PROJETO E ANÁLISE DE ALGORITMOS

PARTE III

ALGORITMOS RECURSIVOS

1. Recursividade

- Um algoritmo recursivo é aquele que direta ou indiretamente chama a si próprio.

- Procedimentos recursivos permitem definir um número infinito de instruções através de uma rotina finita.

- Durante a execução de um procedimento recursivo, diversas ativações são empilhadas na pilha do sistema, ocupando memória e gastando tempo. Isto é uma limitação para o uso da recursividade em programas.

Quando empregar recursão:

- O problema está definido de forma recursiva
- A profundidade da recursão é relativamente pequena
- A conversão do algoritmo para a forma iterativa é difícil (exigindo memória auxiliar)
- A rotina não constitui parte crítica do programa

Exemplo: fatorial recursivo

```
int fat (int n) {
   if (n<=0)
      return (1);
   else
      return (n * fat(n-1));
}
int main() {
   int f;
   f = fat(5);
   printf("%d",f);
   return (0);
}
```

fat(5) = 5 * fat(4)

fat(4) = 4 * fat(3)

fat(3) = 3 * fat(2)

fat(2) = 2 * fat(1)

fat(1) = 1 * fat(0)

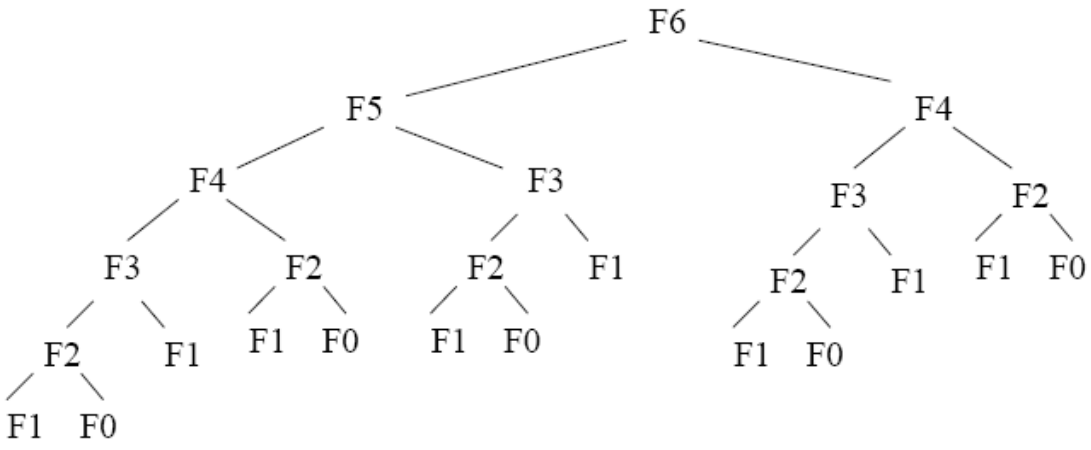
fat(0) = 1

Exemplo: série de Fibonnaci

- Série de Fibonnaci:
  - $F_n = F_{n-1} + F_{n-2}$ $\quad n > 2,$
  - $F_0 = F_1 = 1$
  - 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89...

```
int Fib(int n) {
   if (n<2)
      return (1);
   else
      return (Fib(n-1) + Fib(n-2));
}
```

- Termos F_{n-1} e F_{n-2} são computados independentemente
- Número de chamadas recursivas = número de Fibonacci!
- Custo para cálculo de F_n
  - $O(\phi^n)$ onde $\phi = (1 + \sqrt{5})/2 = 1{,}61803...$
  - Exponencial!!!



Fibonacci iterativo é linear!

2. Análise de Algoritmos Recursivos

a) Equações de Recorrência

- Para cada procedimento recursivo é associada uma função de complexidade *f(n)* desconhecida, onde *n* mede o tamanho dos argumentos para o procedimento.

- Obtemos uma equação de recorrência para *f(n)*.

- **Equação de recorrência**: maneira de definir uma função por uma expressão envolvendo a mesma função.

- Técnicas de solução:
  - Expansão telescópica
  - Árvore de recorrência
  - Método de substituição
  - Teorema mestre

Exemplo:

```
void Pesquisa(n) {
(1)  if (n <= 1) {
(2)    inspecione_elemento(n);  return; }
     else {
(3)    for (int i=0; i<n; i++) inspecione_elemento(i);
(4)    Pesquisa(n/3);
     }
}
```

Análise do Procedimento Recursivo:

- Seja *T(n)* uma função de complexidade que represente o número de inspeções nos *n* elementos do conjunto.
- O custo de execução das linhas (1) e (2) é $\Theta(1)$ e o da linha (3) é exatamente n.
- Usa-se uma equação de recorrência para determinar o no de chamadas recursivas.
- O termo *T(n)* é especificado em função dos termos anteriores *T(1), T(2), . . . , T(n-1).*
- *T(n) = n + T(n/3); T(1) = 1* (para n = 1 fazemos uma inspeção)
- Por exemplo, *T(3) = T(3/3) + 3 = 4, T(9) = T(9/3) + 9 = 13*, e assim por diante.

