Projeto e Analise de Algoritmos

- Notação assintótica *O*-, Ω- e Θ-
- Relações de recorrência
- Método da substituição
- Recorrência
- Árvore de recursão

Notação Assintótica

O- (Limite Superior):

Escrevemos $f(n) = O(g(n))$ se existem constantes $c>0$, $n_0>0$ tais que $0 \leq f(n) \leq cg(n)$ para todo $n \geq n_0$.

Notação Assintótica

O- (Limite Superior):

Escrevemos $f(n) = O(g(n))$ se existem constantes
$c>0,\ n_0>0$ tais que
$0 \leq f(n) \leq cg(n)$ para todo $n \geq n_0$.

Exemplo: $2n^2 = O(n^3)$ $(c = 1,\ n_0 = 2)$

Notação Assintótica

O- (Limite Superior):

Escrevemos $f(n) = O(g(n))$ se existem constantes $c>0,\ n_0>0$ tais que $0 \leq f(n) \leq cg(n)$ para todo $n \geq n_0$.

Exemplo: $2n^2 = O(n^3)$ $\quad (c = 1,\ n_0 = 2)$

funções,
Não são
valores

Notação Assintótica

O- (Limite Superior):

Escrevemos $f(n) = O(g(n))$ se existem constantes $c>0$, $n_0>0$ tais que $0 \leq f(n) \leq cg(n)$ para todo $n \geq n_0$.

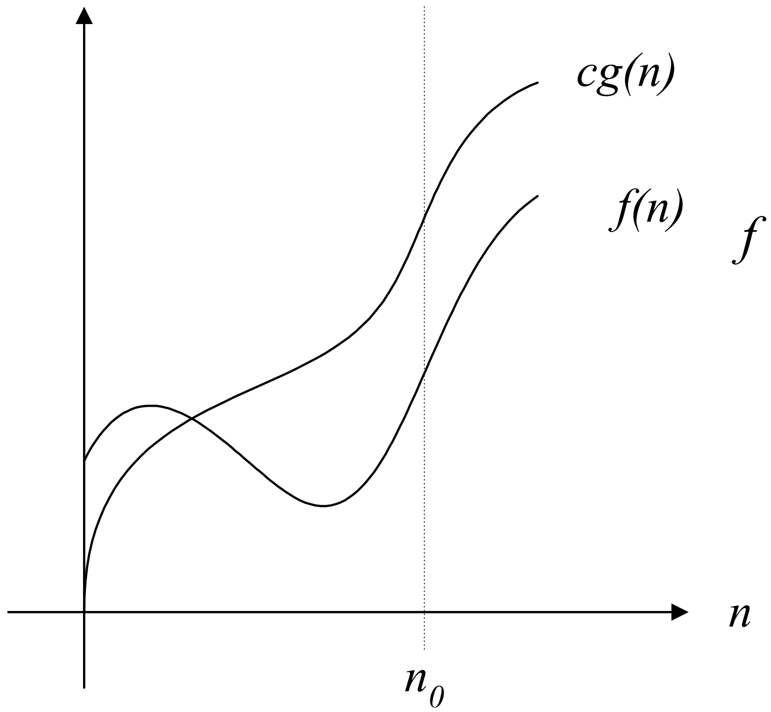
Exemplo: $2n^2 = O(n^3)$ $(c = 1, n_0 = 2)$

funções, Não são valores

Igualdade unidirecional

Notação Assintótica – Limite Superior

$$O(g(n)) = \{f(n) \mid \exists c, n_0 \text{ s.t. } 0 \le f(n) \le cg(n)\ \forall n \ge n_0\}$$



$$f(n) = O(g(n))$$

Notação Assintótica

- Transitividade

  $f(n) = \Theta(g(n))$ e $g(n) = \Theta(h(n))$

  $\Rightarrow f(n) = \Theta(h(n))$

  (tambem é verdade para o, O, ω e Ω).
- Simetria

  $f(n) = \Theta(g(n))$ se e somente se $g(n) = \Theta(f(n))$
- Simetria transposta

