritmo?
  - Prever a quantidade de recursos a serem usados;
    - De espaço de memória (complexidade de espaço);
    - De tempo de execução (complexidade de tempo);
      - A preocupação mais frequente na área de análise de algoritmos;
  - Escolher um modelo de tecnologia adequado para sua implementação;

Análise de algoritmos

- Adotar um modelo matemático de computador torna a análise de algoritmos mais abrangente;
- Existem diversos modelos matemáticos de computadores;
- O modelo adotado para a análise de nossos algoritmos será a máquina de acesso aleatório (random-access machine - RAM) de um único processador;
  - Instruções executadas em sequência sem operações concorrentes;
  - Composto por operações:
    - Aritméticas e lógicas;
    - De movimentação;
    - De Desvio
  - Cada operação é executada em uma unidade de tempo;

Análise de algoritmo

- Em alguns casos algumas análises podem ser simplificadas, levando em consideração o custo das operações mais significativas;
  - A quantidade de comparações entre o elemento procurado e os elementos do vetor em um algoritmo de busca;
  - Quantidade de comparações e trocas num algoritmo de ordenação;

Exemplo

- Considere o seguinte algoritmo, implementado em C, para procurar o maior valor entre um vetor de inteiros;

```
1.  int max(int v[], int n){
2.  //n é a quantidade de elementos
    em v
3.  int i, temp;
4.  temp=v[0];
5.      for(i=1;i<n;i++){
6.          if(temp < v[i]){
7.              temp=v[i]
8.          }
9.      }
10. }
```

Exemplo

- Qual a quantidade de comparações para um vetor de n elementos?

```
1.  int max(int v[], int n){
2.  //n é a quantidade de elementos em v
3.  int i, temp;
4.  temp=v[0];
5.      for(i=1;i<n;i++){
6.          if(temp < v[i]){
7.              temp=v[i]
8.          }
9.      }
10. }
```

Exemplo

- Qual a quantidade de comparações para um vetor de n elementos?
  - Para responder essa questão devemos analisar quantas vezes a linha 6 será executada;
  - A variável de controle do laço for irá assumir os valores 1,2,3,...n-1
  - Dessa forma, em um vetor com n elementos (indexados de 0 até n-1) realizará comparações entre temp e cada elemento do vetor, exceto o primeiro, que está na posição 0;
  - Assim, temos um total de n-1 comparações

```
1.  int max(int v[], int n){
2.  //n é a quantidade de elementos
    em v
3.  int i, temp;
4.  temp=v[0];
5.      for(i=1;i<n;i++){
6.          if(temp < v[i]){
7.              temp=v[i]
8.          }
9.      }
10. }
```

Análise de algoritmos

- A função que expressa a complexidade de tempo de execução é geralmente expressa de acordo com o tamanho da entrada do algoritmo;
- Para alguns, além do tamanho da entrada, o desempenho também é afetado de acordo com algumas entradas em particular;

Exemplo 2

- Suponha um algoritmo que deseja verificar a posição de uma chave única em um vetor, implementado a seguir

```
1.  int buscar(int v[],int n, int chave){ //sendo n a
    quantidade de elementos em v
2.      int i, pos;
3.      i=0;
4.      pos = -1;
5.      while((i<n) && (pos == -1)){
6.          if(v[i]==chave){
7.              pos = i;
8.          }
9.          i++;
10.     }
11.     return pos;
12. }
```

Exemplo 2

- Como poderíamos extrair a quantidade de comparações entre elementos do vetor e a chave buscada?

Exemplo 2

- Para isso precisamos responder a algumas perguntas:
  - O que aconteceria se o primeiro elemento do vetor tivesse o valor da chave?
    - Esse seria o caso em que o tempo de execução teria o melhor desempenho;
      - Menor tempo de execução dentre todos os possíveis para entradas de tamanho n para o algoritmo;
      - Também chamado de **melhor caso**;
  - O que aconteceria se o valor da chave não estivesse em nenhum elemento do vetor?
    - Esse seria o caso em que o tempo de execução teria o maior desempenho;
      - Maior tempo de execução dentre todos os possíveis para entradas de tamanho n para o algoritmo;
      - Também chamado de **pior caso**;