b) Método da Expansão Telescópica

- Substituem-se os termos *T(k), k < n*, até que todos os termos *T(k), k > 1*, tenham sido substituídos por fórmulas contendo apenas *T(1)*:

  Passo i=0: $T(n) = n + T(n/3)$
  Passo i=1: $T(n/3) = n/3 + T(n/3/3)$
  Passo i=2: $T(n/3/3) = n/3/3 + T(n/3/3/3)$
  ...
  Passo i=x: $T(n/3/3 \ldots /3) = n/3/3 \ldots /3 + T(n/3 \ldots /3)$

- Adicionando lado a lado, temos

  $T(n) = n + n(1/3) + n(1/3^2) + \ldots + n(1/3^x) + T(n/3^{x+1})$
  Passo 0 1 2 ... x

  que representa a soma de uma série geométrica de razão 1/3, multiplicada por *n*, e adicionada de *T(n/3/3 ... /3)*, que é 1 quando o valor entre parênteses for menor ou igual a 1.

- $T(n/3^{x+1})$ será igual a 1 quando $n/3^{x+1}=1$. Logo, $x=\log_3 n-1$. Lembrando que T(1)=1, temos que

$$T(n) = 1 + n \sum_{i=0}^{\log_3 n-1} \frac{1}{3^i} = 1 + \frac{3n}{2}\left(1 - \frac{1}{n}\right) = \frac{3n-1}{2}$$

- Logo, o programa do exemplo é $\Theta(n)$

Exemplo: MergeSort

Utiliza uma estratégia do tipo *Divisão e Conquista*:

1. Divisão do problema em *n* sub-problemas
2. Conquista: resolução dos sub-problemas:
   - trivial se tamanho muito pequeno.
   - Utiliza-se a mesma estratégia para tamanho grande.
3. Combinação das soluções dos sub-problemas

MergeSort:

1. Divisão do vetor em dois vetores de dimensões similares
2. Conquista: ordenação recursiva dos dois vetores usando MergeSort; Nada a fazer para vetores de dimensão 1.
3. Combinação: fusão dos dois vetores ordenados. Será implementado por uma função auxiliar merge.

Função merge:

1. Recebe as duas sequências *A[p..q]* e *A[q+1..r]* já ordenadas.
2, No fim da execução da função, a sequência *A[p..r]* estará ordenada.

```
void merge(int A[], int p, int q, int r) {
  int L[MAX], R[MAX];
  int n1 = q-p+1;
  int n2 = r-q;
  for (i=1 ; i<=n1 ; i++) L[i] = A[p+i-1];
  for (j=1 ; j<=n2 ; j++) R[j] = A[q+j];
  L[n1+1] = MAXINT; R[n2+1] = MAXINT;
  i = 1; j = 1;
  for (k=p ; k<=r ; k++)
     if (L[i] <= R[j]) A[k] = L[i++];  else  A[k] = R[j++];
}
```

- Passo básico: comparação dos dois valores contidos nas primeiras posições de ambos os vetores, colocando o menor dos valores no vetor final.

- Cada passo básico é executado em tempo constante

- A dimensão da entrada é $n = r - p + 1$, e nunca poderá haver mais do que n passos básicos.

- Este algoritmo executa então em tempo linear : merge = $\Theta(n)$.

Qual o papel das sentinelas de valor MAXINT?

MergeSort:

```
void merge_sort(int A[], int p, int r)
{
   if (p < r) {
      q = (p+r)/2;
      merge_sort(A,p,q);
      merge_sort(A,q+1,r);
      merge(A,p,q,r);
  }
}
```

- Invocação inicial: merge_sort(A,1,n); em que *A* contém *n* elementos.

- Qual a dimensão de cada sub-sequência criada no passo de divisão?

Análise de Pior Caso:

- Simplificação da análise de "merge sort": tamanho da entrada é uma potência de 2. Em cada divisão, as sub-sequências têm tamanho exatamente *n/2*.

- Seja *T(n)* o tempo de execução (no pior caso) sobre uma entrada de tamanho *n*.

- Se *n* = 1, esse tempo é constante, que escrevemos $T(n)=\Theta(1)$.