  $f(n) = O(g(n))$ se e somente se $g(n) = \Omega(f(n))$

  $f(n) = o(g(n))$ se e somente se $g(n) = \omega(f(n))$

Funções Logarítmicas

- É importante entender como funcionam os logaritmos
- O logaritmos é o inverso da função exponencial.
- Dizer que $b^x = y$ é equivalente a dizer que
  $x = \log_b y$.
- Os logaritmos refletem quantas vezes nos podemos dobrar alguma coisa até que chegue a um valor n ou quantas vezes devemos dividir alguma coisa por dois até que chegue em 1
- $\log_2 1 = ?$
- $\log_2 2 = ?$

Pesquisa Binária

- Em uma pesquisa binária nós eliminamos metade da entrada a cada uma das comparações executadas
- Quantas vezes nos podemos dividir n (potência de 2) ao meio até encontrar o valor 1?
- Resposta: lg n
- E se n não for potência de 2? Teto(log n)

Logaritmos e árvores

- Qual deve ser a altura de uma árvore de forma que ela tenha n folhas?
- O número potencial de folhas dobra em cada um dos níveis
- Quantas vezes devemos dobrar o valor de 1 até atingir um valor de pelo menos n?
- Resposta: teto (lg n)

Logaritmos e bits

- Quantos números podem ser representados por k bits?
- A cada bit adicionado se dobra o número de possibilidades que podem ser representadas
- Se pode representar os números de 0 até $2^k - 1$ com k bits. Total de 2^k números.
- Quantos bits são necessários para representar os números de 0 até n?
- teto (lg (n+1))

Logaritmos

- $\lg n = \log_2 n$
- $\ln n = \log_e n$, $e \approx 2.718$
- $\lg^k n = (\lg n)^k$
- $\lg \lg n = \lg (\lg n) = \lg^{(2)} n$
- $\lg^{(k)} n = \lg \lg \lg \ldots \lg n$
- $\lg^2 4 = ?$
- $\lg^{(2)} 4 = ?$
- $\lg^k n$ vs $\lg^{(k)} n$?

Iteração da função logaritmo

- lg * n é o menor inteiro positivo i tal que $lg^{(i)}$ <= 1. Também conhecido como $\alpha(n)$.
- O número de vezes que é necessário tirar o logaritmo de um número n até que ele se torna menor ou igual a 1
- lg * 256 = ?
- lg 256 = 8
- lg 8 = 3
- lg 3 < 2
- lg lg 3 < 1

lg*2 = 1

lg*4 = 2

lg*16 = 3

lg*65536 = 4

$lg^*2^{65536} = lg^*(10^{19728}) = 5$

Na prática lg*(n) pode ser considerado uma constante

Regras para Logaritmos

- Para todo $a > 0$, $b > 0$, $c > 0$, valem as regras
- $\log_b a = \log_c a / \log_c b = \lg a / \lg b$
- $\log_b a^n = n \log_b a$
- $b^{\log_b a} = a$
- $\log (ab) = \log a + \log b$
  - $\lg (2n) = ?$
- $\log (a/b) = \log (a) - \log(b)$
  - $\lg (n/2) = ?$
  - $\lg (1/n) = ?$
- $\log_b a = 1 / \log_a b$

Ordem de Complexidade

$$n^n >> n! >> 3^n >> 2^n >> n^3 >> n^2 >> n^{1+\varepsilon} >> n \log n \sim \log n!$$

$$>> n >> n / \log n >> \sqrt{n} >> n^{\varepsilon} >> \log^3 n >> \log^2 n >> \log n$$

$$>> \log n / \log \log n >> \log \log n >> \log^{(3)} n >> \alpha(n) >> 1$$

Análise de Complexidade

```
Elemento_Repetido (A, n)
    // Verifica se existe algum elemento repetido em um vetor
    for i = 1 to n-1          {
        for j = i+1 to n {
            if (A[i] == A[j])
                return true;
        }
    }
    return false;
```

- Melhor Caso?

- Pior Caso?

- Caso Médio?

