1955SCC - Projeto e Análise de Algoritmos

Prof. MSc. André Furlan - andre.furlan@unesp.br

Universidade Estadual Paulista Júlio de Mesquita Filho

2022

unesp

Apresentação

unesp

- KLEINBERG, J., TARDOS, E.; Algorithm Design, Addison-Weslwy, 2005.
- CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., STEIN, C.;
- Introduction to Algorithms, 3rd Edition, The MIT Press, 2009.
- SKIENA, S. S., The algorithm design manual, Springer, 2010.

Espera-se que ao final do semestre o aluno domine as técnicas para análise de algoritmos, envolvendo o comportamento assintótico de sua execução e de sua eficiência na solução de problemas específicos. Espera-se também, que o mesmo tenha condições de julgar e escolher as melhores alternativas, tanto do ponto de vista de tempo como espaço, para algoritmos aplicados em problemas reais.

- Revisão de álgebra (funções assintóticas).
- Relações de recorrência.
- Análise de algoritmos através de funções assintóticas, incluindo notação Big-O e demais funções assintóticas.
- Análise de problemas NP-completo e reducibilidade.
- Aplicação da análise assintótica na avaliação de algoritmos de ordenação.
- Análise e projeto de algoritmos de busca.
- Algoritmos randômicos.
- Algoritmos baseados em abordagem gulosa.
- Algoritmos baseados e programação dinâmica.
- Algoritmos para grafos: caminho mínimo, árvore geradora, detecção de ciclos, etc.

Provas práticas (análise e programação) e escritas (discussões).

Etapas da programação

unesp

Etapas de programação

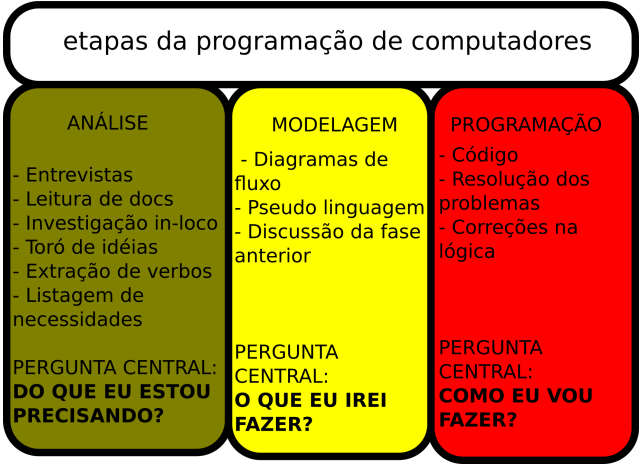


Figura: Etapas de programação

unesp

Etapas para programação

- Entendimento das entradas e saídas.
- Definir estrutura de dados.
- Implementação.
- Demonstrar que funciona.

unesp

Algoritmos

Definição

"Uma série de passos **finitos** e **bem definidos** que resolvem um determinado problema". Sendo assim, um algoritmo deve permitir uma **entrada** que, consequentemente, gerará uma **saída**.

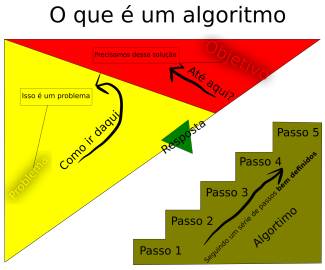


Figura: Algoritmo

Exemplos

"Ande".
"Transações".
"Faça um bolo".
"Resolva uma equação".

unesp

Testando o algoritmo

- Experimentação.
- Análise assintótica.

O foco é no tempo de execução e consumo de memória.

O tempo de execução pode variar segundo:

- A natureza do problema: Tamanho da entrada, formato da entrada, complexidade.
- Hardware: Tipo e quantidade de processadores, discos, etc.
- Software: Kernel, escalonamento de processos, prioridades de processo.

Na experimentação geralmente se mede o tempo de execução variando-se, se possível, a quantidade de núcleos, processos, memória e/ou diferentes formas de escalonamento.

Um cuidado que deve ser tomado nesse tipo de teste é que algum caso particular pode não ser detectado, resultando assim em comportamentos inesperados para certas entradas, portanto, em alguns casos a experimentação sozinha não é suficiente.

Exemplo: Para uma entrada 1500 elementos um algoritmo **A** executa melhor que um algoritmo **B**, por isso, você deseja usa-lo. Será que essa opção foi a melhor, porque?

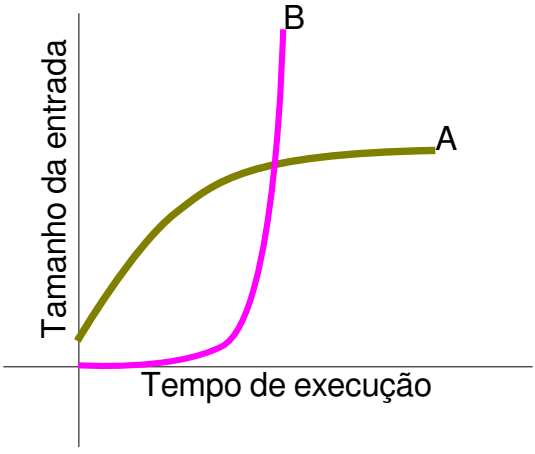


Figura: Comparação entre algoritmos

Análise assintótica

Dada uma $f(x) = b$ uma assíntota é a curva dada pela equação 1.

$$\lim_{x \to \infty} f(x) = b \quad , \qquad (1)$$

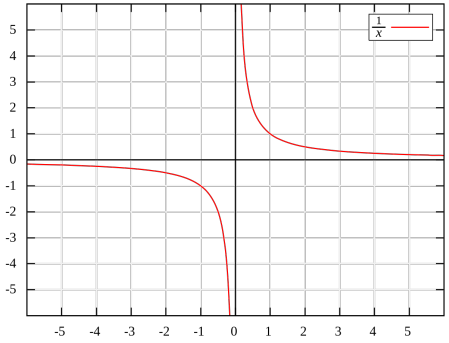


Figura: A função f(x)=1/x tem como assíntotas os eixos coordenados. Fonte: Wikipedia

unesp

Antes de fazer a análise assintótica é necessário aprender a contar o número de instruções primitivas que estão sendo executadas. Geralmente se considera apenas algumas operações elementares como comparações e atribuições.

```
1
2     int valor = 0;                    //Custo 1
3     int vetor[] = int[100];           //Custo 1
4     for (i=0; i < 100; i++){          // <- 100 vezes
5       soma += vetor[i];               //Custo 1
6     }
7
8     std::cout << soma << std::endl; //Custo 1
9
```

Algoritmo: Um algorítimo de tempo constante

```
1
2     int somatorio(int n, int vetor[]){
3
4       int valor = 0;                //Custo 1
5       for (i=0; i < n; i++){        // <- n vezes
6         soma += vetor[i];           //Custo 1
7       }
8
9       return soma;                  //Custo 1
10    }
11
```

Algoritmo: Um algorítimo de tempo variável

Em ordem de tempo crescente:

- 1 - constante
- $\log n$ - logarítimica
- n - linear
- $n.\log n$ - logarítimica
- n^2 - quadrática, cúbica, etc.
- 2^n - exponencial
- $n!$ - fatorial
- n^n - vixi...

Ver o exemplo *ordensDeComportamento.ipynb* no jupyter-lab.

O limite superior de um algoritmo é o **maior tempo** que o mesmo leva para executar.
O limite inferior é o **menor tempo**.

```
1  int menor(int vetor[], int n){
2    int menor = INT_MAX;  // 1
3    for(int i = 0; i < n; i++){ // n
4      if (vetor[i] < menor){ // 1
5        menor = vetor[i]; // 1
6      }
7    }
8
9    if(menor < 0){ // 1
10     for(int i = 0; i < n; i++){ // n
11       for(int j = 0; j < n; j++){ // n
12         vetor[i] = vetor[i]^(i+j); // 1
13       }
14     }
15   }
16   return menor; // 1
17 }
```

Dependendo do valor do menor elemento o comportamento do algorítimo muda.
Testemos o código acima com $vetor = \{1, 2, 3, 4, 5\}$ e com $vetor = \{10, 5, 3, 2, -4\}$ e $n = 5$.

Melhor Caso: $1 + 1 * n + 1 + 1 + 1 = n + 4$ (linear)
Pior Caso: $1 + 2 * n + 1 + 1 * n * n + 1 = n2 + 2n + 3$ (quadrático)

A complexidade desse algoritmo pode ser descrita como: $\Omega(n)$ e $O(n^2)$

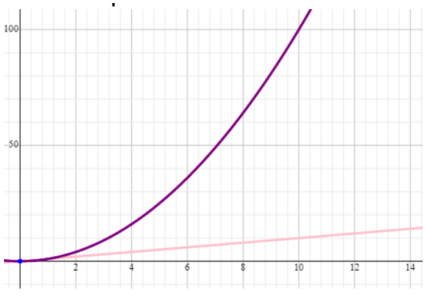


Figura: Gráfico do comportamento do algorítimo

unesp

Como já foi dito o tempo do algoritmo pode variar segundo a entrada de dados, considerando isto, se faz necessário avalia-lo segundo seu **melhor** e **pior** caso de execução.
O melhor caso de execução é aquele que no qual o algoritmo leva menos tempo (executa menos comandos) o pior é aquele que ocupa mais.

```
1    int soma = 0;
2    for (int i=0; i<n; i++)
3    soma = soma + i;
4
```

Algoritmo: Algoritmo 1

```
1    int soma1 = 0;
2    int soma2 = 0;
3    for (int i=0; i<n; i++){
4      soma1 = soma1 + 1;
5      soma2 = soma2 + i;
6    }
7
```

Algoritmo: Algoritmo 2

```
1    int soma1 = 0;
2    for (int i=0; i<n; i++){
3      soma1 = soma1 + 1;
4    }
5    for (int j=0; j<n;j++){
6      soma1 = soma1 + j;
7    }
8
```

Algoritmo: Algoritmo 3

```
1    int soma = 0;
2    for (int i=0; i<n; i++){
3      for (int j=0; j<m; j++){
4        soma = soma + 1;
5      }
6    }
7
```

Algoritmo: Algoritmo 4

```
1     int menor = INT_MAX;
2     for (int i=0; i<n; i++){
3       if (vetor[i] < menor)
4       menor = vetor[i];
5     }
6
```

Algoritmo: Algoritmo 5

Atribui limites superiores, exatos e inferiores para o tempo de execução de um algorítimo expressando os mesmos em forma de funções assintóticas a esse tempo de execução.

Para fazer essa análise usaremos código real e/ou pseudo-código: **Cuidado com as chamadas informais**.

Se assume que o acesso a RAM e as operações primitivas na CPU também tenham um tempo constante.

- o - Limite assintótico superior de ordem superior.
- O - Limite assintótico superior.
- θ - Limite assintótico restrito.
- Ω - Limite assintótico inferior.
- ω - Limite assintótico inferior de ordem inferior.

Se um algorítimo **não** é O então também não será o
Se o algorítimo é O e Ω o mesmo não será Θ.
Se o algorítimo é Θ então não poderá ser ω ou o.
Se o algorítimo é O e **não** é Ω então não poderá ser Θ.
Se é o ou ω então **não** é Θ.

Limites assintóticos

Mas... espera um pouco...

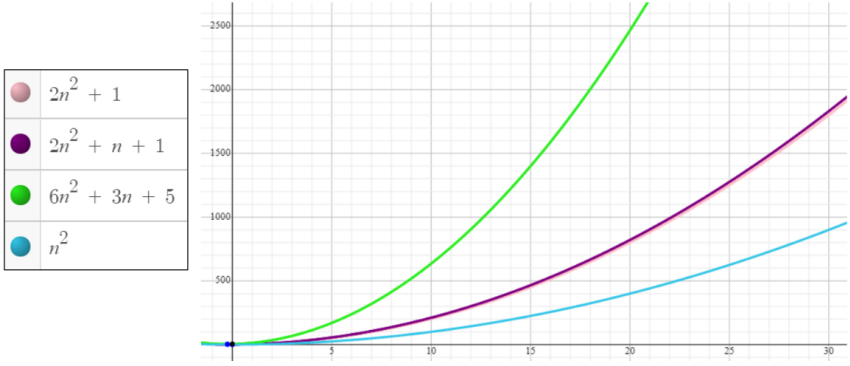


Figura: Assíntotas de n^2

Como pode $O(n^2)$ ser o limite assintótico superior se $n^2 \leq 6n^2 + 3n + 5$?

unesp

Na notação "Ózão" (ou *big O*) se diz que uma função $f(x)$ domina assintoticamente uma função $g(n)$ quando existem constantes c e a tais que, para **qualquer** $n \geq a$ é verdadeiro que $c.f(n) \geq g(n)$.

De outra forma podemos dizer que:

- $g(n) = O(f(n))$
- $g(n)$ é da ordem no máximo $f(n)$
- $g(n)$ é O de $f(n)$
- $g(n)$ é igual a O de $f(n)$
- $g(n)$ pertence a O de $f(n)$

Ver o exemplo *ordensDeComportamento.ipynb* no jupyter-lab.

Análise de algoritmos

Tempo de execução *versus* complexidade

Considerando a tabela abaixo vamos estimar o tempo **assintótico** de execução doos algoritmos representados.

Função de custo	Tamanho da entrada					
	10	20	30	40	50	60
$n.\log n$	33	43	49	53	56	59
n						
n^2						
n^3						
n^5						
2^n						
3^n						
$n!$						

unesp

Notação "Ózão" (ou *big O*)

Na notação "Ózão" (ou *big O*) se diz que uma função $f(x)$ domina assintoticamente uma função $g(n)$ quando existem constantes c e a tais que, para **qualquer** $n \geq a$ é verdadeiro que $c.f(n) \geq g(n)$.

De outra forma podemos dizer que:

- $g(n) = O(f(n))$
- $g(n)$ é da ordem no máximo $f(n)$
- $g(n)$ é O de $f(n)$
- $g(n)$ é igual a O de $f(n)$
- $g(n)$ pertence a O de $f(n)$
- o tempo de execução $T(n)$ é O

Ver o exemplo *ordensDeComportamento.ipynb* no jupyter-lab.

unesp

Notação "Ózão" (ou *big O*)

Sendo assim se pode dizer, **por exemplo**, que:

- $n = O(n^2)$
- $n^2 = O(n^2)$
- $\log n = O(n^2)$
- $n.\log n = O(n^2)$

Já que $g(n) \leq c.n^2$ para um certo valor de $c \geq a$.

unesp

Notação "Ózão" (ou *big O*)

Existe um valor de c para que $n^2 + 100n = O(n^2)$?

Por exemplo, $c = 100$ ou $c = 150$ ou $c \geq 100$.

$$n^2 + 100n \leq c.n^2 \implies n^2 + 100n \leq 100.n^2 \implies \frac{100}{n} \leq 99 \qquad \forall n \geq 2 \qquad (2)$$

unesp

Notação "Ózão" (ou *big O*)

Algebra

- $f(n) = O(f(n))$
- $c.O(f(n)) = O(f(n))$, $c = constante$
- $O(f(n)) + O(f(n)) = O(f(n))$
- $O(O(f(n))) = O(f(n))$
- $O(f(n)) + O(g(n)) = O(max(f(n), g(n)))$
- $O(f(n)).O(g(n)) = O(f(n).g(n))$
- $f(n).O(g(n)) = O(f(n).g(n))$

unesp

Notação "Ózão" (ou *big O*)

Treinando

1. $6n^4 + 12n^3 + 12$ é $O(n^4), O(n^5), O(n^6)$?
2. $3n^2 + 12n.\log n$ é $O(n^2), O(n^5), O(n)$?
3. $5n^2 + n.\log n + 12$ é $O(n^2), O(n^5)$?
4. $\log n + 4$ é $O(\log n), O(n)$?

Notação Ômega (ou Ω)

Na notação "Ômega" (ou Ω) se diz que uma função $f(x)$ **é dominada** assintoticamente por uma função $g(n)$ quando existem constantes c e a tais que, para **qualquer** $n \geq a$ é verdadeiro que $c.f(n) \leq g(n)$.

De outra forma podemos dizer que:

- $g(n) = \Omega(f(n))$
- $g(n)$ é da ordem no mínimo $f(n)$
- $g(n)$ é Ω de $f(n)$
- $g(n)$ é igual a Ω de $f(n)$
- $g(n)$ pertence a Ω de $f(n)$
- o tempo de execução $T(n)$ é Ω

Ver o exemplo *ordensDeComportamento.ipynb* no jupyter-lab.

unesp

Notação Ômega (ou Ω)

Sendo assim se pode dizer, **por exemplo**, que:

- $n^2 = \Omega(\log n)$
- $n = \Omega(\log n)$
- $n^3 = \Omega(\log n)$
- $n! = \Omega(\log n)$

Já que $g(n) \geq c.\log n$ para um certo valor de $c \geq a$.

Notação Ômega (ou Ω)

Algebra

- $f(n) = \Omega(f(n))$
- $c.\Omega(f(n)) = \Omega(f(n))$, $c = constante$
- $\Omega(f(n)) + \Omega(f(n)) = \Omega(f(n))$
- $\Omega(\Omega(f(n))) = \Omega(f(n))$
- $\Omega(f(n)) + \Omega(g(n)) = \Omega(max(f(n), g(n)))$
- $\Omega(f(n)).\Omega(g(n)) = \Omega(f(n).g(n))$
- $f(n).\Omega(g(n)) = \Omega(f(n).g(n))$

unesp

Notação Ômega (ou Ω)

Observações

Na prática a notação Ω não é vista sozinha em análises de algoritmos, pois é muito otimista.

unesp

Notação "ozinho" (ou o)

Na notação "ozinho" (ou o) se diz que uma função $f(x)$ domina assintoticamente uma função $g(n)$ quando existem constantes c e a tais que, para **qualquer** $n \geq a$ é verdadeiro que $c.f(n) \leq g(n)$, além disso, $f(n)$ **é de ordem superior** a $g(n)$.

De outra forma podemos dizer que:

- $g(n) = o(f(n))$
- $g(n)$ é de ordem menor que $f(n)$
- $g(n)$ é o de $f(n)$
- $g(n)$ é igual a o de $f(n)$
- $g(n)$ pertence a o de $f(n)$
- o tempo de execução $T(n)$ é menor que o

Ver o exemplo *ordensDeComportamento.ipynb* no jupyter-lab.

unesp

Notação "omeguinha" (ou ω)

Na notação "omeguinha" (ou ω) se diz que uma função $f(x)$ **é dominada** assintoticamente por uma função $g(n)$ quando existem constantes c e a tais que, para **qualquer** $n \geq a$ é verdadeiro que $c.f(n) \leq g(n)$, além disso, $f(n)$ **é de ordem inferior** a $g(n)$.

.

De outra forma podemos dizer que:

- $g(n) = \omega(f(n))$
- $g(n)$ é de ordem menor que $f(n)$
- $g(n)$ é ω de $f(n)$
- $g(n)$ é igual a ω de $f(n)$
- $g(n)$ pertence a ω de $f(n)$
- o tempo de execução $T(n)$ é menor que ω

Ver o exemplo *ordensDeComportamento.ipynb* no jupyter-lab.

unesp

Na prática a notação ω é o não são usadas sozinhas (isso quando são vistas) em análises de algoritmos, pois são muito imprecisas.

Notação "Theta" (ou θ)

Na notação "Theta" (ou θ) se diz que uma função $f(x)$ **restringe** a função $g(n)$ quando existem constantes c_1, c_2 e a tais que, para **qualquer** $n \geq a$ é verdadeiro que
$0 \leq c_1.g(n) \leq f(n) \leq c_2.g(n) \qquad \forall n \geq a.$

.

De outra forma podemos dizer que:

- $g(n) = \theta(f(n))$
- $g(n)$ é de ordem $f(n)$
- $g(n)$ é θ de $f(n)$
- $g(n)$ é igual a θ de $f(n)$
- $g(n)$ pertence a θ de $f(n)$
- o tempo de execução $T(n)$ é menor que θ

Ver o exemplo *ordensDeComportamento.ipynb* no jupyter-lab.

unesp

Recorrências

Recorrências

Figura

unesp

Recorrências

A recorrência é uma forma de descrever o comportamento de algoritmos recursivos.