- Senão:
  1. Divisão: o cálculo da posição do meio do vetor é feita em tempo constante: $D(n) = \Theta(1)$;
  2. Conquista: são resolvidos dois problemas, cada um de tamanho *n/2*; o tempo total para isto é $2T(n/2)$
  3. Combinação: a função merge executa em tempo linear: $C(n) = \Theta(n)$

- Desta forma:

$$T(n) = \Theta(1), \text{ se } n = 1;$$
$$= \Theta(1) + 2T(n/2) + \Theta(n), \text{ se } n > 1$$

- Considerando o número de comparações como operação crítica:

$$T(n) = 0, \text{ se } n = 1;$$
$$= 2T(n/2) + n, \text{ se } n > 1$$

Solução por expansão telescópica:

T(n) = 0, se n = 1;
= 2T(n/2) + n, se n > 1

Passo i=0: $T(n) = 2T(n/2) + n$
Passo i=1: $T(n/2) = 2T(n/4) + n/2$
Passo i=2: $T(n/4) = 2T(n/8) + n/4$
...
Passo i=x: $T(n/2^x) = 2T(n/2^{x+1}) + n/2^x$

- Para usarmos a expansão, precisamos igualar os coeficientes das chamadas:

Passo i=0: $T(n) = 2T(n/2) + n$
Passo i=1: $2T(n/2) = 4T(n/4) + 2n/2$
Passo i=2: $4T(n/4) = 8T(n/8) + 4n/4$
...
Passo i=x: $2^xT(n/2^x) = 2^{x+1}T(n/2^{x+1}) + n$

- A recursão termina quando o tamanho do vetor for 1:

$1=n/2^{x+1} \Rightarrow x=\lg n - 1$

- Logo:

$$\boldsymbol{T(n) = 0 + \sum_{i=0}^{lgn-1} n = nlgn} = \Theta(n\log n)$$

c) Árvores de Recorrência

- Cada nodo da árvore contém o custo daquele nível de recursão.

- Chamadas recursivas são filhos do nodo.

- Nodos folha contêm o custo da condição de término da recorrência.

- O custo final é a soma dos custos dos nodos. A altura da árvore deve ser calculada.

d) Teorema Mestre

- Recorrências das formas

$$T(n) = aT(n/b) + f(n)$$

  onde $a \geq 1$ e $b > 1$ são constantes e $f(n)$ é uma função assintoticamente positiva, podem ser revolvidas usando o Teorema Mestre

- Neste caso, não estamos achando a forma fechada da recorrência mas sim seu comportamento assintótico

Teorema: Sejam as constantes $a \geq 1$ e $b > 1$ e $f(n)$ uma função definida nos inteiros não-negativos pela recorrência:

$$T(n) = aT(n/b) + f(n)$$

onde a fração n/b pode significar $\lfloor n/b \rfloor$ ou $\lceil n/b \rceil$. A equação de recorrência T(n) pode ser limitada assintoticamente da seguinte forma:

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

Comentários

- Nos três casos estamos comparando a função $f(n)$ com a função $n^{\log_b a}$. Intuitivamente, a solução da recorrência é determinada pela maior das duas funções. Por exemplo:
  - No primeiro caso a função $n^{\log_b a}$ é a maior e a solução para a recorrência é $T(n) = \Theta(n^{\log_b a})$
  - No terceiro caso, a função $f(n)$ é a maior e a solução para a recorrência é $T(n) = \Theta(f(n))$
  - No segundo caso, as duas funções são da mesma ordem. Neste caso, a solução fica multiplicada por um fator logarítmico: $T(n) = \Theta(n^{\log_b a} \log n) = \Theta(f(n) \log n)$

- No primeiro caso, a função *f*(*n*) deve ser não somente menor que $n^{\log_b a}$ mas ser polinomialmente menor. Ou seja, *f*(*n*) deve ser assintoticamente menor que $n^{\log_b a}$ por um fator de n^{ε}, para alguma constante $\varepsilon > 0$.
- No terceiro caso, a função *f*(*n*) deve ser não somente maior que $n^{\log_b a}$ mas ser polinomialmente maior e satisfazer a condição de "regularidade" que $af(n/b) \le cf(n)$. Esta condição é satisfeita pela maior parte das funções polinomiais encontradas neste curso.
- O Teorema **não** cobre todas as possibilidades para *f*(*n*).

Exemplo 1

$$T(n) = 9T(n/3) + n$$

Temos que,

$$a = 9, \quad b = 3, \quad f(n) = n$$

Desta forma,

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

Como $f(n) = O(n^{\log_3 9 - \varepsilon})$, onde $\varepsilon = 1$, podemos aplicar o caso 1 do teorema e concluir que a solução da recorrência é $T(n) = \Theta(n^2)$.

Exemplo 2

$$T(n) = T(2n/3) + 1$$

Temos que,

$$a = 1, \quad b = 3/2, \quad f(n) = 1$$

Desta forma,

$$n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$$

O caso 2 se aplica já que $f(n) = O(n^{\log_b a}) = \Theta(1)$. Temos então que a solução da recorrência é $T(n) = \Theta(\log n)$.