- Melhor Caso
  - A[1] = A[2]
  - $T(n) = \Theta(1)$
- Pior Caso
  - Sem elementos repetidos
  - $T(n) = (n-1) + (n-2) + \ldots + 1 = n(n-1) / 2 = \Theta(n^2)$
- Caso médio
  - Definir o que é médio
  - Necessita mais suposições sobre a distribuição dos dados
    - Qual é a freqüência de repetição nos dados?
  - Uma análise média envolve probabilidade

Encontrar a ordem de crescimento a partir de uma soma

- $T(n) = \sum_{i=1..n} i = \Theta\ (n^2)$
- $T(n) = \sum_{i=1..n} \log\ (i) = ?$
- $T(n) = \sum_{i=1..n} n\ /\ 2^i = ?$
- $T(n) = \sum_{i=1..n} 2^i = ?$
- …

- Como pode ser calculado?

Séries Artiméticas

- Uma série aritmética é a soma de uma seqüência de números tal que a diferença entre dois números sucessivos é uma constante

  Ex: 1, 2, 3, 4, 5

  10, 12, 14, 16, 18, 20
- Definição:

$$a_j = a_{j-1} + d \longleftarrow$$ Definição recursiva

ou: $$a_j = a_1 + (j-1)d \longleftarrow$$ Forma explícita

Soma de uma série aritmética

Se $a_1, a_2, \ldots, a_n$ é uma série aritmética então

$$\sum_{i=1}^{n} a_i = \frac{n(a_1 + a_n)}{2}$$

Ex. 1 + 2 + 3 + …+ 97 + 98 + 99 = ?

Série Geométrica

- Uma série geométrica é a soma de uma seqüência de números tal que a razão entre dois números sucessivos na série e uma constante

  Ex: 1, 2, 4, 8, 16, 32

  10, 20, 40, 80, 160

  1, ½, ¼, 1/8, 1/16
- Definição

$$a_j = r a_{j-1} \longleftarrow$$ Definição recursiva

Ou $$a_j = r^{j-1} a_1 \longleftarrow$$ Fórmula Explícita

Soma de uma série geométrica

$$\sum_{i=0}^{n} r^i = \begin{cases} (1-r^{n+1})/(1-r) & \text{if r < 1} \\ (r^{n+1}-1)/(r-1) & \text{if r > 1} \\ n+1 & \text{if r = 1} \end{cases}$$

$$\sum_{i=0}^{n} 2^i = ?$$

$$\lim_{n\to\infty} \sum_{i=0}^{n} \frac{1}{2^i} = ?$$

$$\lim_{n\to\infty} \sum_{i=1}^{n} \frac{1}{2^i} = ?$$

Soma de uma série geométrica

$$\sum_{i=0}^{n} r^i = \begin{cases} (1-r^{n+1})/(1-r) & \text{if r < 1} \\ (r^{n+1}-1)/(r-1) & \text{if r > 1} \\ n+1 & \text{if r = 1} \end{cases}$$

$$\sum_{i=0}^{n} 2^i = \frac{2^{n+1}-1}{2-1} = 2^{n+1}-1 \approx 2^{n+1}$$

$$\lim_{n\to\infty} \sum_{i=0}^{n} \frac{1}{2^i} = \lim_{n\to\infty} \sum_{i=0}^{n} (1/2)^i = \frac{1}{1-1/2} = 2$$

$$\lim_{n\to\infty} \sum_{i=1}^{n} \frac{1}{2^i} = \lim_{n\to\infty} \sum_{i=0}^{n} (1/2)^0 - (1/2)^0 = 2-1 = 1$$