Exemplo 2

- Número de comparações no melhor caso:
  - No melhor caso o elemento procurado é o primeiro;
  - Dessa forma, a quantidade de comparações (execuções da linha 6) é apenas uma;
    - O algoritmo irá identificar que o elemento é o procurado, ajustar as variáveis e retornar que encontrou o elemento;
  - Assim, a quantidade total de operações de comparação é 1, logo, no **melhor caso**:

$$f(n) = 1;$$

Exemplo 2

- Número de comparações no pior caso:
  - No pior caso o elemento não está no vetor;
  - Dessa forma, a quantidade de comparações (execuções da linha 6) é exatamente igual a quantidade de elementos que existe no vetor;
    - Deve-se comparar o elemento procurado com todos elementos do vetor;
  - Assim, o total de operações de comparação é a quantidade de elementos que existem no vetor, ou seja, no **pior caso**:

$$f(n) = n$$

Exemplo 2

- Qual o desempenho esperado para o algoritmo?
  - O desempenho esperado está relacionado à média dos tempos de execução de todas entradas de tamanho N;
  - Para calcular a média é necessário supor distribuição de probabilidades;
  - É a análise mais difícil!

Exemplo 2

- Para analisarmos o caso médio vamos simplificar e supor o seguinte:
  - Toda pesquisa a ser feita irá encontrar o valor buscado;
    - Desconsiderar os casos em que o valor não é encontrado
  - Uma probabilidade p_i de que o i-ésimo registro seja o procurado;
- Como para que o i-ésimo registro seja o recuperado são necessárias i comparações, temos:

$$f(n) = 1 \times p_1 + 2 \times p_2 + \ldots + n \times p_n$$

- Dessa forma, basta conhecer a distribuição de probabilidades p_i para calcular o caso médio;

Exemplo 2

- Assumindo que cada elemento do vetor tem a mesma probabilidade, ou seja, $p_i = 1/n$, tal que $1 \leq i \leq n$, temos:

$$f(n) = 1 \times \frac{1}{n} + 2 \times \frac{1}{n} + \ldots + n \times \frac{1}{n}$$

$$f(n) = \frac{1}{n} \times (1 + 2 + \ldots + n)$$

$$f(n) = \frac{n+1}{2}$$

Exemplo 3

- Supondo a seguinte implementação do algoritmo selection sort

```
void selection(int v[], int n){
        int i,j, idx_menor, aux;
        for(i=0;i < n-1 ;i++){
                idx_menor = i;
                for(j=i+1;j < n ;j++){
                        if(v[j]<v[idx_menor]){
                                idx_menor=j;
                        }
                }
                aux=v[i];
                v[i]=v[idx_menor];
                v[idx_menor]=aux;
        }
}
```

Exemplo 3

- Qual a quantidade de comparações para um vetor de n elementos?

```
void selection(int v[], int n){
        int i,j, idx_menor, aux;
        for(i=0;i < n-1 ;i++){
                idx_menor = i;
                for(j=i+1;j < n ;j++){
                        if(v[j]<v[idx_menor]){
                                idx_menor=j;
                        }
                }
                aux=v[i];
                v[i]=v[idx_menor];
                v[idx_menor]=aux;
        }
}
```

Exemplo 3

- Qual a quantidade de comparações para um vetor de n elementos?
  - Existem 2 laços no algoritmo, e a comparação está dentro de ambos:
    - O bloco do laço externo é executado n-1 vezes;
    - O bloco do laço interno tem sua execução dependendo do índice que controla o laço externo;
      - Tal bloco é executado (n-1) + (n-2) + (n-3) + (n-4)+...+1;
      - A quantidade de comparações é dada por:

$$\sum_{1}^{n} n-i$$

```
void selection(int v[], int n){
        int i,j, idx_menor, aux;
        for(i=0;i < n-1 ;i++){
                idx_menor = i;
                for(j=i+1;j < n ;j++){
                        if(v[j]<v[idx_menor]){
                                idx_menor=j;
                        }
                }
                aux=v[i];
                v[i]=v[idx_menor];
                v[idx_menor]=aux;
        }
}
```

Exemplo 3

- Resolvendo a somatória:

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

- Assim tem-se que:

$$f(n) = \frac{n^2 - n}{2}$$

Análise de algoritmos

- Para valores pequenos de n todos algoritmos tem custo baixo;
  - Inclusive os algoritmos ineficientes!
- A análise de complexidade de algoritmos é realizada para valores de n grandes;

Notação assintótica

- Para expressar a eficiência de um algoritmo, o mais comum é usar a ordem de crescimento do tempo de execução;
- Uma boa maneira de expressar a ordem de crescimento do tempo de execução é utilizando notações assintóticas;

Notação assintótica

- A notação assintótica descreve o crescimento de funções;
- Abstrai os termos de baixa ordem e constantes;
- É utilizada para comprar funções;
  - De maneira similar as operações relacionais entre números;