Exemplo 3

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

Temos que,

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

Desta forma,

$$n^{\log_b a} = n^{\log_4 3} = O(n^{0,793})$$

Como $f(n) = \Omega(n^{\log_4 3+\varepsilon})$, onde $\varepsilon \approx 0{,}2$, o caso 3 se aplica se mostrarmos que a condição de regularidade é verdadeira para *f*(*n*).

Para um valor suficientemente grande de *n*

$$af(n/b) = 3(n/4)\log(n/4) \leq (3/4)n\log n = cf(n)$$

Para *c* = ¾. Consequentemente, usando o caso 3, a solução para a recorrência é: $T(n) = \Theta(n\log n)$.

3. Técnica de projeto: Força Bruta

Exemplo: Clique

- Considere um conjunto P de n pessoas e uma matriz M de tamanho n x n, tal que M[i][j] = M[j][i] = 1, se as pessoas i e j se conhecem e M[i][j] = M[j][i] = 0, caso contrário
- Problema: existe um subconjunto C (Clique), de r pessoas escolhidas de P, tal que qualquer par de pessoas de C se conhecem?
- Solução usando força bruta: verificar, para todas as combinações simples (sem repetições) C de r pessoas escolhidas entre as n pessoas do conjunto P, se todos os pares de pessoas de C se conhecem.

```
#include<iostream>
using namespace std;

void combinacao(int n, int r, int x[], int next, int k){
   int i;
   if (k == r){
      for (i = 0; i < r; i++)
         cout<<x[i]+1<<" ";
      cout<<endl;
   } else {
      for (i = next; i < n; i++) {
         x[k] = i;
         combinacao(n,r,x,i+1,k+1);
      }
   }
}

int main () {
   int n, r, x[100];
   cout<<"Entre com o valor de n: ";
   cin>>n;
   cout<<"Entre com o valor de r: ";
   cin>>r;
   combinacao(n,r,x,0,0);
   return 0;
}
```

4. Técnica de projeto: Transformar e conquistar

2.

Exemplo: HeapSort

- Um *Heap* é uma estrutura de dados parcialmente ordenada que é adequada para a implementação de filas de prioridade.

- Fila de prioridade é um conjunto de itens com uma característica ordenável chamada prioridade, contendo as seguintes operações:

    - Encontrar um elemento com a prioridade mais alta
    - Remover um elemento com a prioridade mais alta
    - Adicionar um novo item ao conjunto

Definição: Um heap é uma árvore binária essencialmente completa com chaves atribuídas aos seus nós, onde a chave de um nó é maior ou igual a chave dos seus nós-filhos.

Propriedades:

a) A altura de um heap com n nós é $\lfloor \lg n \rfloor$
b) A raiz do heap sempre contém o maior elemento
c) Cada sub-árvore é também um heap

Um heap pode ser implementado eficientemente como um arranjo:

- Os filhos de um nodo na posição *i* estão nas posições *2i* e *2i+1*
- O pai de um nodo na posição *i* está na posição *i/2.*

O uso da estrutura heap permite que:

- O elemento máximo do conjunto seja determinado e corretamente posicionado no vetor em tempo constante, trocando-se o primeiro elemento do heap com o último.
- O trecho restante do vetor (do índice 1 ao $n-1$), que pode ter deixado de ter a estrutura de heap, volte a tê-la com número de trocas de elementos proporcional à altura da árvore.

O algoritmo Heapsort consiste da construção de um heap seguida de sucessivas trocas do primeiro com o último elemento e rearranjos do heap:

Heapsort(A)

- Entrada: Vetor A de *n* números inteiros.
- Saída: Vetor A ordenado.

1. ConstroiHeap(A, n)
2. Para ultimo:=n até 2 faça
◁ t := A[ultimo]
A[ultimo] := A[1]
A[1] := t
◁ AjustaHeap(A, 1, ultimo)

AjustaHeap(A, i , n)

- Entrada: Vetor A de *n* números inteiros com estrutura de heap, exceto, talvez, pela sub-árvore de raiz *i*
- Saída: Vetor A com estrutura de heap

se $2i \leq n$ e $A[2i] \geq A[i]$ então maximo := 2i senão maximo := i
se $2i + 1 \leq n$ e $A[2i + 1] \geq A[maximo]$ então maximo := 2i + 1
se maximo <> i então
t := A[maximo]
A[maximo] := A[i]
A[i] := t
AjustaHeap(A, maximo, n)

Análise: (em função da altura da árvore, h)

Quantas comparações e quantas trocas são executadas no pior caso na etapa de ordenação do algoritmo Heapsort ?

- A seleção e o posicionamento do elemento máximo são feitos em tempo constante.
- No pior caso, a função AjustaHeap efetua $\Theta(h)$ comparações e trocas, onde h é a altura do heap que contém os elementos que restam ordenar.
- Como o heap representa uma árvore binária completa, então $h \in \Theta(\log i)$, onde i é o número de elementos do heap na i-ésima iteração.
- Logo, a complexidade da etapa de ordenação do Heapsort é:

$$\sum_{i=2}^{n} \log i \leq \sum_{i=1}^{n} \log n = n \log n = O(n \log n)$$

- Na verdade, $\Sigma \log i \in \Theta(n \log n)$.
- No entanto, também temos que computar a complexidade de construção do heap

Mas, como construímos o heap ?