Para esse tipo de análise usaremos, entre outras coisas, o que se chama de **relação de recorrência**.
Toda recursão tem um **caso base** ou **ponto de parada**. O caso base é o conjunto de instruções que fará com que um valor trivial seja finalmente retornado (ou não) iniciando o desempilhamento da pilha de recursão. Além disso deve existir o **caso indutivo** ou **recursão** que é o trecho de código onde a função faz uma chamada a ela mesma empilhando a chamada anterior e suas respectivas variáveis/estados.

```
1  int fatorial(int n)
2  {
3    int resp = 0;
4    if(n == 0){ // <-- Caso base
5      resp = 1; // <-- Caso base
6    }else{
7      resp = n * fatorial(n-1); // <-- Caso indutivo
8    }
9    return resp;
10 }
```

unesp

Qual a relação de recorrência desse algoritmo?

```
1  int fatorial(int n)
2  {
3    int resp = 0;
4    if(n == 0){ // <-- Caso base
5      resp = 1; // <-- Caso base
6    }else{
7      resp = n * fatorial(n-1); // <-- Caso indutivo
8    }
9    return resp;
10 }
```

Do **único** caso indutivo é possível perceber que a entrada é diminuída em **n-1**, de todo o resto do algoritmo percebe-se que os tempos de execução são constantes, ou seja, $\Theta(1)$. Então a relação de recorrência é dada por esses valores como colocado na equação 2.

$$T(n) = 1.T(n-1) + \Theta(1) \tag{2}$$

Qual a relação de recorrência desse algoritmo?

```
1 int fibonacci(int n){
2   int res = 0;
3   if(n == 1 || n == 2){ // <-- caso base
4     res = 1; // <-- caso base
5   }else{
6     res = fibonacci(n-1) + fibonacci(n-2); // <-- casos indutivos
7   }
8   return res;
9 }
```

Dos **dois** casos indutivos é possível perceber que a entrada é diminuída em **n-1** e **n-2**, de todo o resto do algoritmo percebe-se que os tempos de execução são constantes, ou seja, $\Theta(1)$.
Então a relação de recorrência é dada por esses valores como colocado na equação 3.

$$T(n) = 1.T(n-1) + 1.T(n-2) + \Theta(1) \quad (3)$$

Qual a relação de recorrência desse algoritmo?

```
1  int uga(int n){
2    int res = 0;
3    if(n == ){ // <-- caso base
4      res = 1; // <-- caso base
5    }else{
6      res = uga(n/2) + uga(n/2); // <-- casos indutivos
7    }
8    for(int i = 0; i < n; i++){
9      int z = res + 1;
10   }
11   return res;
12 }
```

Dos **dois** casos indutivos é possível perceber que a entrada é diminuída em **n/2** e **n-2**, de todo o resto do algoritmo percebe-se que os tempos de execução são **constantes + complexidade n**, ou seja, $\Theta(n)$.
Então a relação de recorrência é dada por esses valores como colocado na equação 4.

$$T(n) = T(n/2) + T(n/2) + \Theta(n) + \Theta(1) \rightarrow 2.T(n/2) + \Theta(n) \quad (4)$$

Resolvendo relações de recorrência

- método da árvore de recursão
- método iterativo
- método mestre

unesp

$$\sum_{k=1}^{n} a_k = a_1 + a_2 + \cdots + a_k$$

$$\sum_{k=1}^{\infty} a_k = \lim_{n\to\infty} \sum_{k=1}^{n} a_k$$

$$\sum_{k=1}^{n} (c.a_k + b_k) = c. \sum_{k=1}^{n} a_k + \sum_{k=1}^{n} b_k \qquad (5)$$

$$\sum_{k=1}^{n} \Theta(f(k)) = \Theta\left(\sum_{k=1}^{n} f(k)\right)$$

$P.A.$

$$a_n = a_1 + (n-1).r$$

$$S_n = \frac{n.(a_1 + a_n)}{2}$$

$P.G.$ (6)

$$a_n = a_1.q^{n-1}$$

$$S_n = \frac{a_1.(q^n - 1)}{q - 1} \qquad \forall q \geq |1|$$

$$S_n = \frac{a_1}{1 - q} \qquad \forall q < |1|$$

unesp

$$\log_c a = k \Leftrightarrow c_k = a$$
$$\log_c(ab) = \log_c a + \log_c b$$
$$\log_b a^n = n.log_b a$$
$$\log_a a = 1$$
$$\log n = \lg n = \log_2 n \qquad (7)$$
$$a = b^{\log_b a}$$
$$a^{\log_b n} = n^{\log_b a}$$
$$\log_a b = \frac{\log_c b}{\log_c a}$$

Figura: Ahhhhhhh!

unesp

- expandir a árvore
- calcular a altura **h**

Para entender como expandir a árvore de recursão, ao lado coloquei um exemplo simples para a seguinte relação de recorrência:

$$T(n) = T(n-1) + \Theta(1) \qquad (8)$$

Ao fazer a expansão se chega no caso-limite em que o tamanho da entrada é igual a 1 ($T(1)$). Dessa forma a altura $\mathbf{h} = \mathbf{\Theta(n)}$ já que $h = n - 1 \implies \Theta(n)$

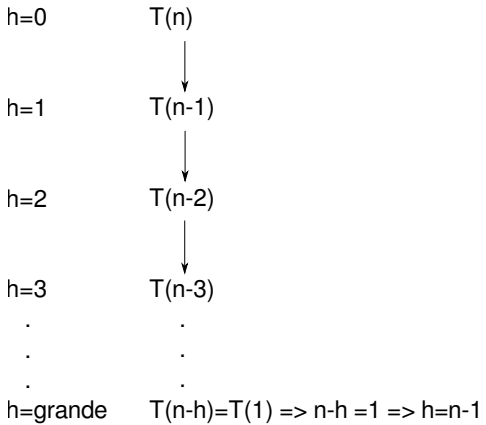


Figura: Árvore de recursão expandida

- calcular os custos por nível
- somatória dos custos dos níveis **h** vezes

Nesta fase devemos nos concentrar no **tempo gasto para combinar os resultados das chamadas recursivas**.

Esse tempo é dado pelo último termo da relação de recorrência.

Então:

$$T(n) \to T(n-1) \to T(n-2) \to \cdots \to T(n-h) \implies$$
$$\Theta(1) + \Theta(1) + \Theta(1) + \cdots + \Theta(1) = \mathbf{\Theta(1)} \implies$$
$$h.\Theta(1) = \Theta(n).\Theta(1) = \mathbf{\Theta(n)} \tag{9}$$

Expansão da árvore ao lado.

$$T(n) = 2T(n/2) + \Theta(n) \qquad (10)$$

Analisando a árvore de recursão nota-se que quando a profundidade da árvore atinge seu máximo a altura da mesma é expressa segundo a expressão da equação.

$$\begin{aligned} & T(n/2^h) = T(1) \implies \\ & \frac{n}{2^h} = 1 \implies \\ & 2^h = n \implies \\ & \mathbf{h} = \log \mathbf{n} \end{aligned} \qquad (11)$$

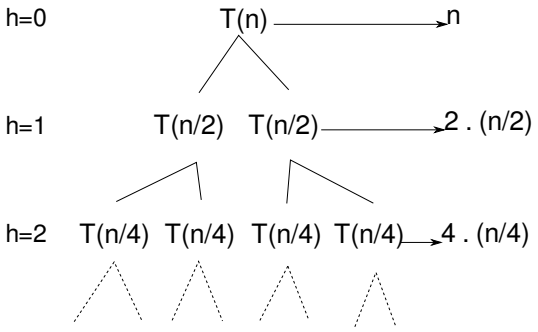


Figura: Árvore de recursão expandida

Nesta fase devemos nos concentrar no **tempo gasto para combinar os resultados das chamadas recursivas**.

Esse tempo é dado pelo último termo da relação de recorrência.

Então:

$$\Theta(n) + \Theta(2.n/2) + \Theta(4.n/4) + \cdots + \Theta(2^h.n/2^h) = \Theta(n) + \Theta(n) + \Theta(n) + \cdots + \Theta(n) = \mathbf{\Theta(n)} \implies h.\Theta(n) = \Theta(\log n).\Theta(n) = \mathbf{\Theta(n.\log n)} \tag{12}$$

Dadas as relações de recorrência, para cada uma, crie a árvore recursiva expandida e determine o tempo de execução do algoritmo.

$$T(n) = 3.T(n/3) + \Theta(n) \tag{13}$$

$$T(n) = 2.T(n/2) + \Theta(1) \tag{14}$$

$$T(n) = 2.T(n/3) + \Theta(1) \tag{15}$$

$$T(n) = \Theta(n.\log n) \tag{16}$$

$$T(n) = \Theta(n) \tag{17}$$

$$T(n) = \Theta(n^{\log_3 2}) \tag{18}$$

No segundo e terceiros exercícios a dica é notar a progressão geométrica de **h** termos e razão 2 já que, como já foi dito, é necessário **somar h vezes** os custos.

Resolvendo relações de recorrência - Método da árvore de recursão

Exercício 1

```
1  void pesquisa(std::vector<int> vetor, int valor)
2  {
3    if (vetor.size() <= 1)
4    {
5      std::cout << "Fim da lista" << std::endl;
6    }else{
7      for(int i = 0; i < vector.size(); i++)
8      {
9        if(vector[i] == valor){
10         std::cout << "Valor encontrado em " << i << std::endl;
11         return;
12       }
13     }
14     pesquisa(vector.slice(0, vector.size()/2));
15   }
16 }
```

Qual a complexidade desse algoritmo?

unesp

Resolvendo relações de recorrência - Método iterativo

Dada a relação de recorrência

$$T(n) \leq T(n/2) + 1 \tag{19}$$

Considerando que **i** indica o número de **iterações** vamos calcular de forma **iterativa** o tempo deste algoritmo.

$$T(n) \leq T(n/2)+1 \leq T(n/2/2)+1+1 \leq T(n/2/2/2)+1+1+1 \leq \mathbf{T(n/2^i)} + \mathbf{i} \tag{20}$$

Sendo assim, para o caso base $T(1)$, note que o tempo do algoritmo **não** depende de **i**, portanto, i é tradada como constante.

$$T(n/2^i) + i = T(1) \implies T(n/2^i) = T(1) \implies n/2^i = 1 \implies 2^i = n \implies i = \log n \tag{21}$$

Substituindo **i** no resultado da equação 20.

$$T(n/2^i) + i \implies T(n/2^{\log n}) + \log n \implies T(1) + \log n \implies T(n) = \mathbf{O(\log n)} \tag{22}$$

É uma fórmula geral para calcular o tempo de um algoritmo mas tem algumas limitações em relação aos outros métodos já descritos:

- usado para resolver problemas do tipo "dividir para conquistar"
- $a =$ é o número de chamadas recursivas simultâneas
- $b =$ é a quantidade de subdivisões do problema
- $f(n)$ deve ser não negativa

Considerando essas limitações a fórmula é dada na equação 23.

$$T(n) = a.T(n/b) + f(n) \tag{23}$$

Aplicada a fórmula, a avaliação é feita segundo as três condições abaixo:

- $f(n) = O(n^{\log_b a - \epsilon}) \exists \epsilon > 0 \implies T(n) = \Theta(n^{\log_b a})$
- $f(n) = \Theta(n^{\log_b a}) \implies T(n) = \Theta(n^{\log_b a}. \log_b n) = \Theta(f(n). \log_b n)$
- $f(n) = \Omega(n^{\log_b a + \epsilon}) \exists \epsilon > 0, a.f(n/b) \leq c.f(n) \forall 0 < c < 1, n = grande \implies T(n) = \Theta(f(n))$

É importante saber que o teorema mestre não necessáriamente descreve o tempo do algoritmo de forma exata: Os resultados obtidos devem ser interpretados como se referindo aos **casos médios** ou **piores casos** a depender da estrutura interna do algoritmo (instruções, desvios condicionais, etc.). Leve em consideração a possibilidade de um melhor ou pior caso que resulte em tempos excepcionais.

Em quais dos casos abaixo se pode aplicar o teorema mestre?

- $T(n) = 2^n . T(n/2) + n^n$
- $T(n) = 0,5 . T(n/2) + 1/n$
- $T(n) = 64 . T(n/2) - n^2 . \log n$

Em quais dos casos abaixo se pode aplicar o teorema mestre?

- $T(n) = 2^n . T(n/2) + n^n$
- $T(n) = 0,5 . T(n/2) + 1/n$
- $T(n) = 64 . T(n/2) - n^2 . \log n$

Resposta:

Em quais dos casos abaixo se pode aplicar o teorema mestre?

- $T(n) = 2^n . T(n/2) + n^n$
- $T(n) = 0,5 . T(n/2) + 1/n$
- $T(n) = 64 . T(n/2) - n^2 . \log n$

Resposta: **Nenhum**:
Na primeiro caso o valor 2^n não é uma constante inteira, no segundo a mesma coisa, no terceiro $f(n) = -n^2 . \log n$

Vamos resolver a relação de recorrência:

$$T(n) = 2.T(n/2) + \Theta(1) \tag{24}$$

- $a = 2$
- $b = 2$
- $f(n) = 1 = constante$

$$1 < n^{\log_b a} \implies 1 < n^{\log_2 2} \implies 1 < n^1 \implies verdadeiro! \implies T(n) = \Theta(n^{\log_2 2}) = \mathbf{\Theta(n)} \tag{25}$$

Vamos resolver a relação de recorrência:

$$T(n) = 2.T(n/2) + \Theta(n) \tag{26}$$

- $a = 2$
- $b = 2$
- $f(n) = n$

$$\begin{aligned} & n = n^{\log_b a} \implies \\ & n = n^{\log_2 2} \implies \\ & n = n^1 \implies \textit{verdadeiro}! \implies \\ & T(n) = \Theta(f(n).\log_b n) = \mathbf{n}.\log \mathbf{n} \end{aligned} \tag{27}$$

Vamos resolver a relação de recorrência:

$$T(n) = 2.T(n/2) + \Theta(n^2) \tag{28}$$

- $a = 2$
- $b = 2$
- $f(n) = n^2$

$$\begin{aligned} & n^2 > n^{\log_b a} \implies \\ & n^2 > n^{\log_2 2} \implies \\ & n^2 > n^1 \implies \textit{verdadeiro}! \implies \\ & T(n) = \Theta(f(n)) = \mathbf{\Theta(n^2)} \end{aligned} \tag{29}$$

Vamos resolver a relação de recorrência:

$$T(n) = 9.T(n/3) + n \tag{30}$$

$$T(n) = 2.T(n/4) + \sqrt{n} \tag{31}$$

$$T(n) = 3.T(n/4) + n.\log n \tag{32}$$

Resolvendo a recorrência 30
Verdadeiro para o primeiro caso:

$$
\begin{aligned}
&T(n) = 9.T(n/3) + n \implies \\
&a = 9, b = 3, f(n) = n \implies \\
&f(n) = O(n^{\log_3 9 - \epsilon}) = \\
&O(n^{2-\epsilon}) \implies \\
&n \leq c.n^{2-\epsilon}, \epsilon = 1 \therefore \\
&T(n) = \Theta(n^{\log_b a}) = \\
&\Theta(n^{\log_3 9}) = \Theta(n^2)
\end{aligned}
\tag{33}
$$

Resolvendo a recorrência 31
Verdadeiro para o segundo caso:

$$
\begin{aligned}
&T(n) = 2.T(n/4) + \sqrt{n} \implies \\
&a = 2, b = 4, f(n) = \sqrt{n} \implies \\
&f(n) = \Theta(n^{\log_4 2}) \implies \\
&\sqrt{n} = \Theta(n^{0,5}) \implies \\
&c_1.n^{0,5} \leq \sqrt{n} \leq c_2.n^{0,5}, c_1 = c_2 = 1 \therefore \\
&T(n) = \Theta(n^{\log_b a}. \log n) = \Theta(n^{\log_4 2}. \log n) = \\
&\Theta(n^{0,5}. \log n) = \Theta(\sqrt{n}. \log_4 n)
\end{aligned}
\tag{33}
$$

Resolvendo a recorrência 32; verdadeiro para o terceiro caso:

$T(n) = 3.T(n/4) + n.\log n \implies$

$a = 3, b = 4, f(n) = n.\log n \implies$

$f(n) = \Omega(n^{\log_b a+\epsilon}) \implies n.\log n = \Omega(n^{\log_4 3+\epsilon}) \implies$

$n.\log n = \Omega(n^{0.79+\epsilon}) \implies n.\log n = \Omega(n^{0.79+0,21}), \epsilon = 0,21 \implies$

$n.\log n = c.n^1 \implies n.\log n = \Omega(n)! \therefore$

$a.f(n/b) \leq c.f(n)? \implies 3.\frac{n}{4}.\log\left(\frac{n}{4}\right) \leq c.n.\log n \implies \frac{3n}{4}.(\log n - \log 4) \leq c.n.\log n \implies$

$\frac{3n}{4}.(\log n - 2) \leq \frac{3n}{4}.\log n, c = \frac{3}{4} \implies \log n - 2 \leq \log n \therefore T(n) = \Theta(f(n)) = \Theta(n.\log n).$

Tudo em Todo Lugar ao Mesmo Tempo

Exercício

Determine a relação de recorrência do algoritmo abaixo para em seguida determinar seu tempo usando o método da **árvore de recursão**, **iterativo** e **mestre**.

```
/**
 * Busca binaria
 * x - Valor procurado
 * vetor - Vetor onde procurar
 * esq - Limite inferior do vetor
 * dir - Limite superior do vetor
 * return -1 se item nao encontrado ou a localizacao do item
 */
int binSearch(int x, std::vector<int> vetor, int esq, int dir)
{
  int meio = (esq + dir) / 2;

  if(esq > dir){
    return -1; // Item nao encontrado
  }

  if(x == vetor[meio]) return meio;

  if(x < vetor[meio])
    return binSearch(x, vetor, esq, meio-1);
  else
    return binSearch(x, vetor, meio+1, dir);
}
```

inesp

Algoritmos de ordenação - baseados em troca

unesp

Analisar algoritmos de ordenação são uma ótima forma de entender e analisar vários outros algoritmos devido a natureza dos problemas, além, é claro, de serem diretamente utilizados em aplicações como banco de dados e localização de padrões.

Os algoritmos que teremos contato a partir de agora serão mais complexos de se analisar pois, nem sempre coisas como a relação de recorrência estarão trivialmente definidos.

Algorítimos de ordenação - baseados em troca

- **Burbble-sort** - método do borbulhação
- **Quick-sort** - ordenação por troca de partição

unesp

Algorítimos de ordenação - baseados em troca

Burbble-sort

Esse algoritmo percorre várias vezes o vetor de dados a cada iteração comparando cada um de seus elementos com o próximo, sendo que, se o próximo é menor do que o anterior há uma troca de lugares entre os dois.

```
1 #include <iostream>
2 #include <vector>
3
4 void burbbleSort(int vetor[], const int tamanho){
5     int temp;
6     for (int j = 0; j < tamanho-1; ++j) {
7         for (int i = 0; i < tamanho-j-1; ++i) {
8             if(vetor[i] > vetor[i+1]){
9                 temp = vetor[i];
10                vetor[i] = vetor[i+1];
11                vetor[i+1] = temp;
12            }
13        }
14    }
15 }
16 int main()
17 {
18     std::vector<int> teste = {20,45,63,10,2,5,9,5,90,101,104,1004};
19     burbbleSort(teste);
20     for(auto item : teste) std::cout << item<< std::endl;
21 }
```

Determine o tempo de execução do *burbble-sort* com seus respectivos melhores e piores tempos se houverem. Use a notação que melhor se encaixa à situação.

Determine o tempo de execução do *burbble-sort* com seus respectivos melhores e piores tempos se houverem. Use a notação que melhor se encaixa à situação.
Resposta: $O(n^2)$.

A ideia usada é **dividir para conquistar**, o vetor é dividido em outros dois menores que são ordenados independentemente para depois serem combinados produzindo assim o resultado final. Tal separação é feita usando-se um **elemento pivô**, dessa forma, todos os elementos a esquerda do pivô são menores do que ele e todos elementos a direita são maiores que o mesmo

Primeiro passo:

- determinar o elemento pivô: Isso pode ser feito de várias formas, localizar a mediana, fazer uma média entre elementos, etc.
- ordenar os sub-vetores de forma que os elementos a esquerda do pivô são menores que o mesmo e os elementos a direita são maiores.
- percorrer simultaneamente o vetor da esquerda para direita, e da direita para esquerda comparando os respectivos elementos e trocando de posição quando necessário.
- quando os ponteiros da esquerda e da direita se cruzarem a troca é finalizada.

Segundo passo
Ordenar os sub-vetores abaixo e acima do elemento pivô.
Vamos escolher como pivô o elemento 25.
Agora caminhamos com o ponteiro **menor** para a direita até que encontremos um valor **maior que o pivô**.
Simultaneamente caminhamos com o ponteiro **maior** para a esquerda até que encontremos um valor **menor que o pivô**.