Notação assintótica

- Existem diferentes notações para comprar o crescimento de funções, dentre elas estão:
  - Notação O (big o);
  - Notação Ω (big ômega);
  - Notação Θ (big teta);

Notação Big o

- Notação O (Limite Superior):
  - A notação O define um limite superior para para a função;
  - Para uma dada função g(n) define-se O(g(n)) como o conjunto de funções:

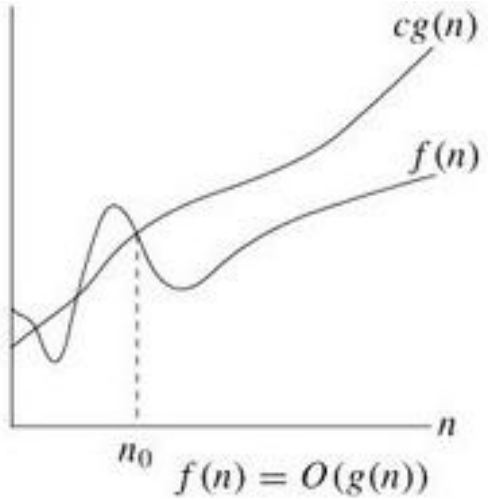
$$O(g(n)) = \{f(n): \exists\ c, n_0 : c, n_0 > 0 \mid 0 \leq f(n) \leq cg(n), \ \forall\ n \geq n_0\}$$

  - Ou ainda::

$$O(g(n)) = \{f(n):\text{existem constantes positivas } c, n_0 \text{ tais que } 0 \leq f(n) \leq cg(n) \text{ para todo } n \geq n_0\}$$

Notação O (Big o)

- Em outras palavras:
  - Dadas duas funções f(n) e g(n), se f(n) pertence a O(g(n)) então:
    - A taxa de crescimento de g(n) é maior que a taxa de crescimento de f(n) a partir de um ponto n_0;



Notação O (Big o)

- Desse modo, para provar que uma função f(n) pertence ao conjunto O(g(n)) devemos encontrar as constantes c e n_0 que satisfazem a inequação;
- Por exemplo:
  - Provar que $f(n)=n^2+2n+1$ pertence a $O(n^2)$;
    - Temos que achar constantes c e n_0 tais que:
      - $n^2+2n+1 \leq cn^2$

$$n^2 + 2n + 1 \leq cn^2 \rightarrow 1 + \frac{2}{n} + \frac{1}{n^2} \leq c$$

      - Assim, com $c \geq 4$ e $n \geq 1$ temos que $f(n)=n^2+2n+1$ pertence a $O(n^2)$;

Notação Ω (Big ômega)

- Notação Ω (Limite inferior)
  - A notação Ω define um limite inferior para a função
  - Para uma dada função g(n) define-se Ω(g(n)) como o conjunto de funções:

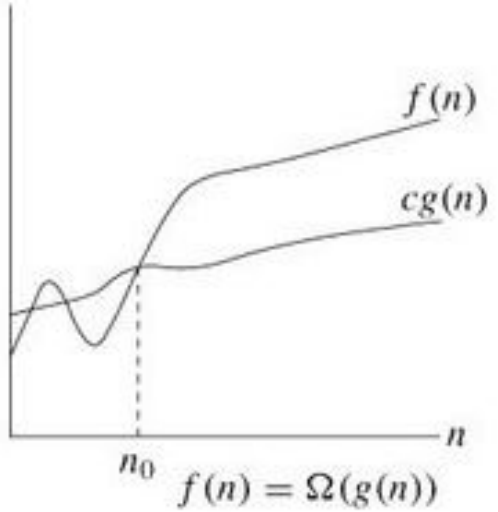
$$\Omega(g(n)) = \{f(n): \exists\ c, n_0 : c, n_0 > 0 \mid 0 \leq cg(n) \leq f(n)\ ,\ \forall\ n \geq n_0\}$$

  - Ou ainda::

$\Omega(g(n)) = \{f(n)$:existem constantes positivas c, n_0 tais que $0 \leq cg(n) \leq f(n)$ para todo $n \geq n_0\}$

Notação Ω (Big ômega)

- Em outras palavras:
  - Dadas duas funções f(n) e g(n), se f(n) pertence a Ω(g(n)) então:
    - A taxa de crescimento de g(n) é maior que a taxa de crescimento de f(n) a partir de um ponto n_0;



Notação Ω (Big ômega)