Se o trecho de 1 a i do vetor tem estrutura de heap, é fácil adicionar a folha $i + 1$ ao heap e em seguida rearranjá-lo, garantindo que o trecho de 1 a $i + 1$ tem estrutura de heap. Esta é a abordagem top-down para construção do heap.

Construção do Heap (top-down):

- Entrada: Vetor A de n números inteiros.
- Saída: Vetor A com estrutura de heap.

```
para i:=2 até n faça
        v := A[ i ]
        j := i
        enquanto j > 1 e A[j / 2] < v faça
                A[j ] := A[j / 2]
                j := j / 2
        A[j ] := v
```

Análise (comparações e trocas no pior caso):

- O rearranjo do heap na iteração i efetua $\Theta(h)$ comparações e trocas no pior caso, onde h é a altura da árvore representada pelo trecho do heap de 1 a i. Logo, $h \in \Theta(\log i)$.

- Portanto, o número de comparações e trocas efetuadas na construção do heap por esta abordagem é $\Sigma_i \log i \in \Theta(n \log n)$

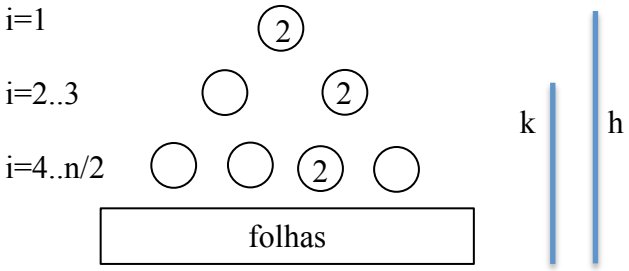
É possível construir o heap de forma mais eficiente. Suponha que o trecho de i a n do vetor é tal que, para todo j , $i \leq j \leq n$, a sub-árvore de raiz j representada por esse trecho do vetor tem estrutura de heap.

Note que, em particular, o trecho de $\lfloor n/2 \rfloor + 1$ a n do vetor satisfaz a propriedade, pois inclui apenas folhas da árvore binária de n elementos.

Construção do Heap (bottom-up):

- Entrada: Vetor A de *n* números inteiros.
- Saída: Vetor A com estrutura de heap.

para i:=n/2 até 1 faça AjustaHeap(A,i,n)



Análise:

Quantas comparações são executadas no pior caso na construção do heap pela abordagem bottom-up ?

- O rearranjo do heap na iteração *i* efetua *2k* comparações no pior caso, onde *k* é a altura da sub-árvore de raiz *i*.
- A altura de um nodo varia entre 1 e h=lg(n+1)-1
- Em cada nível de altura *k* temos 2^{h-k} nodos
- A função de custo para o número de comparações será então da forma:

$$f(n) = \sum_{k=1}^{h} 2k2^{h-k} = 2n - 2\lg(n+1)$$

- Logo, $f(n) \in \Theta(n)$ e a abordagem bottom-up para construção do heap apenas efetua $\Theta(n)$ comparações e trocas no pior caso.

Assim, a complexidade do Heapsort no pior caso é $\Theta(n \log n)$.

Um parênteses: O limite inferior para a classe de problemas de ordenação por comparação de chaves

O Heapsort usa $\Theta(n \log n)$ comparações no pior caso. Será possível projetar um algoritmo de ordenação baseado em comparações ainda mais eficiente?

Não! É possível provar que qualquer algoritmo que ordena n elementos, baseado apenas em comparações, efetua no mínimo $\Omega(n \log n)$ comparações no pior caso.

Para demonstrar esse fato, vamos representar os algoritmos de ordenação em um modelo computacional abstrato, denominado árvore (binária) de decisão:

- Os nós internos de uma árvore de decisão representam comparações feitas pelo algoritmo.
- As sub-árvores de cada nó interno representam possibilidades de continuidade das ações do algoritmo após a comparação.
- No caso das árvores binárias de decisão, cada nó possui apenas duas sub-árvores.
- As folhas são as respostas possíveis do algoritmo após as decisões tomadas ao longo dos caminhos da raiz até as folhas.

Considere a seguinte definição alternativa do problema da ordenação:

Dado um conjunto de *n* inteiros $x_1 \ldots x_n$, encontre uma permutação *p* dos índices $1 \leq i \leq n$ tal que $x_{p(1)} \leq x_{p(2)} \leq \ldots \leq x_{p(n)}$.