25 (menor →), 57, 48, 37, 12, 86, 92, **33** (← maior)

25, **57** (menor →), 48, 37, **12** (← maior) 86, 92, 33

$25,\ \underset{\substack{\uparrow\\ menor\rightarrow}}{\mathbf{57}}\ ,48,37,\ \underset{\substack{\uparrow\\ \leftarrow maior}}{\mathbf{12}}\ 86,92,33$

Permuta-se os elementos apontados por **menor** e **maior**:

$25,\ \underset{\substack{\uparrow\\ menor\rightarrow}}{\mathbf{12}}\ ,48,37,\ \underset{\substack{\uparrow\\ \leftarrow maior}}{\mathbf{57}}\ 86,92,33$

Até que os ponteiros maior e menor **se cruzem**, assim, o ponteiro **maior** indica onde se permutar o pivô pelo valor apontado.

$25,\ \underset{\substack{\uparrow\\ \leftarrow maior}}{\mathbf{12}}\ ,\ \underset{\substack{\uparrow\\ menor\rightarrow}}{\mathbf{48}}\ ,37,57,86,92,33$

Até que os ponteiros maior e menor **se cruzem**, assim, o ponteiro **maior** indica onde se permutar o pivô pelo valor apontado.

$25, \underset{\substack{\uparrow \\ \leftarrow maior}}{\mathbf{12}}, \underset{\substack{\uparrow \\ menor \rightarrow}}{\mathbf{48}}, 37, 57, 86, 92, 33$

$12, \underset{\substack{\uparrow \\ \leftarrow maior}}{\mathbf{25}}, \underset{\substack{\uparrow \\ menor \rightarrow}}{\mathbf{48}}, 37, 57, 86, 92, 33$

Agora **divide-se** o vetor em dois pedaços segundo a **localização do pivô**:

$\underset{\substack{\uparrow \\ sub-vetor0}}{\{12\}} \quad \underset{\substack{\uparrow \\ pivo}}{(25)} \quad \underset{\substack{\uparrow \\ sub-vetor1}}{\{48, 37, 57, 86, 92, 33\}}$

Agora reinicia-se o processo **para cada sub-vetor**, caso um vetor vetor chegue no **caso base** contendo 1 elemento o consideramos ordenado, no caso de 2 elementos é possivel, usando uma comparação simples, permutar os elementos, se necessário.

unesp

Implemente o *quick-sort*.
Determine o tempo de execução do *Quick-sort* com seus respectivos melhores e piores tempos se houverem. Use a notação que melhor se encaixa à situação.

Algorítimos de ordenação - baseados em troca

Exercício 1

Implemente o *quick-sort*.

Determine o tempo de execução do *Quick-sort* com seus respectivos melhores e piores tempos se houverem. Use a notação que melhor se encaixa à situação.

Resposta: No melhor caso as partiçoes são quebradas ao meio, no pior (vetor ordenado) o pivô quebra o vetor no seu início. $\Omega(n.\log n), O(n^2)$.

```
1 int particionar(int vetor[], int menor, int maior, int valorDoPivo){
2   int indiceDoPivo = menor;
3   for(int i=menor;i<=maior;i++) {
4     if(vetor[i]<=valorDoPivo) {
5       std::swap(vetor[indiceDoPivo],vetor[i]);
6       indiceDoPivo++;
7     }
8   }
9   indiceDoPivo--;
10  return indiceDoPivo;
11 }
12 void quickSort(int vetor[], int menor, int maior){
13   if(menor < maior) {
14     int valorDoPivo = vetor[maior];
15     int indiceDoPivo = particionar(vetor, menor, maior, valorDoPivo);
16     quickSort(vetor, menor, indiceDoPivo-1);
17     quickSort(vetor, indiceDoPivo+1, maior);
18   }
19 }
```

inesp

Algoritmos de ordenação - baseados em inserção

unesp

Algoritmos de ordenação - baseados em inserção

Os algoritmos por inserção trabalham de forma análoga a um conjunto de cartas de baralho, essas que são seguradas na mão, a princípio, estão desordenadas, o papel do algoritmo é retirar uma ou mais cartas de suas posições originais e colocar-las na posição correta de forma a configurar uma certa ordem.

$$10, 25, 40, 50, \underset{\substack{\uparrow \\ foraDeOrdem}}{\mathbf{12}}, 80, 2, 23, \ldots$$

$$10, \underset{\substack{\uparrow \\ emOrdem}}{\mathbf{12}}, 25, 40, 50, 80, 2, 23, \ldots$$

Perceba que o número 12 foi **inserido** entre 10 e 25.

unesp

Algoritmos de ordenação - baseados em inserção

- **Inserção simples**
- **Shell-sort**

unesp

Algoritmos de ordenação - baseados em inserção

Insertion-Sort

Algoritmo **Insertion-Sort** segue exatamente a ideia demonstrada no início desta seção, é um algoritmo que retira valores de um local e os coloca em outro deslocando, se necessário, todos os outros elementos do vetor.

```
1  void insertionSort(int vetor[], int tamanho) {
2    for (int i = 1; i < tamanho; i++) {
3      int key = vetor[i];
4      int j = i - 1;
5      while (key < vetor[j] && j >= 0) {
6        vetor[j + 1] = vetor[j];
7        --j;
8      }
9      vetor[j + 1] = key;
10   }
11 }
```

unesp

Determine o tempo de execução do *algoritmo de inserção simples* com seus respectivos melhores e piores tempos se houverem. Use a notação que melhor se encaixa à situação.

Determine o tempo de execução do *algoritmo de inserção simples* com seus respectivos melhores e piores tempos se houverem. Use a notação que melhor se encaixa à situação.
Resposta:
$\Omega(n)$ para o vetor ordenado, $O(n^2)$ para o vetor ordenado de forma decrescente.

O Shell-Sort é uma evolução do algoritmo anterior, avaliando a inserção simples nota-se que o melhor caso é quando o vetor já está ordenado, sendo assim, o Shell-Sort adota a estratégia de **dividir para conquistar** subdividindo o vetor em vários pares de valores usando "saltos"que vão diminuindo em quantidade com o tempo de forma que fica cada vez mais **provável** que o vetor já esteja ordenado, diminuindo assim o tempo total de execução.

No exemplo abaixo, mostro um exemplo com passo $k = 5$:

55	33	**12**	01	**19**	10	**70**	25
↑	↑	↑	↑	↑	↑	↑	↑
par0	*par1*	*par2*	*par3*	*par0*	*par1*	*par2*	*par3*

$$\underset{par0}{55}, \underset{par1}{33}, \underset{par2}{12}, \underset{par3}{01}, \underset{par0}{19}, \underset{par1}{10}, \underset{par2}{70}, \underset{par3}{25}$$

$$\underset{par0}{19}, \underset{par1}{10}, \underset{par2}{12}, \underset{par3}{01}, \underset{par0}{55}, \underset{par1}{33}, \underset{par2}{70}, \underset{par3}{25} \leftarrow \text{Primeira passada } k = 5$$

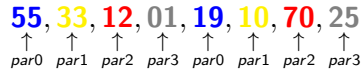
55, **33**, **12**, **01**, **19**, **10**, **70**, **25**
↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑
par0 *par1* *par2* *par3* *par0* *par1* *par2* *par3*

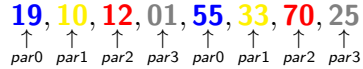
19, **10**, **12**, **01**, **55**, **33**, **70**, **25** ← Primeira passada $k = 5$
↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑
par0 *par1* *par2* *par3* *par0* *par1* *par2* *par3*

19, **10**, **12**, **01**, **55**, **33**, **70**, **25**
↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑
par0 *par1* *par2* *par0* *par1* *par2* *par0* *par1*

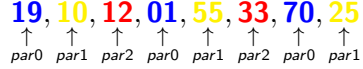
01, **10**, **12**, **19**, **25**, **33**, **70**, **55** ← Segunda passada $k = 4$
↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑
par0 *par1* *par2* *par0* *par1* *par2* *par0* *par1*

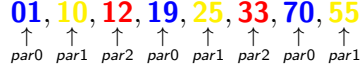
unesp



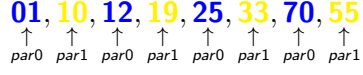


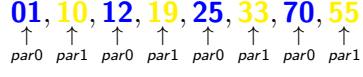
← Primeira passada $k = 5$





← Segunda passada $k = 4$





← Terceira passada $k = 2$

unesp

Quando $k = 1$ (1 partição = todo o vetor) aplica-se então o algoritmo de inserção simples!

```
1  void shellSort(int vetor[], int tamanho)
2  {
3    // ATENCAO: A escolha das particoes eh arbitraria
4    // existem diferentes tempos de algoritmo para diferentes
5    // tamanhos de particao.
6    for (int particao = tamanho/2; particao > 0; particao /= 2)
7    {
8      // Faz o insertion-sort na particao
9      for (int i = particao; i < tamanho; i += 1)
10     {
11       int temp = vetor[i];
12       int j;
13       for (j = i; j >= particao && vetor[j - particao] > temp; j -= particao)
14       vetor[j] = vetor[j - particao];
15       vetor[j] = temp;
16     }
17   }
18 }
```

Determine o tempo de execução do *Shell-Sort* com seus respectivos melhores e piores tempos se houverem. Use a notação que melhor se encaixa à situação. Como o tamanho das partições muda o tempo do algoritmo? **Indique as referências**.

Algoritmos de ordenação - baseados em seleção

unesp

Algoritmos de ordenação - baseados em seleção

A ideia é selecionar um elemento e já colocá-lo em sua posição correta definitiva evitando assim futuras permutações com o mesmo.

Algoritmos de ordenação - baseados em seleção

- **Selection-Sort**
- **Heap-Sort**

Algoritmos de ordenação - baseados em seleção

Selection-sort

É um algoritmo que seleciona o menor elemento do vetor e o coloca no início do mesmo, em seguida, seleciona o menor elemento do **vetor n-1** e o coloca no início desse vetor resultante. Esse algoritmo se repete até que o vetor esteja totalmente ordenado.

$55, 33, 12, 01, 19, 10, 70, 25$

$55, 33, 12, \underbrace{\mathbf{01}}_{\text{menor}}, 19, 10, 70, 25$

Algoritmos de ordenação - baseados em seleção

Selection-sort

É um algoritmo que seleciona o menor elemento do vetor e o coloca no início do mesmo, em seguida, seleciona o menor elemento do **vetor n-1** e o coloca no início desse vetor resultante. Esse algoritmo se repete até que o vetor esteja totalmente ordenado.

$55, 33, 12, 01, 19, 10, 70, 25$

$55, 33, 12, \underbrace{\mathbf{01}}_{\text{menor}}, 19, 10, 70, 25$

$$\underbrace{\underbrace{\mathbf{01}}_{\text{menor}}, 55, 33, 12, 55, 19, 10, 70, 25}$$

Algoritmos de ordenação - baseados em seleção

Selection-sort

É um algoritmo que seleciona o menor elemento do vetor e o coloca no início do mesmo, em seguida, seleciona o menor elemento do **vetor n-1** e o coloca no início desse vetor resultante. Esse algoritmo se repete até que o vetor esteja totalmente ordenado.

$55, 33, 12, 01, 19, 10, 70, 25$

$55, 33, 12, \underbrace{\mathbf{01}}_{\text{menor}}, 19, 10, 70, 25$

$\underbrace{\mathbf{01}, 55, 33, 12, 55, 19, 10, 70, 25}_{\text{menor}}$

$01, \underbrace{55, 33, 12, 55, 19, \mathbf{10}, 70, 25}_{\text{menor}}$

Algoritmos de ordenação - baseados em seleção

Selection-sort

É um algoritmo que seleciona o menor elemento do vetor e o coloca no início do mesmo, em seguida, seleciona o menor elemento do **vetor n-1** e o coloca no início desse vetor resultante. Esse algoritmo se repete até que o vetor esteja totalmente ordenado.

$55, 33, 12, 01, 19, 10, 70, 25$

$55, 33, 12, \underset{\text{menor}}{\mathbf{01}}, 19, 10, 70, 25$

$$\overbrace{\underset{\text{menor}}{\mathbf{01}}, 55, 33, 12, 55, 19, 10, 70, 25}$$

$$\overbrace{01, 55, 33, 12, 55, 19, \underset{\text{menor}}{\mathbf{10}}, 70, 25}$$

$$01, \overbrace{\underset{\text{menor}}{\mathbf{10}}, 33, 12, 55, 19, 55, 70, 25}$$

Algoritmos de ordenação - baseados em seleção

Selection-sort

$$01, \underbrace{\overset{\text{menor}}{\overset{\uparrow}{\mathbf{10}}}, 33, 12, 55, 19, 55, 70, 25}$$

Algoritmos de ordenação - baseados em seleção

Selection-sort

$$01, \underbrace{\mathbf{10}, 33, 12, 55, 19, 55, 70, 25}_{\text{menor}}$$

$$01, 10, \underbrace{33, \mathbf{12}, 55, 19, 55, 70, 25}_{\text{menor}}$$

Algoritmos de ordenação - baseados em seleção

Selection-sort

$$01, \underset{\uparrow}{\mathbf{10}}, 33, 12, 55, 19, 55, 70, 25$$

$$\overbrace{}^{\text{menor}}$$

$$01, 10, 33, \underset{\uparrow}{\mathbf{12}}, 55, 19, 55, 70, 25$$

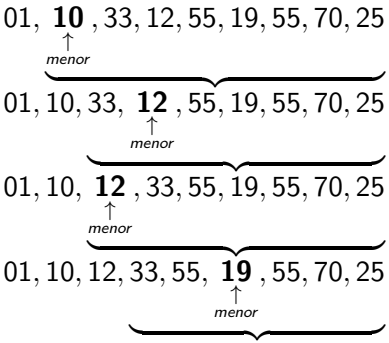
$$\overbrace{}^{\text{menor}}$$

$$01, 10, \underset{\uparrow}{\mathbf{12}}, 33, 55, 19, 55, 70, 25$$

$$\overbrace{}^{\text{menor}}$$

Algoritmos de ordenação - baseados em seleção

Selection-sort



Algoritmos de ordenação - baseados em seleção

Selection-sort

$$01, \underset{\uparrow}{\mathbf{10}}, 33, 12, 55, 19, 55, 70, 25$$
$$\text{menor}$$

$$01, 10, 33, \underset{\uparrow}{\mathbf{12}}, 55, 19, 55, 70, 25$$
$$\text{menor}$$

$$01, 10, \underset{\uparrow}{\mathbf{12}}, 33, 55, 19, 55, 70, 25$$
$$\text{menor}$$

$$01, 10, 12, 33, 55, \underset{\uparrow}{\mathbf{19}}, 55, 70, 25$$
$$\text{menor}$$

$$01, 10, 12, \underset{\uparrow}{\mathbf{19}}, 55, 33, 55, 70, 25$$
$$\text{menor}$$

$$01, 10, 12, 19, 55, 33, 55, 70, \underset{\uparrow}{\mathbf{25}}$$
$$\text{menor}$$

unesp

Algoritmos de ordenação - baseados em seleção

Selection-sort

$$01, \underset{\uparrow}{\mathbf{10}}, 33, 12, 55, 19, 55, 70, 25$$

menor

$$01, 10, 33, \underset{\uparrow}{\mathbf{12}}, 55, 19, 55, 70, 25$$

menor

$$01, 10, \underset{\uparrow}{\mathbf{12}}, 33, 55, 19, 55, 70, 25$$

menor

$$01, 10, 12, 33, 55, \underset{\uparrow}{\mathbf{19}}, 55, 70, 25$$

menor

$$01, 10, 12, \underset{\uparrow}{\mathbf{19}}, 55, 33, 55, 70, 25$$

menor

$$01, 10, 12, 19, 55, 33, 55, 70, \underset{\uparrow}{\mathbf{25}}$$

menor

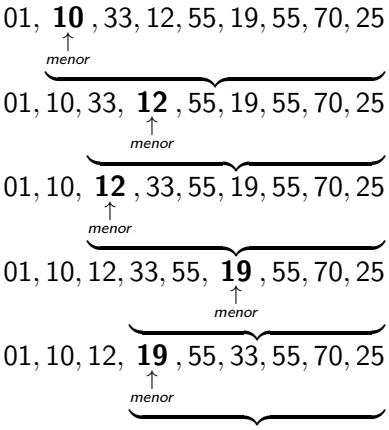
$$01, 10, 12, 19, \underset{\uparrow}{\mathbf{25}}, 33, 55, 70, 55$$

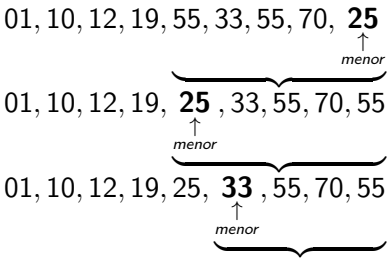
menor

unesp

Algoritmos de ordenação - baseados em seleção

Selection-sort

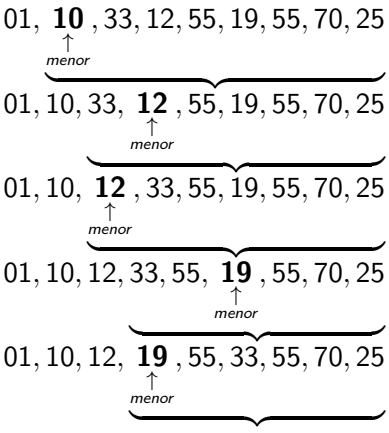


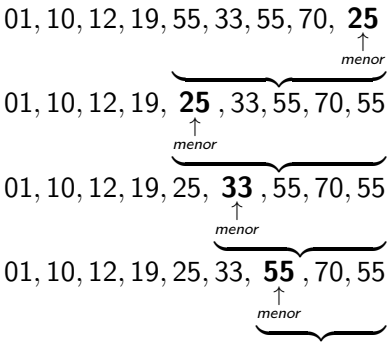


unesp

Algoritmos de ordenação - baseados em seleção

Selection-sort

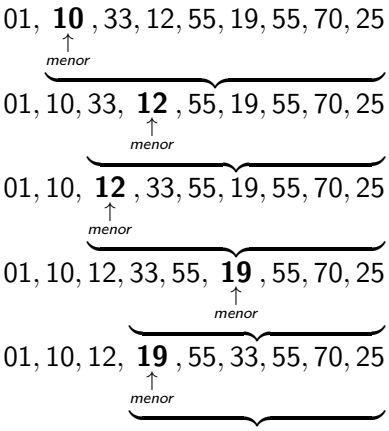


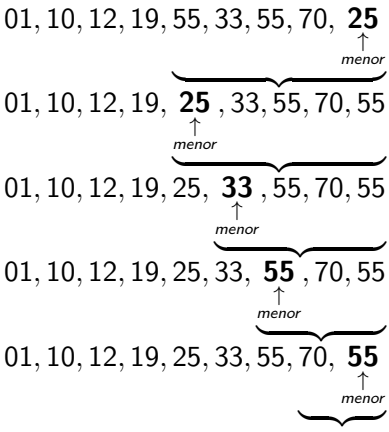


unesp

Algoritmos de ordenação - baseados em seleção

Selection-sort

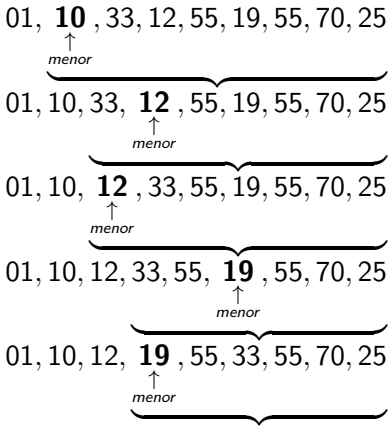




unesp

Algoritmos de ordenação - baseados em seleção

Selection-sort



Algoritmos de ordenação - baseados em seleção

Selection-sort - Exercício 0

Algoritmo:

```
1  void selectionSort(int vetor[], int tamanho)
2  {
3    int i, j, menor, index;
4    for (int i = 0; i < tamanho -1 ; ++i) {
5      menor = vetor[i];
6      index = i;
7      for (int j = i + 1; j < tamanho; ++j) {
8        if (vetor[j] < menor) {
9          menor = vetor[j];
10         index = j;
11       }
12     }
13     vetor[index] = vetor[i];
14     vetor[i] = menor;
15   }
16 }
```

Determine o tempo de execução do *Selection-sort* com seus respectivos melhores e piores tempos se houverem. Use a notação que melhor se encaixa à situação.

unesp

Algoritmos de ordenação - baseados em seleção

Selection-sort - Exercício 0

Algoritmo:

```
1  void selectionSort(int vetor[], int tamanho)
2  {
3    int i, j, menor, index;
4    for (int i = 0; i < tamanho -1 ; ++i) {
5      menor = vetor[i];
6      index = i;
7      for (int j = i + 1; j < tamanho; ++j) {
8        if (vetor[j] < menor) {
9          menor = vetor[j];
10         index = j;
11       }
12     }
13     vetor[index] = vetor[i];
14     vetor[i] = menor;
15   }
16 }
```

Determine o tempo de execução do *Selection-sort* com seus respectivos melhores e piores tempos se houverem. Use a notação que melhor se encaixa à situação.