- Desse modo, para provar que uma função f(n) pertence ao conjunto Ω (g(n)) devemos encontrar as constantes c e n_0 que satisfazem a inequação;
- Por exemplo:
  - Provar que $f(n)=n^2+2n+1$ pertence a $\Omega(n^2)$;
    - Temos que achar constantes c e n_0 tais que:
      - $cn^2 \leq n^2+2n+1$

$$cn^2 \leq n^2 + 2n + 1 \rightarrow c \leq 1 + \frac{2}{n} + \frac{1}{n^2}$$

      - Assim com $c \leq 1$ e $n \geq 1$ temos que $f(n)=n^2+2n+1$ pertence a $\Omega(n^2)$

Notação Θ (Big teta)

- Notação Θ (Limite restrito):
  - A notação Θ define um limite restrito para para a função;
  - Para uma dada função g(n) define-se Θ(g(n)) como o conjunto de funções:

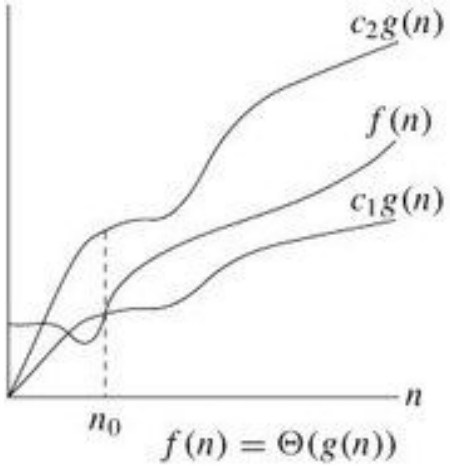
$$\Theta(g(n)) = \{f(n): \exists\ c_1, c_2, n_0: c_1, c_2, n_0 > 0 \mid c_1 g(n) \leq f(n) \leq c_2 g(n),\ \forall\ n \geq n_0\}$$

  - Ou ainda:

$$\Theta(g(n)) = \{f(n):\text{existem constantes positivas } c_1, c_2 \text{ e } n_0 \text{ tais que } c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ para todo } n \geq n_0\}$$

Notação Θ (Big teta)

- Em outras palavras:
  - Dadas duas funções f(n) e g(n), se f(n) pertence a (g(n)) então:
    - A taxa de crescimento de g(n) é a mesma que a taxa de crescimento de f(n) a partir de um ponto n_0;



Notação Θ (Big teta)

- Desse modo, para provar que uma função f(n) pertence ao conjunto Θ (g(n)) devemos encontrar as constantes c_1, c_2 e n_0 que satisfazem a inequação;
- Por exemplo:
  - Provar que f(n)=n^2+2n+1 pertence a Θ (n^2);
    - Temos que achar constantes c_1, c_2 e n_0 tais que:
      - $c_1n^2 \leq n^2+2n+1 \leq c_2n^2$

$$c_1 n^2 \leq n^2 + 2n + 1 \leq c_2 n^2 \rightarrow c_1 \leq 1 + \frac{2}{n} + \frac{1}{n^2} \leq c_2$$

- Assim com $c_1 \leq 1$, $c_2 \geq 4$ e $n \geq 1$ temos que f(n)=n^2+2n+1 pertence a Θ (n^2);

Notação Assintótica

- É possível fazer uma analogia entre as notações assintóticas e as operações relacionais entre números reais:

$$O \approx \leq$$
$$\Omega \approx \geq$$
$$\Theta \approx =$$

Classes de funções de complexidade

- As principais classes de problemas possuem as seguintes funções de complexidade:
  - $f(n) = O(1)$;
  - $f(n) = O(\log n)$;
  - $f(n) = O(n)$;
  - $f(n) = O(n \log n)$;
  - $f(n) = O(n^2)$;
  - $f(n) = O(n^3)$;
  - $f(n) = O(2^n)$;
  - $f(n) = O(n!)$;

Exercícios

1. Dado o algoritmo bubble sort, apresentado a seguir, extraia a função de complexidade de tempo, considerando somente o número de comparações, e mostre que tal função é $O(n^2)$;

```
void bubble(int v[], int n){
    int i,j,aux;
    for(i=0;i<n-1;i++){
        for(j=0;j<(n-1)-i;j++){
            if(v[j]>v[j+1]){
                aux=v[j]; v[j]=v[j+1]; v[j+1]=aux;
            }
        }
    }
}
```

2. Mostre que o pior caso da quantidade de comparações do algoritmo Selection sort é $\Theta(n^2)$.