É possível representar um algoritmo para o problema da ordenação através de uma árvore de decisão da seguinte forma:

- Os nós internos representam comparações entre dois elementos do conjunto, digamos $x_i \leq x_j$.
- As ramificações representam os possíveis resultados da comparação: verdadeiro se $x_i \leq x_j$, ou falso se $x_i > x_j$.
- As folhas representam possíveis soluções: as diferentes permutações dos *n* índices.

Ao representarmos um algoritmo de ordenação qualquer baseado em comparações por uma árvore binária de decisão, todas as permutações de *n* elementos devem ser possíveis soluções. Assim, a árvore deve ter pelo menos *n!* folhas.

O caminho mais longo da raiz a uma folha representa o pior caso de execução do algoritmo.

A altura mínima de uma árvore binária de decisão com pelo menos *n!* folhas dá o número mínimo de comparações realizadas: $\log n! \in \Omega(n \log n)$.

Logo, o heapsort é um algoritmo ótimo.

5. Técnica de projeto: Decrementar e conquistar

Exemplo: Geração de permutações

Gerar todas as permutações dos elementos de um vetor.

Solução: Fixar um elemento e gerar todas as (n-1)! permutações com os elementos 2..n do vetor.

Solução:

Permutação (v, i, n)

```
se i=n então imprima(v,n)
        senão para k:=1 até n-i+1 faça
            Permutação (v,i+1,n)
            Rotaciona (v,i,n)
```

Análise em função do número de impressões:

- Faça uma mudança de variável: m=n-i+1
- Atenção na expansão telescópica

Obs: O algoritmo de Johnson-Trotter evita ao máximo as trocas entre elementos, com o custo de uma estrutura de dados auxiliar.

Exemplo: Ciclo Hamiltoniano

- Considere um conjunto de n cidades e uma matriz M de tamanho n x n tal que M[i][j] = 1, se existir um caminho direto entre as cidades i e j, e M[i][j] = 0, caso contrário
- Problema: existe uma forma de, saindo de uma cidade qualquer, visitar todas as demais cidades, sem passar duas vezes por nenhuma cidade e, no final, retornar para a cidade inicial?
- Se existe uma forma de sair de uma cidade X qualquer, visitar todas as demais cidades (sem repetir nenhuma) e depois retornar para X, então existe um ciclo Hamiltoniano e qualquer cidade do ciclo pode ser usada como ponto de partida
- Como vimos, qualquer cidade pode ser escolhida como cidade inicial. Sendo assim, vamos escolher, arbitrariamente a cidade n como ponto de partida
- Solução: testar todas as permutações das n-1 primeiras cidades, verificando se existe um caminho direto entre a cidade n e a primeira da permutação, assim como um caminho entre todas as cidades consecutivas da permutação e, por fim, um caminho direto entre a última cidade da permutação e a cidade n. Similar a força-bruta.
- Ciclo Hamiltoniano:

$$n \rightsquigarrow [p_1 \rightsquigarrow p_2 \rightsquigarrow p_3 \rightsquigarrow \cdots \rightsquigarrow p_{n-1}] \rightsquigarrow n$$

```cpp
#include<iostream>
using namespace std;

void permutacao(int n, int x[], bool used[], int k){
   int i;
   if (k == n){
      for (i = 0; i < n; i++)
         cout<<x[i]+1<<" ";
      cout<<endl;
   } else {
      for (i = 0; i < n; i++) {
         if (!used[i]) {
            used[i] = true;
            x[k] = i;
            permutacao(n, x, used, k+1);
            used[i] = false;
         }
      }
   }
}
int main () {
   int i, n, x[100];
   bool used[100];
   cout<<"Entre com o valor de n: ";
   cin>>n;
   for (i = 0; i < n; i++)
      used[i] = false;
   permutacao(n,x,used,0);
   return 0;
}
```

6. Técnica de projeto: Divisão e conquista

- O paradigma divisão e conquista consiste em dividir o problema a ser resolvido em partes menores, encontrar soluções para as partes, e então combinar as soluções obtidas em uma solução global
- O paradigma divisão e conquista envolve três passos em cada nível de recursão:
    - **Dividir** o problema em um determinado número de subproblemas
    - **Conquistar** os subproblemas, resolvendo-os recursivamente. Se o tamanho dos subproblemas forem pequenos o bastante, basta resolver os subproblemas de maneira direta
    - **Combinar** as soluções dadas aos problemas a fim de formar a solução para o problema original

- Existem três condições que indicam que a estratégia de divisão e conquista pode ser utilizada com sucesso:
    - Deve ser possível decompor uma instância em sub-instâncias
    - A combinação dos resultados deve ser eficiente (muitas vezes, trivial)
    - As subinstâncias devem ser mais ou menos do mesmo tamanho

a) Exemplo: Exponenciação

Problema: Calcular a^n, para todo real a e inteiro $n \geq 0$.