Resposta:

$\Theta(n^2)$

unesp

São algoritmos baseados em uma estrutura dados chamada de **Heap** que nada mais é do que um tipo de **árvore binária** de dados. É uma evolução do *selection-sort*, no nosso caso vai trazer sempre em seu elemento raíz o maior valor do vetor (*heap máximo*) e será balanceado a esquerda.
Pode ser implementado de várias formas mas, como o objetivo aqui é ordenar um vetor, tal implementação não passará disso, portanto a árvore binária que representa esta estrutura de dados terá como nós filhos as expressões 34 e 35 e como nó pai 36.

$$filhoEsquerdo(k) = 2k + 1 \tag{34}$$

$$filhoDireito(k) = 2k + 2 \tag{35}$$

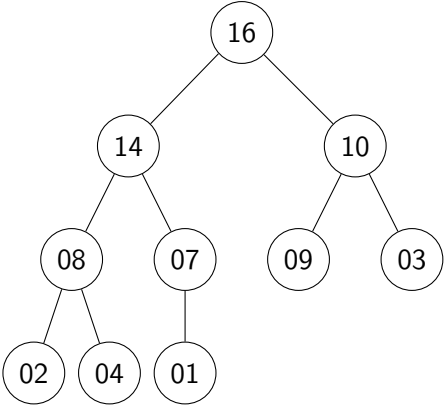
$$pai(k) = \frac{k-1}{2} \tag{36}$$

Decerto o ajuste do *Heap* tem seu custo, no entanto, devido ao balanceamento da árvore tal custo é de ordem (descubra isso no exercício).
Por fim, eis a definição do *heap-sort*:

- construir o *heap* máximo
- trocar a raiz (o maior elemento cuja localização é 0) com o elemento da última posição do vetor.
- diminuir o tamanho do *heap* em 1
- rearranjar o *heap* máximo se necessário
- repetir o processo n-1 vezes

Algoritmos de ordenação - baseados em seleção

Heap-sort

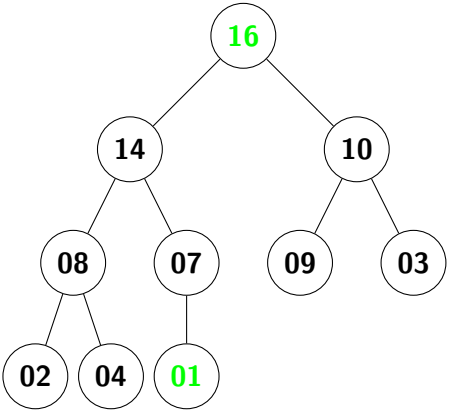


$$\underset{0}{\mathbf{16}}, \underset{1}{\mathbf{14}}, \underset{2}{\mathbf{10}}, \underset{3}{\mathbf{08}}, \underset{4}{\mathbf{07}}, \underset{5}{\mathbf{09}}, \underset{6}{\mathbf{03}}, \underset{7}{\mathbf{02}}, \underset{8}{\mathbf{04}}, \underset{9}{\mathbf{01}}$$

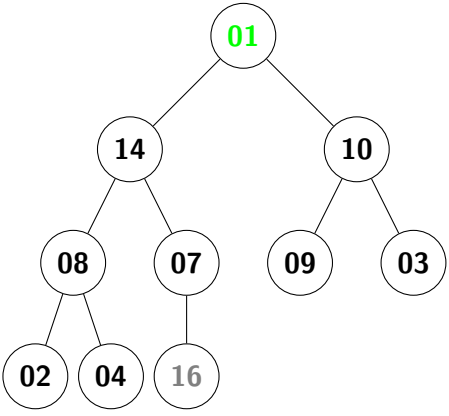
unesp

Algoritmos de ordenação - baseados em seleção

Heap-sort



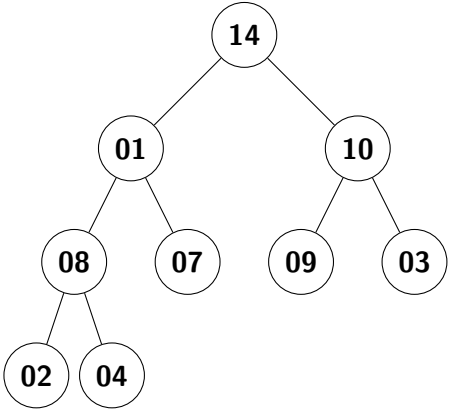
$$\underset{0}{16}, \underset{1}{14}, \underset{2}{10}, \underset{3}{08}, \underset{4}{07}, \underset{5}{09}, \underset{6}{03}, \underset{7}{02}, \underset{8}{04}, \underset{9}{01}$$



$$\underset{0}{01}, \underset{1}{14}, \underset{2}{10}, \underset{3}{08}, \underset{4}{07}, \underset{5}{09}, \underset{6}{03}, \underset{7}{02}, \underset{8}{04}, \underset{9}{16}$$

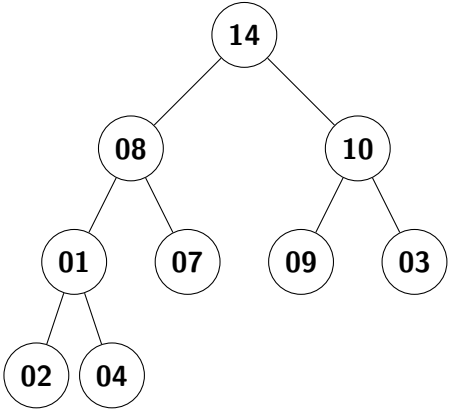
unesp

Ajustando o *heap*



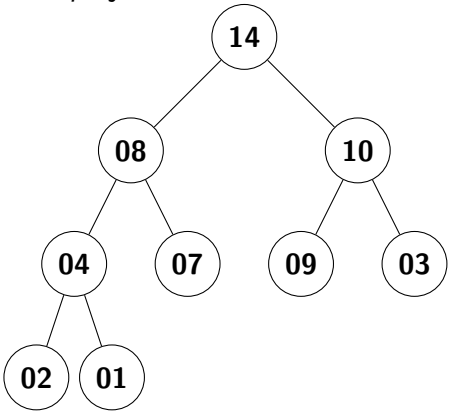
$$\underset{0}{14}, \underset{1}{01}, \underset{2}{10}, \underset{3}{08}, \underset{4}{07}, \underset{5}{09}, \underset{6}{03}, \underset{7}{02}, \underset{8}{04}, \underset{9}{16}$$

Ajustando o *heap*



$$\underset{0}{14}, \underset{1}{08}, \underset{2}{10}, \underset{3}{01}, \underset{4}{07}, \underset{5}{09}, \underset{6}{03}, \underset{7}{02}, \underset{8}{04}, \underset{9}{16}$$

heap ajustado!



14, **08**, **10**, **04**, **07**, **09**, **03**, **02**, **01**, **16**
0 1 2 3 4 5 6 7 8 9

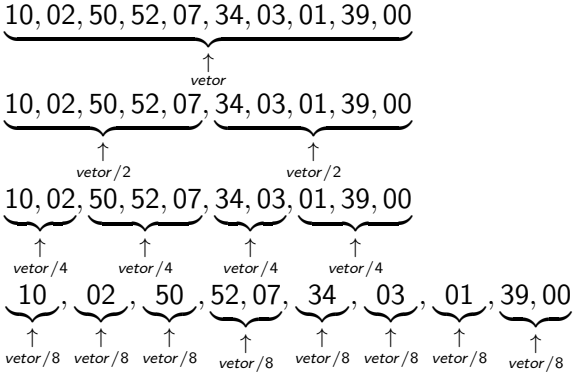
Agora é só repetir o processo!

Implemente o *heap-sort*.
Determine o tempo de execução para o **ajuste do heap máximo**, em seguida, descubra o tempo total de execução do *heap-sort* com seus respectivos melhores e piores tempos se houverem. Use a notação que melhor se encaixa à situação.

Resposta: Ajuste do heap máximo: $\Theta(\log n)$, tempo do algoritmo: $O(n.\log n)$.

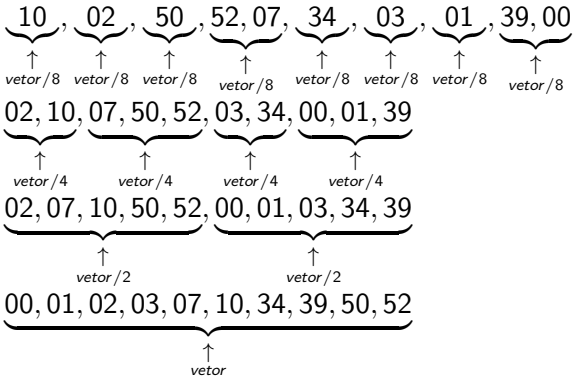
```
1  void heapify(int vetor[], int tamanho, int posicao)
2  {
3    int posicaoMaior = posicao; // Initialize largest as root
4    int filheEsquerda = 2 * posicao + 1; // left = 2*i + 1
5    int filheDireita = 2 * posicao + 2; // right = 2*i + 2
6    if (filheEsquerda < tamanho && vetor[filheEsquerda] > vetor[posicaoMaior])
7      posicaoMaior = filheEsquerda;
8    if (filheDireita < tamanho && vetor[filheDireita] > vetor[posicaoMaior])
9      posicaoMaior = filheDireita;
10   if (posicaoMaior != posicao) {
11     std::swap(vetor[posicao], vetor[posicaoMaior]);
12     heapify(vetor, tamanho, posicaoMaior);
13   }
14 }
15 void heapSort(int vetor[], int tamanho)
16 {
17   // Monta o heap
18   for (int posicao = tamanho / 2 - 1; posicao >= 0; posicao--)
19   heapify(vetor, tamanho, posicao);
20   // Ordena o vetor
21   for (int i = tamanho - 1; i >= 0; i--) {
22     // Move a raiz para o final
23     std::swap(vetor[0], vetor[i]);
24     // Arruma a arvore
25     heapify(vetor, i, 0);
26   }
27 }
```

Merge-sort também usa uma estratégia de "dividir para conquistar" até o momento em que cada elemento do vetor esteja com um par ou sozinho (caso base), a partir desse momento, os elementos são comparados dois a dois no que se chama "etapa de intercalação" , essa etapa, remonta o vetor de forma que o mesmo fique ordenado.



Algoritmos de ordenação - baseados em seleção

Merge-sort



Algoritmos de ordenação - baseados em seleção

Merge-Sort

```
1 void mergeSort(int vetor[], int const inicio, int const fim)
2 {
3   if (inicio >= fim) return; // Caso base
4   int meio = inicio + (fim - inicio) / 2; // Determina a metade do vetor
5   mergeSort(vetor, inicio, meio); // Processa a primeira metade
6   mergeSort(vetor, meio + 1, fim); // Processa a segunda metade
7   merge(vetor, inicio, meio, fim); // Combina as metades ordenadas
8 }
```

Algoritmos de ordenação - baseados em seleção

Merge-Sort

```
1  void merge(int vetor[], int const esquerda, int const meio, int const direita){
2    int const tamanhoVetor1 = meio - esquerda + 1;
3    int const tamanhoVetor2 = direita - meio;
4    int *vetorEsquerdo = new int[tamanhoVetor1];
5    int *vetorDireito = new int[tamanhoVetor2];
6    for (int i = 0; i < tamanhoVetor1; i++) vetorEsquerdo[i] = vetor[esquerda + i];
7    for (int j = 0; j < tamanhoVetor2; j++) vetorDireito[j] = vetor[meio + 1 + j];
8    int posicaoVetor1 = 0;
9    int posicaoVetor2 = 0;
10   int posicaoVetorMesclado = esquerda;
11   while (posicaoVetor1 < tamanhoVetor1 && posicaoVetor2 < tamanhoVetor2) {
12     if (vetorEsquerdo[posicaoVetor1] <= vetorDireito[posicaoVetor2]) {
13       vetor[posicaoVetorMesclado] = vetorEsquerdo[posicaoVetor1];
14       posicaoVetor1++;
15     } else {
16       vetor[posicaoVetorMesclado] = vetorDireito[posicaoVetor2];
17       posicaoVetor2++;
18     }
19     posicaoVetorMesclado++;
20   }
21   while (posicaoVetor1 < tamanhoVetor1) { // Copia os elementos restantes do esquerdo
22     vetor[posicaoVetorMesclado] = vetorEsquerdo[posicaoVetor1];
23     posicaoVetor1++;
24     posicaoVetorMesclado++;
25   }
26   while (posicaoVetor2 < tamanhoVetor2) { // Copia os elementos restantes do direito
27     vetor[posicaoVetorMesclado] = vetorDireito[posicaoVetor2];
28     posicaoVetor2++;
29     posicaoVetorMesclado++;
30   }
```

Determine o tempo de execução do *Merge-sort* com seus respectivos melhores e piores tempos se houverem. Use a notação que melhor se encaixa à situação.

Determine o tempo de execução do *Merge-sort* com seus respectivos melhores e piores tempos se houverem. Use a notação que melhor se encaixa à situação.

Resposta:

Com a relação de recorrência $T(n) = 2T(n/2) + n$ e pelo teorema mestre temos: $O(n.\log n)$

Algoritmos de busca

unesp

Algoritmos de busca

Agora que conhecemos uma boa parte dos métodos de ordenação é possível mergulhar nos algoritmos de busca, esses que são extensamente usados em aplicações como bancos de dados ou qualquer outra aplicação em que seja necessário encontrar alguma informação.

Algoritmo simples: Itere sobre o vetor comparando item-a-item com o valor procurado. Se encontrar retorne sua posição, do contrário retorne -1.

A busca binária, para que funcione bem, tem como requerimento um vetor ordenado representado como uma **grafo**, posto isso, o algoritmo recursivamente vai dividindo o vetor ao meio até que o item central seja o que está sendo procurado.

Implemente a busca binária.

Implemente a busca binária.

Resposta:

```
/**
 * Busca binaria
 * x - Valor procurado
 * vetor - Vetor onde procurar
 * esq - Limite inferior do vetor
 * dir - Limite superior do vetor
 * return -1 se item nao encontrado ou a localizacao do item
 */
int binSearch(int x, std::vector<int> vetor, int esq, int dir)
{
  int meio = (esq + dir) / 2;

  if(esq > dir){
    return -1; // Item nao encontrado
  }

  if(x == vetor[meio]) return meio;

  if(x < vetor[meio])
    return binSearch(x, vetor, esq, meio-1);
  else
    return binSearch(x, vetor, meio+1, dir);
}
```

Algoritmos baseados em grafos

unesp

Algoritmos baseados em grafos

Essa estrutura pode ajudar na resolução e modelagem de vários problemas como o de ordenação como já foi vista com o *heap-sort*, de busca como na *árvore binária*, modelagem de mapas, relacionamentos em redes de pessoas ou computadores, etc.
Os relacionamentos podem ser de:

- conexão (o que se conecta com o que)
- precedência (o que vem antes do que)
- exclusão mutua (como as coisas se agrupam? elas tem ligação?)
- prefêrência (o que se encaixa melhor e onde)

unesp

Algoritmos baseados em grafos

Grafos G são estruturas que representam relacionamentos par-a-par entre pontos ou objetos. Os pontos são os **vértices ou nós** e as **arestas** (ou arcos quando são direcionais) são os relacionamentos. **Neste material**, ao conjunto dos vértices daremos o nome de $V(G)$ e ao conjunto das arestas nomearemos como $A(G)$.

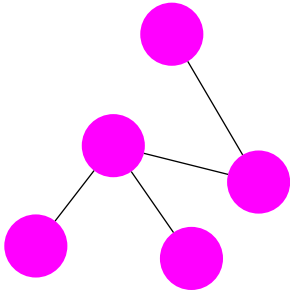


Figura: Um grafo

- se $\{a, b, c, d, e, f, g, h, i\}$ são vértices de um grafo então $G = \{a, b, c, d, e, f, g, h, i\}$
- se $a = uv / u \in G, v \in G \therefore a = aresta \in A(G)$ então:
  - u, v são adjacentes
  - u, v são extremos de a
  - a passa por uv e a é delimitada por u, v
- Se $u \in V(G)$ o grau de u é a quantidade de vizinhos $d_G(u)$
- o grau mínimo é $\delta(G)$ e o máximo é $\Delta(G)$
- $\sum_{v \in V(G)} d_G(v) = 2.|A(G)|$, ou seja, a soma dos graus é igual a duas vezes o total de arestas

unesp

- não tem arestas paralelas (duas arestas ligando o mesmo par de vértices)
- laços (um vértice ligado a ele mesmo)

Quando a lista dos vértices não é suficiente para representar as relações entre os mesmos e essas dependem de uma **procedência** se pode usar os digrafos como representação de tal situação. Um digrafo é um conjunto de dois outros conjuntos onde o primeiro representa os vértices e o segundo contém **pares ordenados** indicando as relações de precedência.

$$D = \begin{cases} V(G) = \{a, b, c, d, e, f, g\} \\ A(G) = \{ab, bg, ce, de, fg\} \end{cases} \quad (37)$$

Se $a = uv$ é um **arco** de D então:

- u é cabeça e v é a cauda
- a sai de u e vai para v
- o grau de entrada $d_D^-(u)$ de u é a qtde de arcos que entram em u
- o grau de saída $d_D^+(u)$ de u é a qtde de arcos que saem de u
- a vizinhança de entrada de u é o conjunto de todos os vértices que entram em u, $N_D^-(u)$
- a vizinhança de saída de u é o conjunto de todos os vértices que saem de u, $N_D^+(u)$

Para qualquer digrafo D vale:

- $\sum_{v \in V(D)} d_D^+(v) = \sum_{v \in V(D)} d_D^-(v) = |A(D)|$, ou seja, a soma total de todos os graus de saída é igual a soma de todos os graus de entrada que é igual ao número total de arcos.
- Todo digrafo pode gerar um grafo subjacente que nada mais é do que um grafo sem suas direções.
- Todo grafo pode ser a base de construção de um **digrafo associado** $D(G)$ adicionando-se **arcos paralelos** a cada par de vértices. Ao digrafo que não usa arcos paralelos chamaremos de **digrafo orientado** $\overrightarrow{G}$.
- Os arcos podem ter associados a eles valores $w(uv)$ que serão chamados de **pesos**.

No entanto, na maioria das vezes, representaremos grafos e digrafos da mesma forma pois, mesmo que não haja relações de precedência entre os nós, é preciso indicar as arestas de tal grafo.

Como já foi visto um grafo pode ser representado por um vetor simples desde que o algoritmo que o trata seja projetado devidamente para usar esta estrutura. No entanto, nem sempre é prático, possível ou eficiente usar uma estrutura tão simples, portanto temos que usar outros tipos:

- lista de adjacências: São listas que, dado um vértice, contém uma **lista ligada** que contêm todos os vizinhos daquele nó ou, no caso dos digrafos todos os nós de saída daquele vértice.
- matrizes de adjacências: É uma **matriz quadrada** de dimensão igual a quantidade de vértices do grafo na qual a linha u que se cruza com a coluna v indicam se tais vértices são adjacentes.

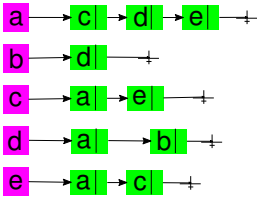


Figura: Ponteiros em rosa, lista ligada em verde

Memória ocupada: $V(G)$ ponteiros $+ 2.A(G)$ itens de lista ligada $= \Theta(V(G) + A(G))$
Tempo de busca por adjacências: $\Omega(1), O(n)$
Bom para grafos esparsos com poucas arestas pois, se $V(G) = n \implies A(G) = \frac{n.(n+1)}{2} \implies O(n^2)$ de memória.

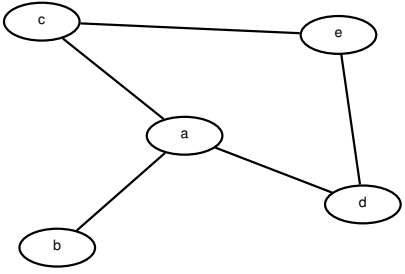


Figura: Exemplo de grafo

unesp

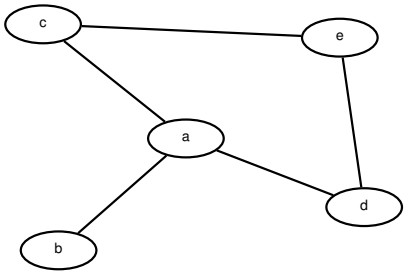


Figura: Exemplo de grafo

$$\begin{bmatrix} & \mathbf{a} & \mathbf{b} & \mathbf{c} & \mathbf{d} & \mathbf{e} \\ \mathbf{a} & 0 & 1 & 1 & 1 & 0 \\ \mathbf{b} & 1 & 0 & 0 & 0 & 0 \\ \mathbf{c} & 1 & 0 & 0 & 0 & 1 \\ \mathbf{d} & 1 & 0 & 0 & 0 & 1 \\ \mathbf{e} & 0 & 0 & 1 & 1 & 0 \end{bmatrix} \quad (38)$$

Na matriz as arestas ab, bc e dc são representadas.

Memória ocupada: $\Theta(n^2)$ para todos os casos.
Tempo de busca por adjacências: $\Theta(1)$
Melhor para grafos muito densos.

unesp

Represente os conjuntos do grafo abaixo, informe o grau de cada nó, represente a matriz de adjacências e a lista de adjacências.

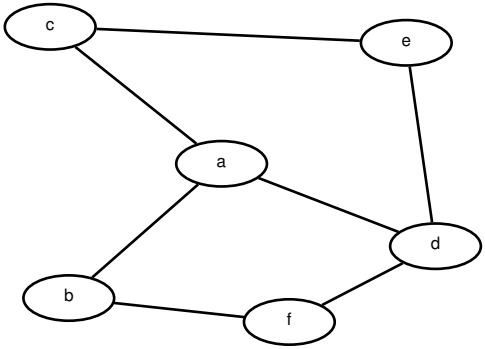


Figura: Outro exemplo de grafo

Represente os conjuntos do grafo abaixo, informe o grau de cada nó, represente a matriz de adjacências e a lista de adjacências.

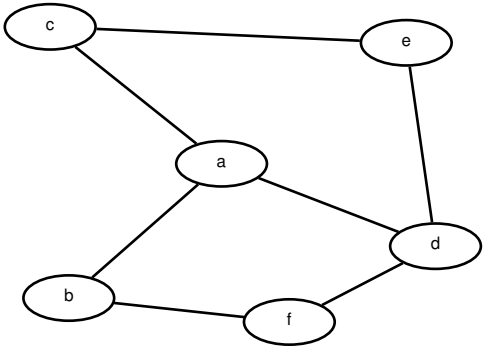


Figura: Outro exemplo de grafo

Resposta:

$V(G) = \{a, b, c, d, e, f\}$

$A(G) =$

$\{ab, ac, ad, de, df, ce, fd, fb\}$

$d_G(a) = 3$

$d_G(b) = 2$

$d_G(d) = 3$

$d_G(c) = 2$

$d_G(e) = 2$

$d_G(f) = 2$

unesp

Represente os conjuntos do grafo abaixo, informe o grau de cada nó, represente a matriz de adjacências e a lista de adjacências.

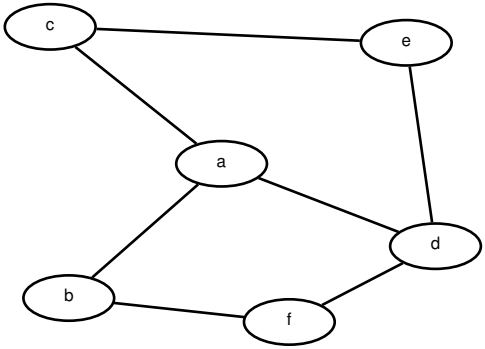


Figura: Outro exemplo de grafo

Resposta:

$$\begin{bmatrix} & \mathbf{a} & \mathbf{b} & \mathbf{c} & \mathbf{d} & \mathbf{e} & \mathbf{f} \\ \mathbf{a} & 0 & 1 & 1 & 1 & 0 & 0 \\ \mathbf{b} & 1 & 0 & 0 & 0 & 0 & 1 \\ \mathbf{c} & 1 & 0 & 0 & 0 & 1 & 0 \\ \mathbf{d} & 1 & 0 & 0 & 0 & 1 & 1 \\ \mathbf{e} & 0 & 0 & 1 & 1 & 0 & 0 \\ \mathbf{f} & 0 & 1 & 0 & 1 & 0 & 0 \end{bmatrix} \quad (39)$$

unesp

Represente os conjuntos do grafo abaixo, informe o grau de cada nó, represente a matriz de adjacências e a lista de adjacências.

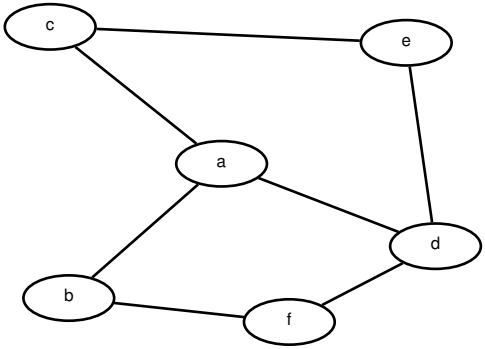


Figura: Outro exemplo de grafo

Resposta:

$a \rightarrow b \rightarrow c \rightarrow d$
$b \rightarrow a \rightarrow f$
$c \rightarrow a \rightarrow e$
$d \rightarrow e \rightarrow f$
$e \rightarrow d \rightarrow c$
$f \rightarrow d \rightarrow b$

Algoritmos baseados em grafos I

Exemplo de lista de adjacências

```
1 #include "grafo.h"
2 #include <iostream>
3 grafo* criaGrafo(int vertices, int arestas)
4 {
5   grafo* G = new grafo;
6   G->qtdeDeVertices = vertices;
7   G->qtdeDearestas = arestas;
8   G->listaDeAdjacencias.resize(vertices);
9   return G;
10 }
11
12
13 void adicionaAresta(grafo* G, int u, int v)
14 {
15   vertice* ultimo = &G->listaDeAdjacencias[u];
16
17   if (ultimo->folha) {
18     vertice* novaFolha = new vertice;
19     novaFolha->rotulo = -1;
20     novaFolha->folha = true;
21     novaFolha->adjacente = nullptr;
22
23     ultimo->rotulo = u;
24     ultimo->folha = false;
25     ultimo->adjacente = novaFolha;
26   }
27
```

inesp

Algoritmos baseados em grafos II

Exemplo de lista de adjacências

```
28  while (!ultimo->folha) {
29    if (ultimo->rotulo == v)
30    return;
31    ultimo = ultimo->adjacente;
32  }
33
34  vertice* novaFolha = new vertice;
35  novaFolha->rotulo = -1;
36  novaFolha->folha = true;
37  novaFolha->adjacente = nullptr;
38
39  ultimo->rotulo = v;
40  ultimo->folha = false;
41  ultimo->adjacente = novaFolha;
42  ////////////////----///////////////////
43  ultimo = &G->listaDeAdjacencias[v];
44
45  if (ultimo->folha) {
46    vertice* novaFolha = new vertice;
47    novaFolha->rotulo = -1;
48    novaFolha->folha = true;
49    novaFolha->adjacente = nullptr;
50
51    ultimo->rotulo = v;
52    ultimo->folha = false;
53    ultimo->adjacente = novaFolha;
54  }
```

inesp

Algoritmos baseados em grafos III

Exemplo de lista de adjacências

```
55
56    while (!ultimo->folha) {
57      if (ultimo->rotulo == u)
58      return;
59      ultimo = ultimo->adjacente;
60    }
61
62    novaFolha = new vertice;
63    novaFolha->rotulo = -1;
64    novaFolha->folha = true;
65    novaFolha->adjacente = nullptr;
66
67    ultimo->rotulo = u;
68    ultimo->folha = false;
69    ultimo->adjacente = novaFolha;
70  }
71
72  void imprimeGrafo(grafo* G)
73  {
74    for (vertice v : G->listaDeAdjacencias) {
75      std::cout << v.rotulo << " -> ";
76
77      v = *v.adjacente;
78      while (!v.folha) {
79        std::cout << v.rotulo << " ";
80        v = *v.adjacente;
81      }
```

Algoritmos baseados em grafos IV

Exemplo de lista de adjacências

```
82      std::cout << std::endl;
83    }
84  }
85
86  void deletaGrafo(grafo* G)
87  {
88    vertice *x, *y;
89
90    for (int indiceDoVertice = 0; indiceDoVertice < G->qtdeDeVertices;
91    indiceDoVertice++) {
92      x = &G->listaDeAdjacencias[indiceDoVertice];
93      while (x) {
94        y = x->adjacente;
95        delete x;
96        x = nullptr;
97        x = y;
98      }
99    }
100   delete G;
101 }
```

Programe em c ou c++ a **matriz de adjacências** e adicione os nós como representado abaixo:

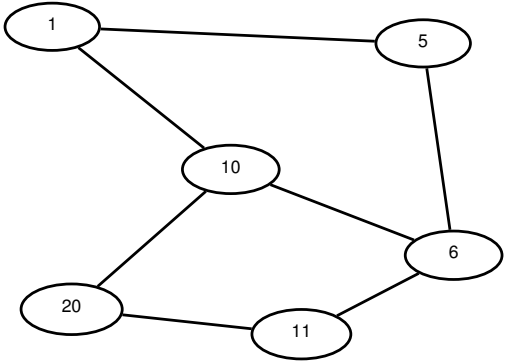


Figura: Outro exemplo de grafo

Até agora vimos como construir e como consultar algumas informações de um grafo mas e se precisarmos **remover** algum nó?

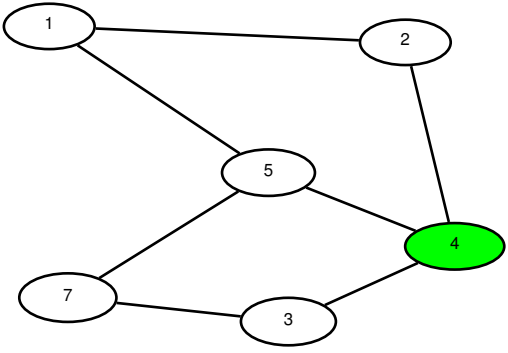


Figura: Remoção de vértice

$$\begin{bmatrix} & \mathbf{1} & \mathbf{2} & \mathbf{3} & \mathbf{4} & \mathbf{5} & \mathbf{6} & \mathbf{7} \\ \mathbf{1} & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ \mathbf{2} & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ \mathbf{3} & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ \mathbf{4} & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ \mathbf{5} & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ \mathbf{6} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \mathbf{7} & 0 & 0 & 1 & 0 & 1 & 0 & 0 \end{bmatrix} \quad (39)$$

Basta essa representação para excluir um nó???

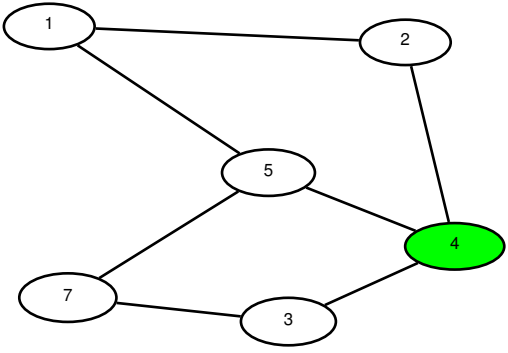


Figura: Remoção de vértice

$$\begin{bmatrix} & \mathbf{1} & \mathbf{2} & \mathbf{3} & \mathbf{4} & \mathbf{5} & \mathbf{6} & \mathbf{7} \\ \mathbf{1} & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ \mathbf{2} & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ \mathbf{3} & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ \mathbf{4} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \mathbf{5} & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ \mathbf{6} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \mathbf{7} & 0 & 0 & 1 & 0 & 1 & 0 & 0 \end{bmatrix} \quad (39)$$

Não! O nó continua existindo mas, sem ligação alguma.

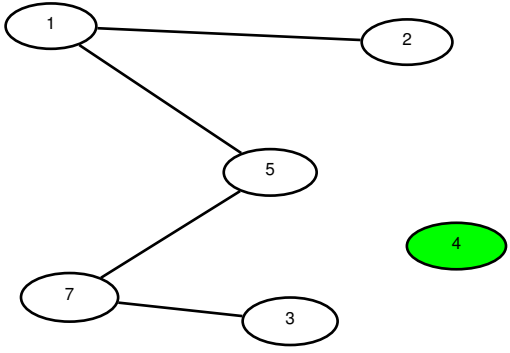


Figura: Remoção de vértice

$$\begin{bmatrix} & \mathbf{1} & \mathbf{2} & \mathbf{3} & \mathbf{4} & \mathbf{5} & \mathbf{6} & \mathbf{7} \\ \mathbf{1} & 0 & 1 & 0 & -1 & 1 & 0 & 0 \\ \mathbf{2} & 1 & 0 & 0 & -1 & 0 & 0 & 0 \\ \mathbf{3} & 0 & 0 & 0 & -1 & 0 & 0 & 1 \\ \mathbf{4} & -1 & -1 & -1 & -1 & -1 & -1 & -1 \\ \mathbf{5} & 1 & 0 & 0 & -1 & 0 & 0 & 1 \\ \mathbf{6} & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ \mathbf{7} & 0 & 0 & 1 & -1 & 1 & 0 & 0 \end{bmatrix} \quad (39)$$

Porém podemos introduzir um terceiro símbolo que dá conta do recado!

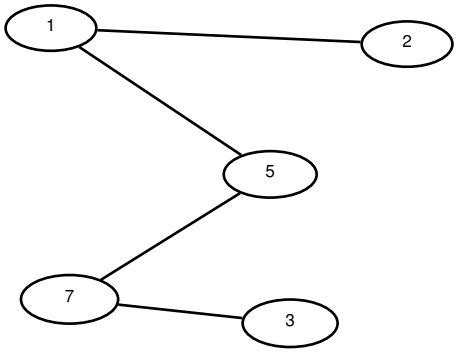


Figura: Remoção de vértice

Implemente em c ou c++ a **remoção de nós** e a **remoção de arestas**

Considere:

$$\begin{cases} V(G) = \{a, b, c, d, e, f\} \\ A(G) = \{ab, ac, ad, dc, ef, fa, be\} \end{cases} \quad (40)$$

Então:
Seja u e v dois vértices, um conjunto H é subgrafo de G se
$V(H) \subseteq V(G), A(H) \subseteq A(G)/(u, v) \in V(H)$
Um subgrafo H é **gerador** se $H \subseteq G, V(H) = V(G)$
Se $S \subseteq V(G), G[S]$ é o grafo induzido por S
Se $F \subseteq A(G), G[S]$ é o grafo induzido por F

É importante notar que os subgrafos mantém as relações definidas anteriormente entre seus vértices.

Passeios são operações cuja finalidade é visitar os vértices de um grafo, tal procedimento só é possível se houverem arestas entre os vértices visitados.

$$\begin{cases} V(G) = \{a, b, c, d, e, f\} \\ A(G) = \{ab, ac, ad, dc, ef, fa, be\} \end{cases} \tag{41}$$

considerando o grafo acima **um dos** passeios possíveis é:
$P_1 = \{a, b, e, f, a, c\}$
O comprimento do passeio é dado pela **quantidade de vértices do mesmo**.
Passeios fechados começam e terminam no mesmo vértice.
Passeios abertos começam e terminam em vértices distintos.

Passeios que **não repetem** vértices são chamados de **caminhos**. Caminhos com n vértices são chamados de P_n: P_1, P_5, etc.
Passeios que **não repetem** vértices **exceto os extremos** são chamados de **ciclo**. Ciclos com n vértices são chamados de C_n
Passeios geram subgrafos.
Grafos completos são aqueles nos quais existem arestas entre todos os vértices. Grafos completos com n vértices são chamados de K_n: $A(K_n) = \frac{n(n-1)}{2}$
G é conexo se **todos** os pares de vértices são conectados por arestas.
Em **digrafos** pode **ou não** ser verdade que $C_n = \{a, b, c, d\} \implies C_n^{'} = \{d, c, b, a\}$
Em um **digrafo** se houverem caminhos entre quaisquer vértices $a \to b$ e $b \to a$ então esse digrafo é **fortemente conexo**.

Crie exemplos que mostrem **todas** as propriedades dos **digrafos, grafos e seus respectivos subgrafos e passeios**.

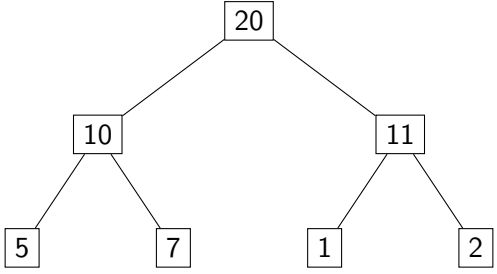
A distância ($dist_G(u, v)$) entre dois vértices em um digrafo é dada pela quantidade de arestas de um vértice até o outro, se estes vértices não estiverem conectados então se diz que a distância é infinita (∞).
Lembre-se que a distância em grafos e digrafos pode variar sensivelmente.
Em grafos e digrafos cujas arestas tem **pesos** a distância ($dist^w_G(u, v)$) é dada pelo **menor** valor da soma das arestas (menor distância).

O caminho mínimo é aquele que tem o menor valor de soma das arestas, atualmente esse problema só é possivel de se resolver se:

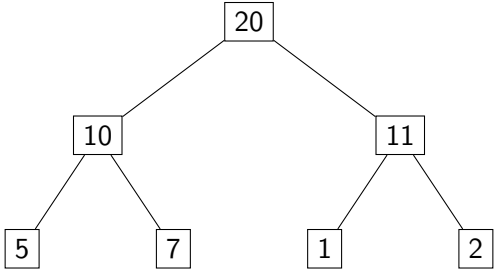
- o grafo não tem aresta com peso negativo
- o digrafo não tem um ciclo com peso negativo

Crie um algoritmo que calcula a distância mínima entre dois vértices.
Qual o tempo de execução assintótico?
Esse tempo é bom? Justifique.

Como já vimos anteriormente no algoritmo *heap-sort* as vezes é necessário a implementação (seja na lógica ou em uma estrutura) de uma árvore mas, o que é uma isso? Assim como na natureza a árvore representa uma estrutura hierárquica na qual algumas partes se ligam a **uma** outra parte de forma recursiva formando assim uma estrutura parecida com que se pode chamar de fractal de linhas e nós. No contexto dos algoritmos de busca árvores são extremamente úteis pois, geralmente, diminuem muito o tempo para localização de um valor.



unesp



Considerando que um grafo é dito **conexo** se existir **pelo menos um caminho entre cada par de vértices** do grafo. Se pode dizer que uma árvore é um **grafo conexo** que **não contêm ciclos** e, por consequência, cujo o número de arestas é o número de vértices - 1.

Dê exemplos de grafos que não são árvores.

Se o grafo G representa uma árvore (T) então **é equivalente** dizer que:

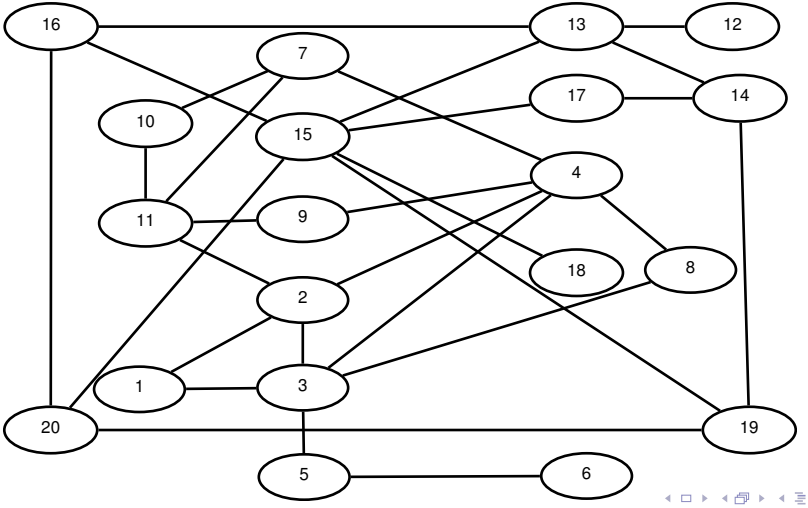
- existe um único caminho entre os dois vértices de G
- G é conexo para toda aresta pertencente a G ($a \in A(G)$) e remover um aresta o torna desconexo.
- é um **grafo conexo** que **não contêm ciclos** e, por consequência, cujo o número de arestas é o número de vértices - 1
- se G não tem ciclos então todo par de vértices **não adjacentes** formará um ciclo caso se trace um aresta entre os mesmos.

Se o grafo G representa uma árvore então:

- Se $uv \notin A(T), uv \in V(T)$ então adicionar esta aresta cria um **ciclo**.
- Se $\forall c \in A(T)$ formam ciclos então suas deleções criam uma árvore.
- Se $T \subseteq G$ e T é uma árvore e $V(T) = V(G)$ então G é conexo e $V(T) = V(G)$ é **geradora**.
- Criar um aresta entre um vértice que **não** pertence a árvore e outro que pertence mantém a árvore já que não cria ciclos.
- Em **digrafos** árvores se chamam **arvorecências** sendo que todo vértice deve ter grau de entrada $= 1$ **exceto o vértice raíz**.