Primeira solução (incremental):

- Caso base: n = 0; $a^0 = 1$.
- Hipótese de indução: Suponha que, para qualquer inteiro $k < n$ e real a, sei calcular a^k.
- Passo da indução: Queremos provar que conseguimos calcular a^k, para $k=n$. Por hipótese de indução, sei calcular a^{n-1}. Então, calculo a^n multiplicando a^{n-1} por a.

```
Exponenciação(a, n)

    se n = 0 então retorne(1)
    senão an:=Exponenciação(a, n − 1)
    an := an' * a
    retorne(an)
```

Análise:

Vamos agora projetar um algoritmo para o problema usando o método de divisão e conquista.

```
ExponenciaçãoDC(a, n)

        se n = 0 então retorne(1)
        senão
                an := ExponenciaçãoDC(a, n/2)
◁               an := an' * an
                se (n mod 2) = 1 an := an * a
        retorne(an)
```

Análise:

- Colocar 2 condições de contorno: n=0, n=1
- Pior caso: o teste de paridade é sempre tomado

b) Exemplo: Busca Binária

BuscaBinaria(A, e, d, x)

- Entrada: Vetor A, delimitadores e e d do subvetor e x.
- Saída: Indice $0 \leq i \leq n-1$ tal que A[i] = x ou i = -1.
- Chamada inicial: BuscaBinaria(A, 0, n-1, x)

```
        se e = d então
                se A[e] = x então retorne(e) senão retorne(-1)
        senão
                i := (e + d)/2
                se A[i ] = x então retorne(i)
                senão se A[i ] > x
                                i := BuscaBinaria(A, e, i − 1, x)
                        senao i := BuscaBinaria(A, i + 1, d, x)
                retorne(i)
```

Análise:

- Pior caso: elemento não encontrado, fazer a análise considerando que $n=2^k-1$
- Caso médio: cada elemento tem probabilidade 1/n de ser o valor procurado. Usar uma árvore para análise.

Exercícios: Proponha versões não recursivas para os exemplos acima. A eliminação da recursividade altera a complexidade das soluções?

c) Exemplo: MergeSort

- O algoritmo de ordenação Mergesort obedece ao paradigma de dividir e conquistar
  - **DIVIDIR**: divide a sequência de *n* elementos a serem ordenados em duas subsequências de *n*/2 elementos cada uma
  - **CONQUISTAR**: ordena as duas subsequências recursivamente, utilizando o MERGE-SORT
  - **COMBINAR**: faz a intercalação das duas sequências ordenadas, de modo a produzir a resposta ordenada (função MERGE)

```
MERGE-SORT(A,p,r)
if p < r then
   q ← ⌊(p + r)/2⌋        // Dividir
   MERGE-SORT(A,p,q)      //Conquistar
   MERGE-SORT(A,q+1,r)    //Conquistar
   MERGE(A,p,q,r)         //Combinar
```

```
MERGE(A,p,q,r)
n1 ← q-p+1
n2 ← r-q
criar arranjos L[1..n1+1] e R[1..n2+1]
for i ← 1 to n1 do
   L[i] ← A[p+i-1]
for j ← 1 to n2 do
   R[j] ← A[q+j]
L[n1+1] ← ∞
R[n2+1] ← ∞
i ← 1
j ← 1
for k ← p to r do
   if L[i] ≤ R[j] then
      A[k] ← L[i]
      i ← i+1
   else
      A[k] ← R[j]
      j ← j+1
```

Exemplo: 52471326

Análise de Complexidade (supondo $n=2^k$ e que a operação relevante seja a comparação com os elementos do vetor):

- Dividir: a etapa de dividir simplesmente calcula o ponto médio do subarranjo e não realiza comparação
- Conquistar: resolvemos recursivamente dois subproblemas, cada um tem o tamanho n/2 e contribui com 2T(n/2) para o tempo de execução
- Combinar: o procedimento MERGE leva o tempo n, onde n=r-p+1 é o número de elementos que estão sendo intercalados

$$\begin{cases} T(n) = 2T\left(\frac{n}{2}\right) + n & se\ n > 1 \\ T(1) = 0 \end{cases}$$

$$T(n) = n \log n$$

d) Exemplo: QuickSort

- Algoritmo de ordenação baseado na estratégia de Dividir e Conquistar

- Em contraste ao Mergesort, no Quicksort é a operação de divisão a mais custosa: depois de escolhemos o pivot, temos que separar os elementos do vetor maiores que o pivot dos menores que o pivot.

- Conseguimos fazer essa divisão com $\Theta(n)$ operações: basta varrer o vetor com dois apontadores, um varrendo da direita para a esquerda e outro da esquerda para a direita, em busca de elementos situados na parte errada do vetor, e trocar um par de elementos de lugar quando encontrado.

- Após essa etapa, basta ordenarmos os dois trechos do vetor recursivamente para obtermos o vetor ordenado, ou seja, a conquista é imediata.