Dado um grafo não árvore, é possível, a partir dele, construir uma árvore usando o algoritmo de busca em largura, encontrar algum vértice ou ainda determinar se um grafo é conexo ou não.

Dado o grafo abaixo vamos fazer a busca em largura do mesmo.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
V	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

$F : \{\}$

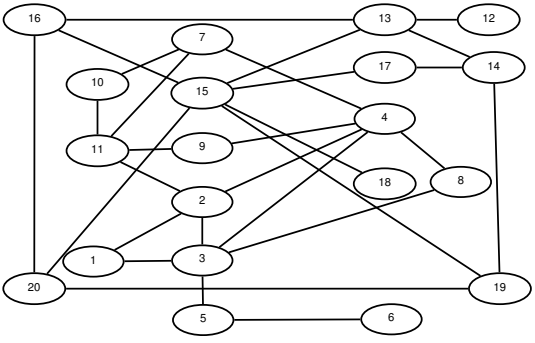


Figura: Grafo

Esse algoritmo começa com a criação de uma estrutura que marca quais vértices foram visitados (V), quais seus respectivos predecessores (A) e uma fila (F).
Em seguida se escolhe de forma arbitrária um vértice por onde começar. Comecemos pelo vértice 2, sendo assim, o mesmo entra na fila.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
V	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	2		2	2	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0

$F : \{2, 4, 3, 1, 11\}$

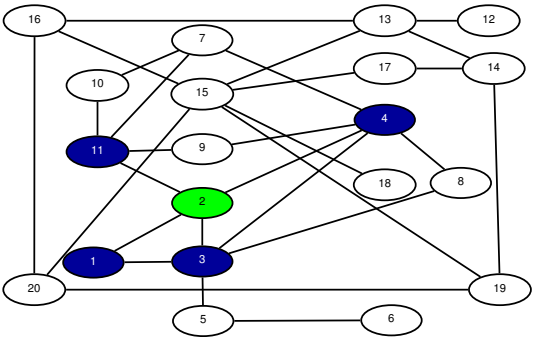


Figura: Grafo

Agora removemos o item mais antigo da fila e depois adicionamos na mesma seus vizinhos.
Em seguida marcamos os nós visitados e indicamos seu predecessor (que é o 2).

unesp

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
V	1	1	1	1	0	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0
A	2		2	2	0	0	4	4	0	0	2	0	0	0	0	0	0	0	0	0

$F : \{\cancel{2}, \cancel{4}, 3, 1, 11, 7, 8\}$

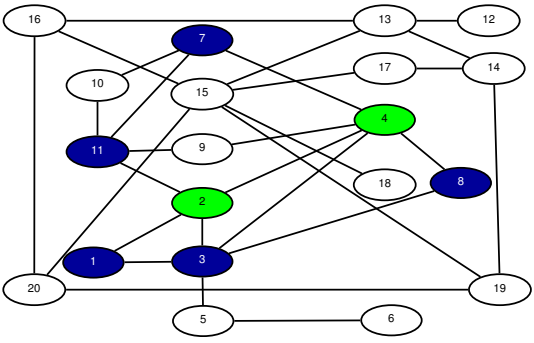


Figura: Grafo

Então **novamente** removemos o item mais antigo da fila e depois adicionamos na mesma seus vizinhos.
Em seguida marcamos os nós visitados e indicamos seu predecessor (que é o 4).
Se faz isso **até que não exista item algum na fila**

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
V	1	1	1	1	1	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0
A	2		2	2	3	0	4	4	0	0	2	0	0	0	0	0	0	0	0	0

$F : \{2, 4, 3, 1, 11, 7, 8, 5\}$

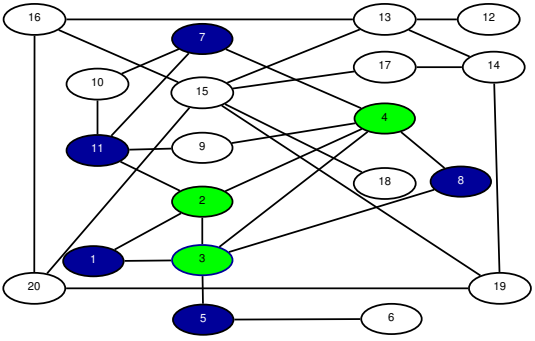


Figura: Grafo

unesp

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
V	1	1	1	1	1	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0
A	2		2	2	3	0	4	4	0	0	2	0	0	0	0	0	0	0	0	0

$F : \{2, 4, 3, 1, 11, 7, 8, 5\}$

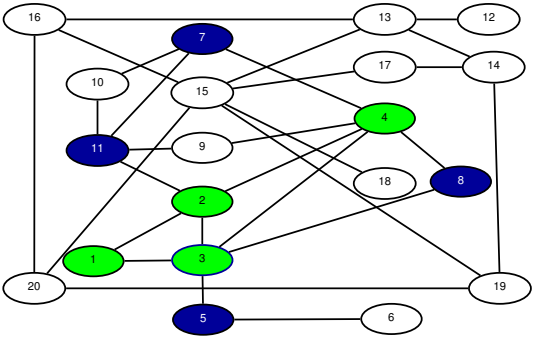


Figura: Grafo

unesp

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
V	1	1	1	1	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0
A	2		2	2	3	0	4	4	11	11	2	0	0	0	0	0	0	0	0	0

$F : \{2, 4, 3, 1, 11, 7, 8, 5, 10, 9\}$

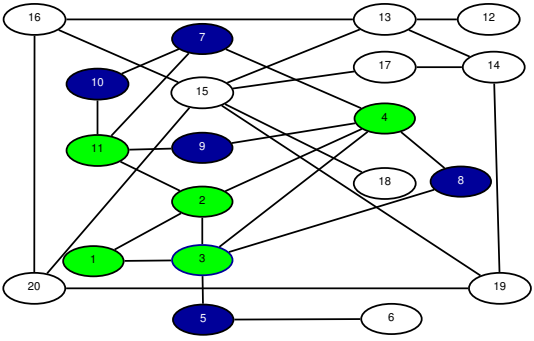


Figura: Grafo

unesp

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
V	1	1	1	1	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0
A	2		2	2	3	0	4	4	11	11	2	0	0	0	0	0	0	0	0	0

$F : \{2, 4, 3, 1, 11, 7, 8, 5, 10, 9\}$

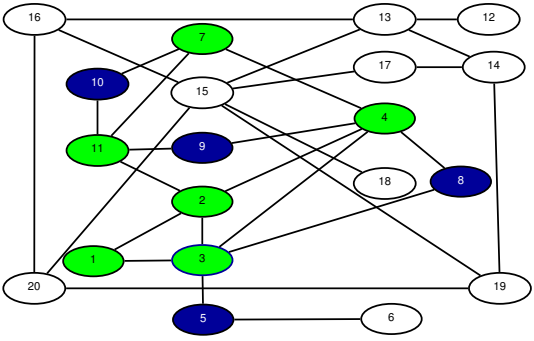


Figura: Grafo

unesp

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
V	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0
A	2		2	2	3	5	4	4	11	11	2	0	0	0	0	0	0	0	0	0

$F : \{2, 4, 3, 1, 11, 7, 8, 5, 10, 9\}$

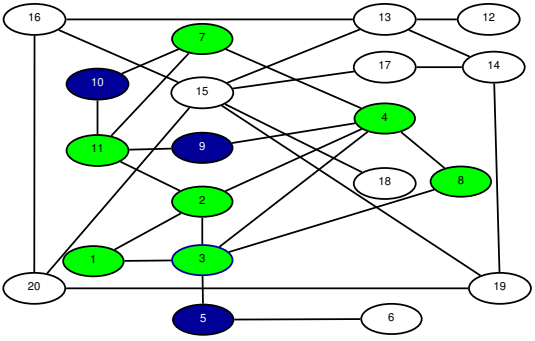


Figura: Grafo

unesp

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
V	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0
A	2		2	2	3	5	4	4	11	11	2	0	0	0	0	0	0	0	0	0

$F : \{2, 4, 3, 1, 11, 7, 8, 5, 10, 9, 6\}$

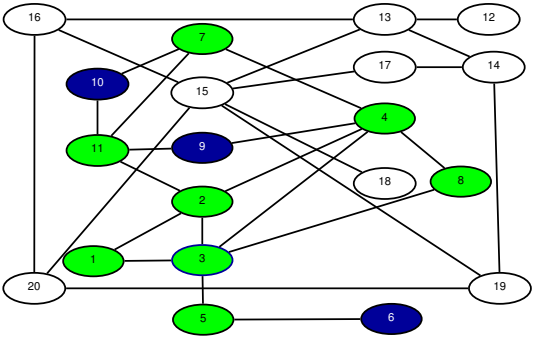


Figura: Grafo

unesp

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
V	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0
A	2		2	2	3	5	4	4	11	11	2	0	0	0	0	0	0	0	0	0

$F : \{2, 4, 3, 1, 11, 7, 8, 5, 10, 9, 6\}$

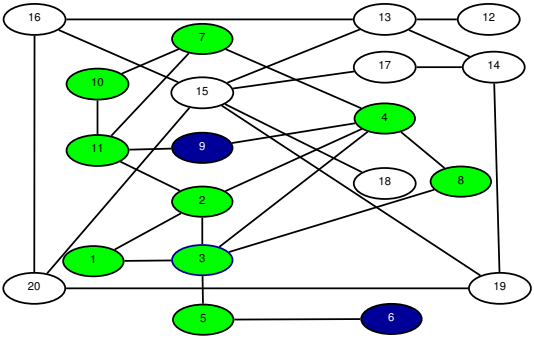


Figura: Grafo

unesp

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
V	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0
A	2		2	2	3	5	4	4	11	11	2	0	0	0	0	0	0	0	0	0

$F : \{2, 4, 3, 1, 11, 7, 8, 5, 10, 9, 6\}$

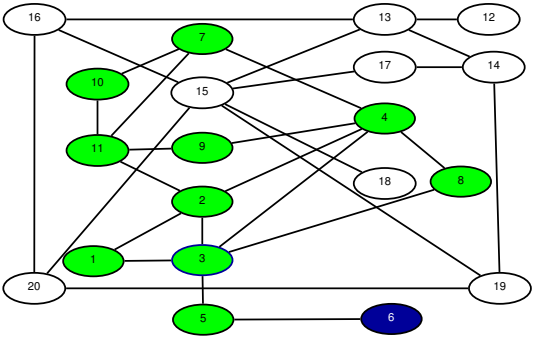


Figura: Grafo

unesp

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
V	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0
A	2		2	2	3	5	4	4	11	11	2	0	0	0	0	0	0	0	0	0

$F : \{2, 4, 3, 1, 11, 7, 8, 5, 10, 9, 6\}$

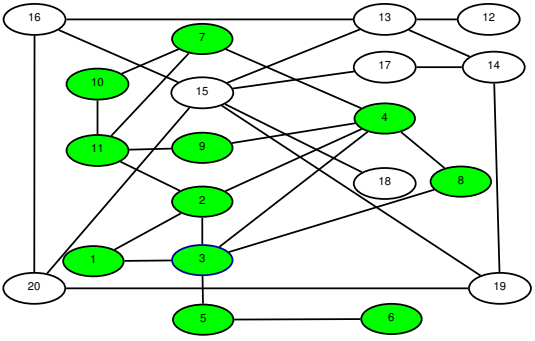


Figura: Grafo

unesp

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
V	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0
A	2		2	2	3	5	4	4	11	11	2	0	0	0	0	0	0	0	0	0

$F : \{2, 4, 3, 1, 11, 7, 8, 5, 10, 9, 6\}$

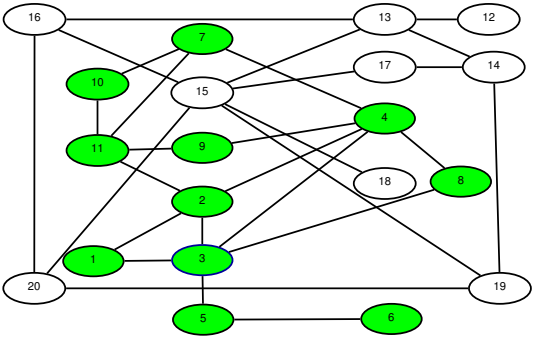


Figura: Grafo

Fim!
Perceba que uma parte do grafo ficou de fora, ou seja, este é um grafo desconexo, mesmo que dentro dele existam **dois** grafos conexos.

unesp

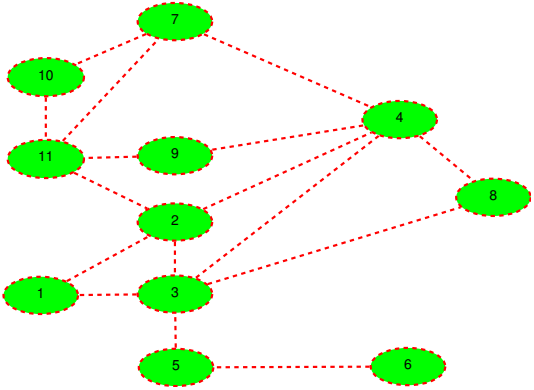


Figura: Subgrafo 01

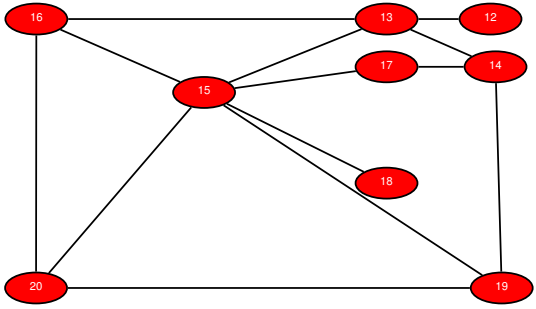


Figura: Subgrafo 02

unesp

Implemente a busca em largura e resolva o problema demonstrado nos slides anteriores. Qual o tempo assintótico de execução?
A partir da raiz da árvore criada é possível achar o menor caminho até um outro vértice qualquer? Como?

Resposta:

O tempo de execução depende da implementação (matriz ou lista de adjacências).

Para listas: $\Theta(1) + \Theta(1) + \Theta(1) + \Theta(V(G)) + O(A(G)) = O(V(G) + A(G))$

Para Matriz: $\Theta(1) + \Theta(1) + \Theta(1) + \Theta(V)G)) * \Theta(V(G)) = \Theta(V(G)^2)$

```
1  void buscaEmLargura(grafo* G, int item)
2  {
3    // Cria fila vazia
4    std::queue<vertice*> fila; // Theta(1)
5
6    // Marca o primeiro como visitado
7    G->listaDeAdjacencias[item].visitado = true; // Theta(1)
8
9    // Enfileira
10   fila.push(&G->listaDeAdjacencias[item]); // Theta(1)
11
12   // Enquanto a fila nao esta vazia -> Theta(V(G)) ja que cada vertice eh enfileirado apenas UMA vez
13   while (!fila.empty()) {
14
15     // Desinfileira
16     auto u = fila.front();
17     fila.pop();
18
```

Algoritmos de busca em grafos II

Árvores - Exercício 06

```
19      // Para todo vertice vizinho de u -> Para percorrer todos os elementos das listas de adjacencias
          gasta-se O(A)
20      auto vizinho = u->adjacente;
21      while (vizinho) {
22        if (G->listaDeAdjacencias[vizinho->rotulo].folha || vizinho->folha)
23        break;
24        if (!G->listaDeAdjacencias[vizinho->rotulo].visitado) {
25
26          // Adiciona na fila
27          fila.push(&G->listaDeAdjacencias[vizinho->rotulo]);
28
29          // Marca como visitado -> O(V(G)) pelos motivos acima
30          G->listaDeAdjacencias[vizinho->rotulo].visitado = true;
31
32          // Exibe -> O(V(G)) pelos motivos acima
33          std::cout << " -> " << vizinho->rotulo << std::flush;
34        }
35        vizinho = vizinho->adjacente;
36      }
37    }
38  }
```

unesp

Ao contrário do algoritmo anterior essa busca se dá pelo **vértice descoberto mais recentemente** até que o mesmo não tenha mais vizinhos. Descobertos todos os vizinhos a busca se volta para o vértice anterior. Esse tipo de busca monta **florestas** a partir dos grafos dados.
Os passos do algoritmo são os seguintes:

1. escolha um vértice não vistado, se não houver **pare**
2. marque-o como visitado
3. **empilhe** ele
4. escolha um vizinho, senão houver volte para 1
5. marque-o como visitado
6. se o vizinho tiver vizinhos **empilhe ele** senão desempilhe
7. Volte ao item 4

Implemente a busca em profundidade alterando o BFS
Alerta de spoiler o tempo de execução é $O(V(G) + A(G))$! Porque?

Uma **árvore binária completa** tem todos os níveis completamente cheios, exceto possivelmente o último, sendo que todos os nós no último nível estão tanto à esquerda quanto possível. Já uma **cheia** tem cada um do nós com 2 ou 0 ramos.

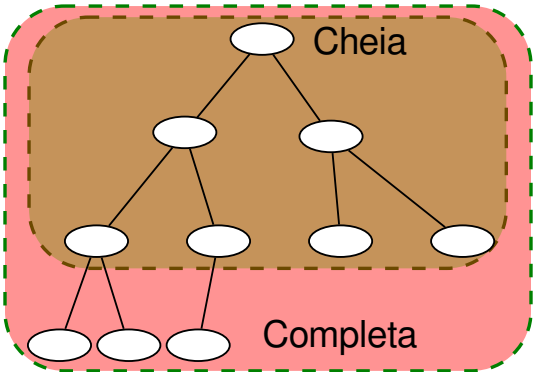


Figura: Árvore binária

Além de cada nó ter, no máximo, dois ramos os nós filhos a esquerda devem ser menores que sua raiz e os da direita devem ser maiores.
Se não houver um cuidado em balancear a árvore ela pode degenerar em uma lista.

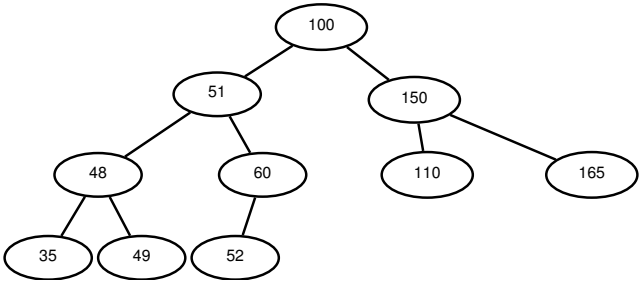


Figura: Árvore binária de busca

Cada nó tem, no máximo, dois ramos ou folhas.
Não é degenerada, ou seja, não balanceada.
Dada um árvore binária a diferença de altura entre os ramos à esquerda e a direita vai ser **no máximo** 1.
A AVLs tem nós, se diz que um nó é **regulado** quando a diferença entre as alturas de seus ramos é **no máximo** 1.
Quando todos os nós da árvore estão regulados então temos uma AVL.