Quicksort(A, esq, dir)

// Entrada: Vetor A de inteiros e os índices *esq* e *dir* que delimitam início e fim do subvetor a ser ordenado.
// Saída: Subvetor de *A* de *esq* a *dir* ordenado.

```
início
  i=esq
  j=dir
  pivô=A[dir]
  repita
    enquanto (A[i] < pivo) faça i= i + 1
    enquanto (A[j] > pivo) faça j= j – 1
    se (i <= j) então
      troca (A[i], A[j])
      i = i + 1
      j = j – 1
  até_que (i > j)
  se (j > esq) então QuickSort(A, esq, j)
  se (i < dir) então QuickSort(A, i, dir)
fim
```

Análise do pior caso:

- Quantas comparações e quantas trocas o algoritmo Quicksort executa no pior caso?

- Certamente a operação de divisão tem complexidade $\Theta(n)$, mas o tamanho dos dois subproblemas depende do pivot escolhido.
- No pior caso, cada divisão sucessiva do Quicksort separa um único elemento dos demais, recaindo na recorrência:

$T(n) = 0, n = 1$
$T(n) = T(n - 1) + n, n > 1,$

- Portanto, $\Theta(n^2)$ comparações e trocas são executadas no pior caso.
- Então, o algoritmo Quicksort é assintoticamente menos eficiente que o Mergesort no pior caso.
- Veremos que, no caso médio, o Quicksort efetua $\Theta(n \log n)$ comparações e trocas.
- Assim, na prática, o Quicksort é bastante eficiente, com uma vantagem adicional em relação ao Mergesort: é *in place*, isto é, não utiliza um vetor auxiliar.

Análise do caso médio:

- Considere que i é o índice da posição do pivot escolhido no vetor ordenado.
- Supondo que qualquer elemento do vetor tem igual probabilidade de ser escolhido como o pivot
- Então, na média, o tamanho dos subproblemas resolvidos em cada divisão sucessiva será (n>=2):

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

- Supondo T(0)=0, Não é difícil ver que:

$$\sum_{i=1}^{n}T(i-1)=\sum_{i=1}^{n}T(n-i)=\sum_{i=1}^{n-1}T(i)$$

- Assim, no caso médio, o número de operações efetuadas pelo Quicksort é dado pela recorrência:

$$T(n)=\begin{cases}0, n<2\\ \frac{2}{n}\sum_{i=1}^{n-1}T(i)+n-1, n\geq 2\end{cases}$$

- T(n) ≅ 1.4nlogn = Θ(n log n). Portanto, na média, o Quicksort executa Θ(n log n) trocas e comparações.

7. Técnica de projeto: Retrocesso (Backtracking)

A técnica de backtracking consiste em:

- Construir soluções adicionando um componente de cada vez.

- Avaliar estas soluções candidatas parcialmente construídas:

    - Se uma solução parcialmente construída puder continuar a ser desenvolvida, sem violar as condições do problema, pega-se a primeira opção remanescente.

    - Se não houver opção para o próximo componente, o algoritmo retrocede para trocar o último componente inserido pela próxima opção.

- É conveniente construir uma árvore para acompanhar o processo de escolha de opções. As folhas representam nós que não podem levar a uma solução ou soluções completas.

- O espaço de estados representado pela árvore é explorado através de uma busca em profundidade.

- Enquanto algoritmos de forca bruta geram todas as possíveis soluções e só depois verificam se elas são válidas, *backtracking* só gera soluções válidas

a) Exemplo: Problema das *n* rainhas

Colocar *n* rainhas em um tabuleiro de xadrez *n x n* de modo que duas delas não estejam na mesma linha, coluna ou diagonal.

- Para algumas instâncias de *n*, o problema não tem solução.
- O algoritmo posiciona a primeira rainha na casa (1,1).
- Obviamente, a próxima rainha não poderá ficar na linha 1.
- Ao ser posicionada em (2,2), verifica-se que está na mesma diagonal da primeira rainha, logo esta solução parcial é inviável.
- O algoritmo retrocede para escolher a próxima opção (2,3).
- Eventualmente, o retrocesso pode chegar a raiz.
- A árvore terá altura *n.*

b) Exemplo: Soma do subconjunto

Encontrar um subconjunto de um dado conjunto *S* de *n* inteiros positivos cuja soma seja igual a um inteiro *d*.

Análise:

- No pior caso, algoritmos de backtracking têm comportamento exponencial.

- O algoritmo, no entanto, acaba podando ramos inviáveis, reduzindo bastante o tempo total.

- Se apenas uma solução for suficiente, o algoritmo pode parar bem antes do final.

- O sucesso desta técnica varia bastante de problema a problema e entre instâncias do mesmo problema.

- É uma técnica geralmente aplicada a problemas combinatoriais, para os quais não existam algoritmos eficientes.

- A implementação não recursiva requer uma pilha que é *O(n)* onde *n* é a altura da árvore.