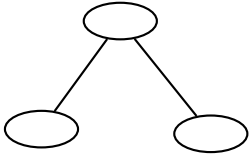


Figura: AVL

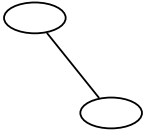


Figura: AVL

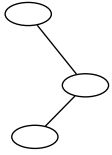


Figura: Não AVL

É uma forma de realizar um **percurso** na árvore binária. Um percurso é uma forma **sistemática** de percorrer os nós de uma árvore.
No percuso em ordem o último elemento a ser exibido está sempre ao lado direito da raiz das árvores e sub-árvores e quando um elemento **não tem** filho a esquerda **que não foi mostrado** ele também é mostrado

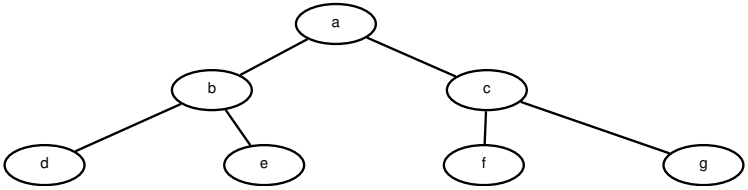


Figura: Árvore binária

unesp

É uma forma de realizar um **percurso** na árvore binária. Um percurso é uma forma **sistemática** de percorrer os nós de uma árvore.
No percuso em ordem o último elemento a ser exibido está sempre ao lado direito da raiz das árvores e sub-árvores e quando um elemento **não tem** filho a esquerda **que não foi mostrado** ele também é mostrado
$G = \{d\}$

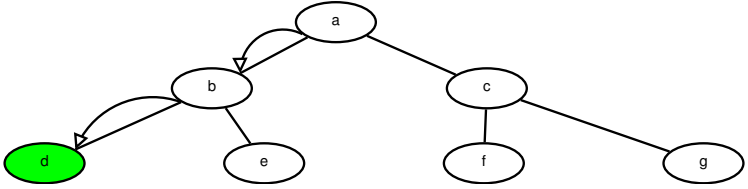


Figura: Em ordem

É uma forma de realizar um **percurso** na árvore binária. Um percurso é uma forma **sistemática** de percorrer os nós de uma árvore.
No percuso em ordem o último elemento a ser exibido está sempre ao lado direito da raiz das árvores e sub-árvores e quando um elemento **não tem** filho a esquerda **que não foi mostrado** ele também é mostrado
$G = \{d, b\}$

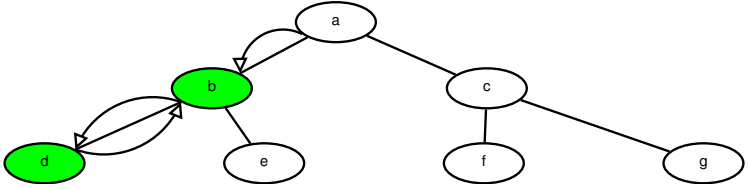


Figura: Em ordem

É uma forma de realizar um **percurso** na árvore binária. Um percurso é uma forma **sistemática** de percorrer os nós de uma árvore.
No percuso em ordem o último elemento a ser exibido está sempre ao lado direito da raiz das árvores e sub-árvores e quando um elemento **não tem** filho a esquerda **que não foi mostrado** ele também é mostrado
$G = \{d, b, e\}$

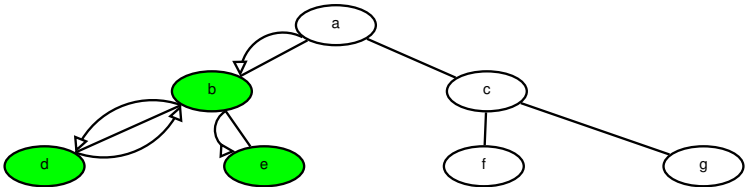


Figura: Em ordem

É uma forma de realizar um **percurso** na árvore binária. Um percurso é uma forma **sistemática** de percorrer os nós de uma árvore.
No percuso em ordem o último elemento a ser exibido está sempre ao lado direito da raiz das árvores e sub-árvores e quando um elemento **não tem** filho a esquerda **que não foi mostrado** ele também é mostrado
$G = \{d, b, e, a\}$

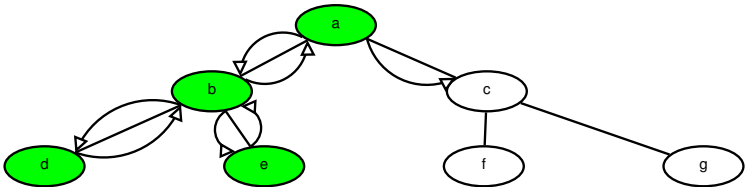


Figura: Em ordem

É uma forma de realizar um **percurso** na árvore binária. Um percurso é uma forma **sistemática** de percorrer os nós de uma árvore.
No percuso em ordem o último elemento a ser exibido está sempre ao lado direito da raiz das árvores e sub-árvores e quando um elemento **não tem** filho a esquerda **que não foi mostrado** ele também é mostrado
$G = \{d, b, e, a, f\}$

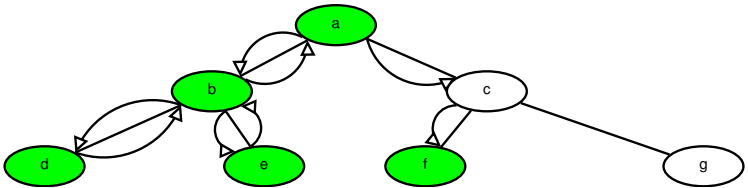


Figura: Em ordem

É uma forma de realizar um **percurso** na árvore binária. Um percurso é uma forma **sistemática** de percorrer os nós de uma árvore.
No percuso em ordem o último elemento a ser exibido está sempre ao lado direito da raiz das árvores e sub-árvores e quando um elemento **não tem** filho a esquerda **que não foi mostrado** ele também é mostrado
$G = \{d, b, e, a, f, c\}$

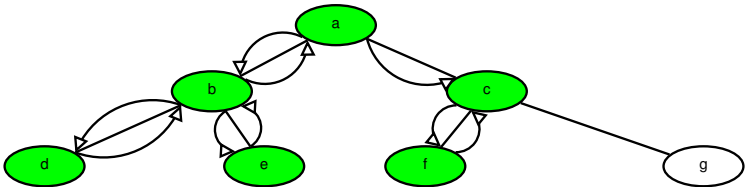


Figura: Em ordem

É uma forma de realizar um **percurso** na árvore binária. Um percurso é uma forma **sistemática** de percorrer os nós de uma árvore.
No percuso em ordem o último elemento a ser exibido está sempre ao lado direito da raiz das árvores e sub-árvores e quando um elemento **não tem** filho a esquerda **que não foi mostrado** ele também é mostrado
$G = \{d, b, e, a, f, c, g\}$

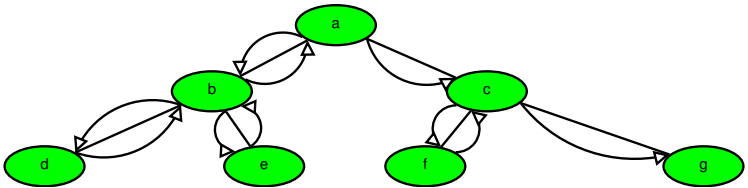


Figura: Em ordem

No percuso pós-ordem a raiz da árvore será o último nós a ser visitado, sendo assim, considerando que sub-árvores também tem suas raízes as mesmas terão, da mesma forma, suas raízes visitadas por último. Notou o padrão recursivo?

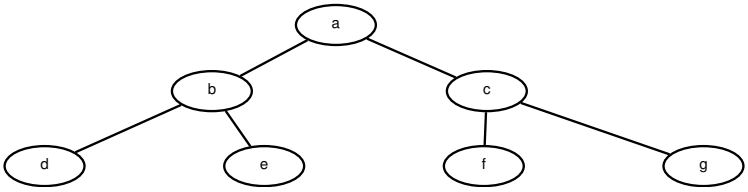


Figura: Árvore binária

No percuso pós-ordem a raiz da árvore será o último nós a ser visitado, sendo assim, considerando que sub-árvores também tem suas raízes as mesmas terão, da mesma forma, suas raízes visitadas por último. Notou o padrão recursivo?
$G = \{\emptyset\}$

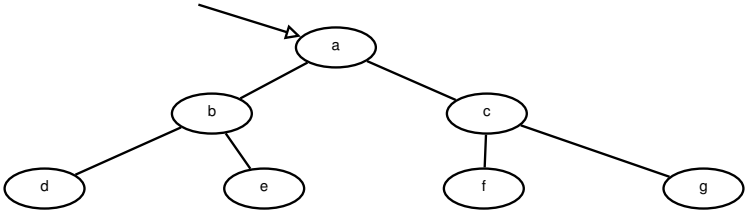


Figura: Pós-ordem

No percuso pós-ordem a raiz da árvore será o último nós a ser visitado, sendo assim, considerando que sub-árvores também tem suas raízes as mesmas terão, da mesma forma, suas raízes visitadas por último. Notou o padrão recursivo?
$G = \{\emptyset\}$

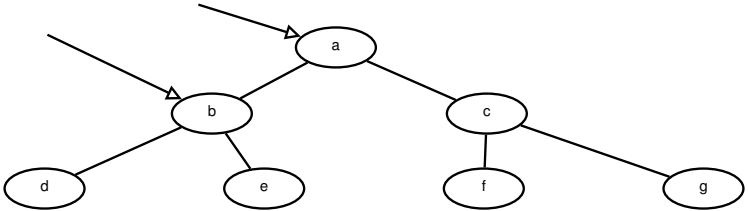


Figura: Pós-ordem

No percuso pós-ordem a raiz da árvore será o último nós a ser visitado, sendo assim, considerando que sub-árvores também tem suas raízes as mesmas terão, da mesma forma, suas raízes visitadas por último. Notou o padrão recursivo?
$G = \{d\}$

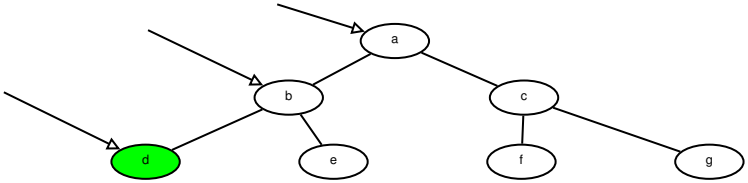


Figura: Pós-ordem

No percuso pós-ordem a raiz da árvore será o último nós a ser visitado, sendo assim, considerando que sub-árvores também tem suas raízes as mesmas terão, da mesma forma, suas raízes visitadas por último. Notou o padrão recursivo?
$G = \{d, e\}$

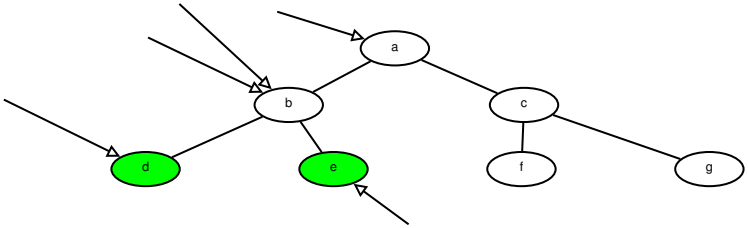


Figura: Pós-ordem

No percuso pós-ordem a raiz da árvore será o último nós a ser visitado, sendo assim, considerando que sub-árvores também tem suas raízes as mesmas terão, da mesma forma, suas raízes visitadas por último. Notou o padrão recursivo?
$G = \{d, e, b\}$

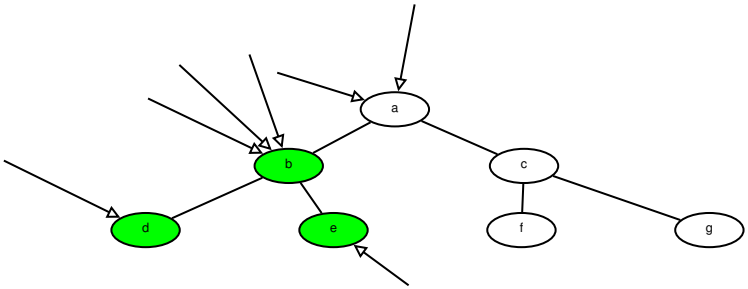


Figura: Pós-ordem

No percuso pós-ordem a raiz da árvore será o último nós a ser visitado, sendo assim, considerando que sub-árvores também tem suas raízes as mesmas terão, da mesma forma, suas raízes visitadas por último. Notou o padrão recursivo?
$G = \{d, e, b\}$

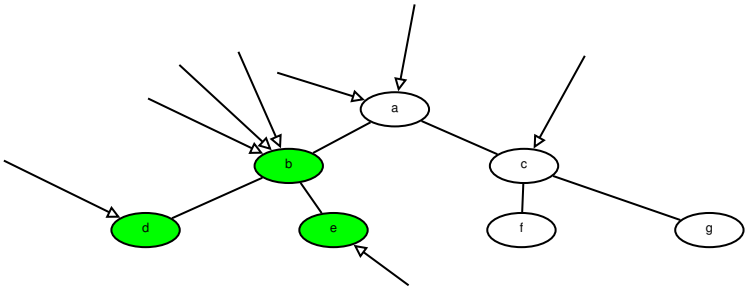


Figura: Pós-ordem

No percuso pós-ordem a raiz da árvore será o último nós a ser visitado, sendo assim, considerando que sub-árvores também tem suas raízes as mesmas terão, da mesma forma, suas raízes visitadas por último. Notou o padrão recursivo?
$G = \{d, e, b, f\}$

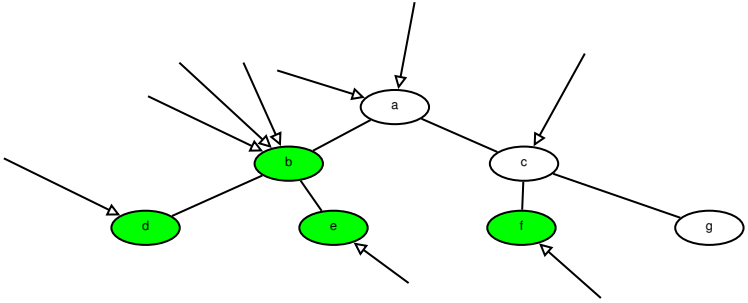


Figura: Pós-ordem

No percuso pós-ordem a raiz da árvore será o último nós a ser visitado, sendo assim, considerando que sub-árvores também tem suas raízes as mesmas terão, da mesma forma, suas raízes visitadas por último. Notou o padrão recursivo?
$G = \{d, e, b, f, g\}$

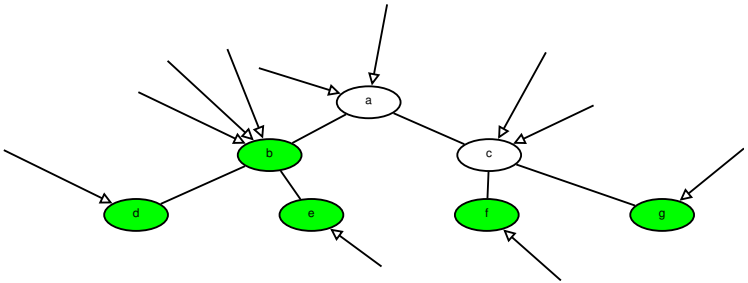


Figura: Pós-ordem

No percuso pós-ordem a raiz da árvore será o último nós a ser visitado, sendo assim, considerando que sub-árvores também tem suas raízes as mesmas terão, da mesma forma, suas raízes visitadas por último. Notou o padrão recursivo?
$G = \{d, e, b, f, g, c\}$

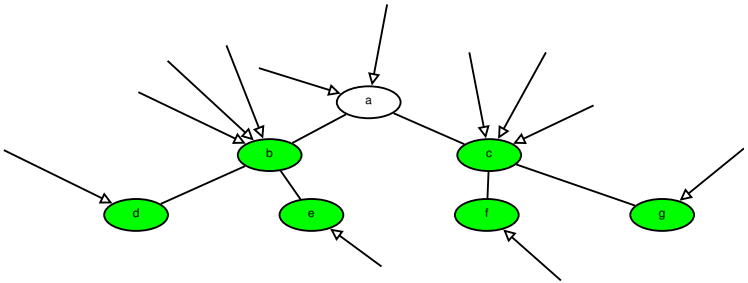


Figura: Pós-ordem

No percuso pós-ordem a raiz da árvore será o último nós a ser visitado, sendo assim, considerando que sub-árvores também tem suas raízes as mesmas terão, da mesma forma, suas raízes visitadas por último. Notou o padrão recursivo?
$G = \{d, e, b, f, g, c, a\}$

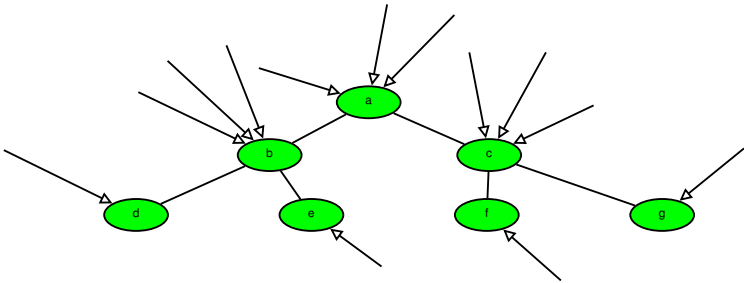


Figura: Pós-ordem

No percurso em pré-ordem sempre que a raíz de uma árvore ou sub-árvore é visitado o mesmo é exibido para em seguida se visitar o próximo nó a esquerda **até que não haja mais nós esquerdos**, depois disso se exibe os nós a direita.
Desenhe o percurso da árvore binária em pré-ordem.

No percurso em pré-ordem sempre que a raíz de uma árvore ou sub-árvore é visitado o mesmo é exibido para em seguida se visitar o próximo nó a esquerda **até que não haja mais nós esquerdos**, depois disso se exibe os nós a direita.
Desenhe o percurso da árvore binária em pré-ordem.
Resposta: $G = \{a, b, d, e, c, f, g\}$

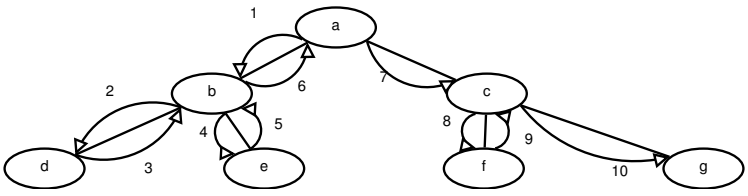


Figura: Pré-ordem

Algoritmos de busca em grafos

AVL - Árvore de busca balanceada - Algoritmos de percurso

```
1  void emOrdem(node* vertice) {
2    if (vertice == NULL) return;
3    emOrdem(vertice->esquerda);
4    printf("%f\n", *vertice->valor);
5    emOrdem(vertice->direita);
6  }
7
8  void preOrdem(node* vertice) {
9    if (vertice == NULL) return;
10   printf("%f\n", *vertice->valor);
11   preOrdem(vertice->esquerda);
12   preOrdem(vertice->direita);
13 }
14
15 void posOrdem(node* vertice) {
16   if (vertice == NULL) return;
17   posOrdem(vertice->esquerda);
18   posOrdem(vertice->direita);
19   printf("%f\n", *vertice->valor);
20 }
```

unesp

Uma árvore binária, como já foi dito, pode degenerar em uma lista ligada sequencial prejudicando em muito o tempo de acesso aos seus dados. Para resolver isso se usa algumas técnicas de **rebalanceamento** da árvore.

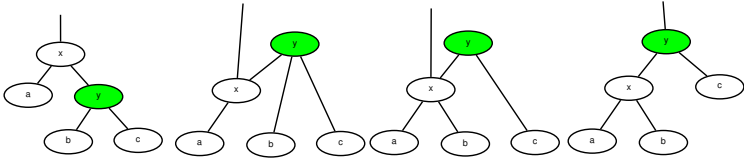


Figura: Rotação anti-horária

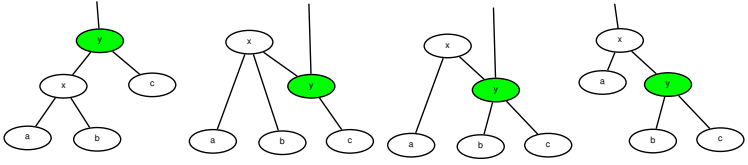


Figura: Rotação horária

Baixe o código de https://github.com/ensismoebius/1955SCC-Projeto-e-Analise-de-Algoritmos.git e abra o projeto "arvoresEListas"lá você encontrará o arquivo *main.cpp*. **Pergunta**: Qual o tempo de execução para adição e remoção de um item na AVL?

Uma Árvore Trie (termo definido em 1960 pelo professor Edward Fredkin a partir da abstração da palavra “retrieval”) pode ser descrita como uma estrutura de dados utilizada para armazenar de forma hierárquica cadeias de caracteres e suportar uma rápida procura de registros, sendo sua principal aplicação a procura por padrões e prefixos.

O grande diferencial das Árvores Trie em relação as, por exemplo, árvores de busca binárias (ABB), é que o valor a ser armazenado (daqui em diante chamaremos esse valor de chave) não precisa ser totalmente guardado em alguma estrutura interna da árvore (nó), pois a própria estrutura da árvore Trie consegue representar senão toda, boa parte da chave.

Organização dos dados em uma trie:

- dados não são guardados em apenas um nó
- dados são tratados como elementos divisíveis dispersos na árvore
- cada nó possui como valor implícito um caractere de um alfabeto pré-definido
- **prefixo de chave** é o caminho desde a raiz até um nó não-folha

O primeiro nível da trie é composto tão e somente por um único vetor, no qual é colocado o primeiro caractere da chave que se pretende armazenar, caso a chave tenha mais de um caractere então será criado dinamicamente um outro vetor em um nível abaixo do primeiro, e, assim por diante até que todos os caracteres da chave sejam mapeados.

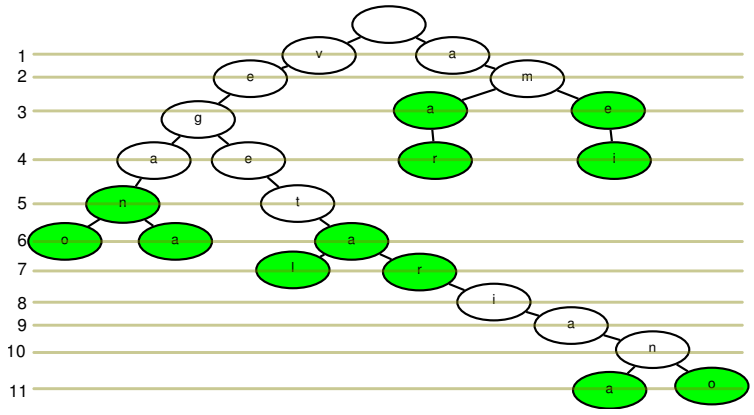


Figura: Uma trie

Nós marcados em verde são as folhas da trie.
Perceba que essa estrutura tem níveis.
O que esses níveis tem de especial?

unesp

Cada nó é representado usando uma estrutura como a abaixo:

```
#ifndef TRIENODE_H
#define TRIENODE_H

#include <vector>

class TrieNode {

  public:
  bool isLeaf;
  static unsigned alphabetSize;
  std::vector<TrieNode*> arrSubTries;

  TrieNode();
  void setAlphabetSize(unsigned alphabetSize);
};
#endif // TRIENODE_H
```

```
#include "trienode.h"

TrieNode::TrieNode()
: isLeaf(false)
{
}

unsigned TrieNode::alphabetSize = 0;

void TrieNode::setAlphabetSize(unsigned alphabetSize
    )
{
  TrieNode::alphabetSize = alphabetSize;
  this->arrSubTries.resize(alphabetSize);
}
```

A trie com suas operações pode ser representada dessa forma:

```
#ifndef TRIE_H
#define TRIE_H

#include <string>
#include "trienode.h"

class Trie
{
  public:
  TrieNode* trieRoot = nullptr; // root of trie
  int amountOfKeysStored; // number of keys in trie

  public:
  Trie(unsigned alphabetSize);
  bool search(const std::string key);
  void insert(std::string key);
  void deleteNode(std::string key);

  private:
  TrieNode* get(TrieNode* x, const std::string key, int trieLevel);
  TrieNode* add(TrieNode* x, const std::string key, int trieLevel);
  TrieNode* deleteNode(TrieNode* x, const std::string key, int trieLevel);
};
#endif // TRIE_H
```

A trie com suas operações pode ser representada dessa forma:

```
#include "trie.h"
#include <stdexcept>

Trie::Trie(unsigned alphabetSize)
: amountOfKeysStored(0)
{
  TrieNode::alphabetSize = alphabetSize;
}

bool Trie::search(const std::string key)
{

  /* Nao sao permitidas buscas nulas */
  if (key.empty())
  throw std::invalid_argument("The string must have some chars");

  /* chama o metodo recursivo get que fara a busca na arvore */
  TrieNode* x = get(trieRoot, key, 0);

  /* Chave nao encontrada */
  if (x == nullptr)
  return false;

  /* se isLeaf == true entao a chave foi encontrada */
  return true;
```

```
26 }
27
28 TrieNode* Trie::get(TrieNode* x, const std::string key, int trieLevel)
29 {
30   /* No nulo a chave nao se encontra na arvore */
31   if (x == nullptr)
32   return nullptr;
33
34   /*
35   * Se a profundidade da arvore (trieLevel) for igual a quantidade de caracteres
36   * (key.length()) entao a chave tem um cadidato a "hit" se este no for uma folha
37   * (isLeaf == true) entao se tem um "hit" se nao se tem um "miss"
38   */
39   if (trieLevel == key.length()) {
40     if (x->isLeaf)
41     return x;
42     else
43     return nullptr;
44   }
45
46   /*
47   * Elemento nao encontrado, recupera o proximo caractere e reinicia as
48   * verificacoes com uma chamada recursiva
49   */
50   char c = key[trieLevel];
51   return get(x->arrSubTries[c], key, trieLevel + 1);
52 }
```

Algoritmos de busca em grafos III

Árvores Trie

```
53
54 void Trie::insert(const std::string key)
55 {
56   /* Nao sao permitidas insercoes nulas */
57   if (key.empty())
58   throw std::invalid_argument("The string must have some chars");
59
60   /* chama o metodo recursivo get que fara a insercao na arvore */
61   trieRoot = add(trieRoot, key, 0);
62 }
63
64 TrieNode* Trie::add(TrieNode* x, const std::string key, int trieLevel)
65 {
66   /*
67   * Se o no e nulo entao a chave completa nao esta na arvore, e necessario criar
68   * um novo no para alocar tal caractere, perceba que este no deve ser do tamanho
69   * do alfabeto adotado
70   */
71   if (x == nullptr) {
72     x = new TrieNode();
73     x->setAlphabetSize(TrieNode::alphabetSize);
74   }
75
76   /*
77   * Se a profundidade da arvore (trieLevel) for igual a quantidade de caracteres
78   * (key.length()) entao a chave esta completa dentro da arvore;
79   */
```

Algoritmos de busca em grafos IV

Árvores Trie

```
80  if (trieLevel == key.length()) {
81
82    /*
83    * Ha que se verificar se a chave encontrada NAO e uma folha. Caso NAO seja
84    * incrementa o indicador de quantidade de chaves armazenadas
85    */
86    if (!x->isLeaf) {
87      this->amountOfKeysStored++;
88    }
89
90    /* Marca como folha */
91    x->isLeaf = true;
92
93    /* Retorna a folha inserida */
94    return x;
95  }
96
97  /*
98  * Se a profundidade da arvore (trieLevel) ainda nao for igual a quantidade de
99  * caracteres (key.length()) entao prepara uma chamada recursiva para
100 * verficacao/criacao de um novo no
101 */
102 char c = key[trieLevel];
103 x->arrSubTries[c] = add(x->arrSubTries[c], key, trieLevel + 1);
104
105 /* Retorna a folha inserida */
106 return x;
```

Algoritmos de busca em grafos V

Árvores Trie

```
107 }
108
109 void Trie::deleteNode(const std::string key)
110 {
111
112   /* Nao sao permitidas insercoes nulas */
113   if (key.empty())
114     throw std::invalid_argument("The string must have some chars");
115
116   /* chama o metodo recursivo get que fara a remocao na arvore */
117   trieRoot = deleteNode(trieRoot, key, 0);
118 }
119
120 TrieNode* Trie::deleteNode(TrieNode* x, std::string key, int trieLevel)
121 {
122
123   /* No nulo a chave nao se encontra na arvore */
124   if (x == nullptr)
125   return nullptr;
126
127   /*
128   * Se a profundidade da arvore (trieLevel) for igual a quantidade de caracteres
129   * (key.length()) entao a chave tem um cadidato a remocao.
130   */
131   if (trieLevel == key.length()) {
132     /*
133     * Se este no for uma folha (isLeaf == true) entao o no e desmarcado como
```

```
134     * folha e a contagem de item da trie e decrementada
135     */
136     if (x->isLeaf)
137     amountOfKeysStored--;
138
139     x->isLeaf = false;
140   } else {
141
142     /*
143     * Se a profundidade da arvore (trieLevel) ainda nao for igual a quantidade de
144     * caracteres (key.length()) entao prepara uma chamada recursiva para
145     * verficacao/remocao de proximo no
146     */
147     char c = key[trieLevel];
148     x->arrSubTries[c] = deleteNode(x->arrSubTries[c], key, trieLevel + 1);
149   }
150
151   /*
152   * Na volta das chamadas recursivas, no momento em que uma folha for encontrada a
153   * delecao da arvore e terminada retornando-se um no valido
154   */
155   if (x->isLeaf)
156   return x;
157
158   /*
159   * Se o no nao e valido, anula todos os nos abaixo deste e, em seguinda, retorna
160   * um endereco nulo para o nivel acima
```

Algoritmos de busca em grafos VII

Árvores Trie

```
161   */
162   for (int i = 0; i < TrieNode::alphabetSize; i++)
163   if (x->arrSubTries[i] != nullptr)
164   return x;
165
166   delete x;
167   return nullptr;
168 }
```

Exercício 1542660005445454545877: Usando a trie crie um sistema que dado o **prefixo** de uma palavra o mesmo retorne todas as palavras cadastradas com aquele prefixo. Por exemplo: "par" $\rightarrow$ **par**angaricutirimiruaro, **par**aná, **par**cimônia, etc.
Considerando que o tamanho da entrada n é a quantidade de letras na palavra, qual o tempo de execução para **localizar uma palavra**?
E quando a palavra não é encontrada?

Exercício 1542660005445454545877: Usando a trie crie um sistema que dado o **prefixo** de uma palavra o mesmo retorne todas as palavras cadastradas com aquele prefixo. Por exemplo: "par" $\rightarrow$ **par**angaricutirimiruaro, **par**aná, **par**cimônia, etc.
Considerando que o tamanho da entrada n é a quantidade de letras na palavra, qual o tempo de execução para **localizar uma palavra**?
E quando a palavra não é encontrada?

Resposta:
$\Theta(n)$ e $O(n), \Omega(1)$

Análise de problemas P / NP

Análise de problemas P / NP

Inicialmente definiremos as classes de problemas P/NP para problemas de **decisão**, ou seja, problemas cujas respostas são **sim/não**
No entanto, **pode ser** relativamente fácil converter um problema de otimização em um de decisão.
Por exemplo:
Existe um caminho de distância K entre dois pontos de um grafo? A resposta será sim ou não.

De forma mais genérica: Existe uma solução que satisfaz uma certa propriedade?

No entanto podemos estimar um valor para K e ir perguntando ao algoritmo que, caso responda **não**, poderá responder sim para $K + j$ ou $K - j$ tal que j seja um valor que faça com que exista uma solução. A forma de encontrar esse j pode variar de acordo com a situação. Ou seja, associado a um problema de decisão pode estar vinculado um problema de **otimização**.

unesp

Algoritmos polinomiais são aqueles que podem ter seu **tempo representado na forma de um polinômio** de grau qualquer, basicamente quase todos os que vimos até agora, por outro lado, algoritmos não polinomiais são aqueles cuja a representação é dada, por exemplo, por funções exponenciais.

A verdade é que não se pode ter **total certeza** que um algoritmo é não polinomial pois pode ser que, para um determinado problema, exista uma solução polinomial portanto, o que se pode afirmar é que **até o momento** se sabe que a solução desses problemas tem algoritmos de tempo não polinomial.

Dizemos que um algoritmo resolve um dado problema se, ao receber uma entrada do problema, devolve uma solução ou informa que não há solução.
Se diz que um algoritmo é **razoavelmente rápido** quando o mesmo **tem um tempo polinomial**.
Um algoritmo é polinomial se **o pior caso** do mesmo for algo como $O(n^i) \forall i \in \mathbb{N}$
Exemplos:

- $10n^3 + 3n^2 + 1 \implies O(n^3)$
- $n^5 . \log n \leq n^6 \implies o(n^6)$

É importante separar os algoritmos polinomiais dos não polinomiais pois polinômios tem algumas características interessantes como fechamento em **adição, multiplicação e composição** que, quando aplicadas **não mudam** a natureza polinomial do tempo do algoritmo.

São todos aqueles que consomem um tempo **não polinomial** como, por exemplo, e^n, $n!$, 2^n, etc.
Geralmente a resolução desses problemas é custosa e se dá pela estratégia da **força bruta**. É verdade que, ainda assim, é possível economizar alguns ciclos de processamento usando uma **árvore de estados** mas isso não muda o tempo assintótico do algoritmo.

A classe P de problemas compreende todos os algoritmos que podem ser executados em um tempo polinomial, **o fato de ainda não se ter encontrado um algoritmo polinomial para um problema não garante definitivamente que a natureza do problema seja não polinomial**.

Dada uma fórmula booleana na forma normal **conjuntiva** ou **disjuntiva**. Existem valores para $x_1, x_2, x_3, \ldots x_n$ que resultam em **Verdadeiro** para essa fórmula?
Exemplos de instâncias:

1. conjuntiva: $(x_1 \vee \neg x_2) \wedge (x_1 \vee x_3) \wedge (\neg x_1 \vee x_2)$
2. disjuntiva: $(x_1 \wedge \neg x_2) \vee (x_1 \wedge x_3) \vee (\neg x_1 \wedge x_2)$

À **solução** dessa fórmula daremos o nome de **certificado**.
O **certificado** garante que, dado o problema, teremos a resposta **sim**. De outra forma: O certificado é a **prova** de que a resposta sim é verdadeira.
Para o exemplo 1 o certificado é $x_1 = \textit{Verdadeiro}, x_2 = \textit{Verdadeiro}$

Pergunta: Existe um **certificado** para o problema 2? Qual?
Considerando o problema da satisfazibilidade entremos no campo dos problemas **NP**. Mais especificamente **NP-completo**.

A classe *NP* é aquela dos algoritmos não polinomiais certo? **Errado!**
NP é um acrônimo para *Non-deterministic polynomial time*, ou seja, é uma classe de problemas cuja validade de um certificado pode ser avaliada em tempo **polinomial** e cuja a solução pode ser obtida em tempo polinomial usando um sistema de computação **hipotético** não determinístico (o que se aproxima mais disso são os atuais computadores quânticos).
De outra forma: Dada uma solução hipotética para um problema NP é rápido e fácil (tempo polinomial) verificar se aquela solução é **verdadeira**. Já, descobrir uma solução para o problema, bom... aí é outra estória pois esse tipo de problema **não tem algoritmos conhecidos** cujo tempo de geração de um certificado seja polinomial.
Daí tiramos que $P \subset NP$.
Obs: **Não se sabe** se $P = NP$ ou não, quem conseguir provar isso (que $P = NP$ ou $P \neq NP$) revolucionará a área da computação.

Considerando que qualquer problema que possa, de alguma forma, ser transformado no problema da satisfazibilidade ou em qualquer outro problema NP **também é NP**, afim de provar que $P = NP$ ou $P \neq NP$ basta provar isso para **um** único problema NP!

A página do professor *Paulo Feofiloff* tem muitos exemplos:
https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/NPcompleto.html

Da definição anterior tiramos que: Seja A um problema e B outro cuja solução é conhecida. Se a conversão de A para B é possível então solucionar A também o será. Ao procedimento descrito acima damos o nome de redução polinomial de A para B ou $A \preceq B$ (lê-se A é redutível a B) onde B é um algoritmo polinomial
É importante dizer que $B \subseteq A$ e B pode ser chamado múltiplas vezes em A.

Exemplo: *Selecione o enésimo termo de uma sequência numérica randômica.*
Tal problema pode ser reduzido a uma de ordenação seguida de uns poucos comandos. Sendo assim, suponha que $A \preceq B$ então:

- se $B \in P \implies A \in P$
- se $B \preceq C \implies A \preceq C$
- se $B \in P \implies A \in P$
- se $B \in NP$ **talvez** haja uma solução de $A \in P$ que não use B.
- se A não tem solução polinomial e $A \preceq B / B \in NP \implies A \in NP$

unesp

Então se $\mathbf{NP-completo} \subset NP, A \in NP, B \preceq A \forall B \in NP \implies A \in \mathbf{NP-completo}$

Embora NP (e NP-completo) de tenha como domínio os problemas de decisão, **como ja vimos** existe uma correlação entre problemas de otimização e de decisão que garante que o problema de decisão correspondente ao de otimização deve ser pelo menos **não mais difícil** que o de otimização.
De outra forma: Se um problema de otimização é polinomial sua respectiva decisão também o será, **o inverso não é necessariamente verdade**.
Finalmente problemas NP-completos são aqueles que podem de alguma forma serem reduzidos ao problema da satisfazibilidade (meh...)

unesp

Análise de problemas P / NP

Exercício 0

Crie um programa que:

- verifique se um número é primo ou não
- gere o enésimo numero primo dado um valor de n

Determine o tempo de execução de ambos
Algum dos dois é NP?
Qual a relação entre eles?

- Determine um circuito hamiltoniano de custo mínimo.
- Dado um valor L existe um circuito de custo L?

Determine o tempo de execução de ambos
Algum dos dois é NP?
Qual a relação entre eles?

Um *circuito hamiltoniano* é um circuito/ciclo cujos elementos não se repetem.

- Determine quantas cliques com 2 e 3 elementos um certo grafo pode ter.
- Dado uma quantidade e elementos k existe um clique com k elementos?

Determine o tempo de execução de ambos
Algum dos dois é NP?
Qual a relação entre eles?

Uma *clique* é um subgrafo completo.

- Dado um grafo o mesmo é euleriano?

Determine o tempo de execução
Esse algoritmo é NP?

Um grafo euleriano é aquele em que é possível percorrer todas as arestas do mesmo podendo apenas repetir os vértices. Grafos eulerianos tem todos os vértices com grau par.

Análise de problemas P / NP

Exercício 4

Considere $S \subseteq \mathbb{Z}/k \in S, |S| = n, n \in \mathbb{N}$, ou seja, S é subconjunto de $\mathbb{Z}$ e a quantidade de elementos de S é igual a um n qualquer natural, então: Existe $\sum_{i \in S} k_i = \sum_{i \notin S} k_i$?

unesp

Algoritmos de força bruta

unesp

Figura: Processei pra crl! Vai dar sim!

Algoritmos de força bruta

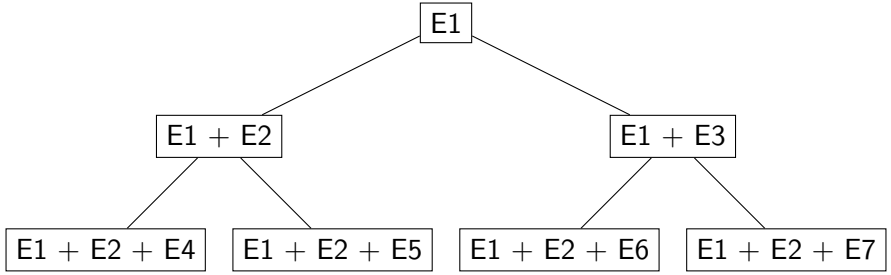
São aqueles cuja a definição não se pode expressar em termos polinomiais ou cujos tempos são muito grandes, geralmente são feitos para mapear todas as opções dentro de um certo espaço de procura resultando assim em ordens de tempo muito grandes. Geralmente tais algoritmos estão ligados a um pensamento mais direto de resolução do problema e dependente do poder de processamento do computador e não da inteligência na modelagem.

Alguns dos exemplos que já vimos são o **burbble-Sort**, **Selection-Sort**, além de outros como **Busca sequencial**, **multiplicação de matrizes**, etc.

No entanto, existem técnicas como o **bactracking** ou **retro-rastreamento** e **branching-and-bound** ou **ramificar e limitar** que são estratégias que usam uma **árvores de estados** e podem melhorar um pouco tais tempos de execução.

unesp

Esta árvore indica em cada um dos seus nós um estado alcançado pelo sistema, assim, é possível, considerando-se certas condições, julgar a possibilidade ou a utilidade de se fazer novos testes, ou ainda, se é necessário voltar a algum estado anterior. Existem técnicas como o **backtracking** ou **retro-rastreamento** e **branching-and-bound** ou **ramificar e limitar** que são estratégias que usam uma **árvores de estados** e podem melhorar um pouco os tempos de execução de algoritmos de força bruta.



unesp

Dado um problema de otimização, se o custo de uma solução possível ultrapassou o valor de outra então se pode parar como o processamento daquela possibilidade.

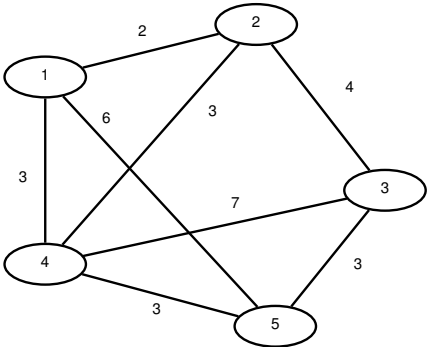


Figura: Caixeiro viajante

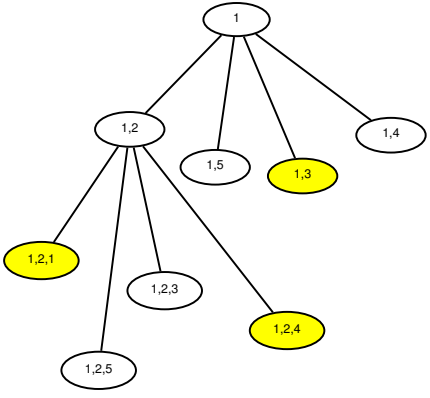


Figura: Árvore de estados

unesp

Posicione 4 rainhas em um tabuleiro 4x4 sem que elas se ataquem. Monte a árvore de estados com Retro-rastreamento e Ramificar e limitar.