Fórmulas Importantes

$$\sum_{i=1}^{n} 1 = n \in \Theta(n)$$

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

$$\sum_{i=0}^{n} r^i = \frac{r^{n+1}-1}{r-1} \in \begin{cases} \Theta(1) & (r<1) \\ \Theta(r^n) & (r>1) \end{cases}$$

$$\sum_{i=1}^{n} i^2 \approx \frac{n^3}{3} \in \Theta(n^3)$$

$$\sum_{i=1}^{n} i^k \approx \frac{n^{k+1}}{k+1} \in \Theta(n^{k+1})$$

$$\sum_{i=1}^{n} i2^i = (n-1)2^{n+1} + 2 \in \Theta(n2^n)$$

$$\sum_{i=1}^{n} \frac{1}{i} \in \Theta(\lg n)$$

$$\sum_{i=1}^{n} \lg i \in \Theta(n \lg n)$$

Regras para manipulação de somas

$$\sum_i (a_i + b_i) = \sum_i a_i + \sum_i b_i$$

$$\sum_i c a_i = c \sum_i a_i$$

$$\sum_{i=m}^{n} a_i = \sum_{i=m}^{x} a_i + \sum_{i=x+1}^{n} a_i$$

Exemplo:

$$\sum_{i=1}^{n} (4i + 2^i) = ?$$

$$\sum_{i=1}^{n} \frac{n}{2^i} = ?$$

Regras para manipulação de somas

$$\sum_i (a_i + b_i) = \sum_i a_i + \sum_i b_i$$

$$\sum_i c a_i = c \sum_i a_i$$

$$\sum_{i=m}^{n} a_i = \sum_{i=m}^{x} a_i + \sum_{i=x+1}^{n} a_i$$

Exemplo:

$$\sum_{i=1}^{n} (4i + 2^i) = 4\sum_{i=1}^{n} i + \sum_{i=1}^{n} 2^i = 2n(n+1) + 2^{n+1} - 2$$

$$\sum_{i=1}^{n} \frac{n}{2^i} = n \sum_{i=1}^{n} \frac{1}{2^i} \approx n$$

- $\sum_{i=1..n} n / 2^i = n * \sum_{i=1..n} (½)^i = ?$

- Usando a fórmula para séries geométricas:
  $\sum_{i=0..n} (½)^i = 1 + ½ + ¼ + \ldots (½)^n = 2$

- Aplicação para algoritmos de alocação de memórias

- $\sum_{i=1..n} \log (i) = \log 1 + \log 2 + \ldots + \log n$

  $= \log 1 \times 2 \times 3 \times \ldots \times n$

  $= \log n!$

  $= n \log n$

- Aplicação para algoritmos de ordenação

Definição recursiva de uma soma de série

- $T(n) = \sum_{i=0..n} i$ é equivalente a:

$$\begin{cases} T(n) = T(n-1) + n \\ T(0) = 0 \end{cases}$$

Relação de Recorrência

Condição Limite

- $T(n) = \sum_{i=0..n} a^i$ é equivalente a:

$$\begin{cases} T(n) = T(n-1) + a^n \\ T(0) = 1 \end{cases}$$

A definição recursiva normalmente é intuitiva e fácil de ser obtida. É bastante útil para analisar algoritmos recursivos

Definição recursiva de uma soma de série

- Como resolver estas recorrências na forma:
- $T(n) = aT(n-b) + f(n)$ ou
- $T(n) = aT(n/b) + f(n)$

Indução

- Suponha
  - S(k) é verdadeiro para uma constante k
    - Normalmente k = 0
  - S(n) → S(n+1) para todo n >= k
- Então S(n) é verdadeiro para todo n >= k

Prova por Indução

- Declaração: S(n) é verdadeiro para todo n >= k
- Base da indução:
  - Mostrar que é verdadeiro quando n = k
- Hipótese de indução:
  - Supõe que fórmula é verdadeira para um n arbitrário
- Passo da indução:
  - Mostrar que fórmula é então verdadeira para n+1
- Objetivo: Mostrar que se fórmula é válida para n então é válida para n+1. Se isto é verdade e também que a fórmula vale para um n>=c inicial, então vale para todo n >=c.

Exemplo de Indução: Fórmula de Gauss

- Prove $1 + 2 + 3 + \ldots + n = n(n+1) / 2$
  - Base:
    - ?
  - Hipótese de indução:
    - ?
  - Passo (mostrar que é verdade para n+1):

    ?

Exemplo de Indução: Fórmula de Gauss

- Prove $1 + 2 + 3 + \ldots + n = n(n+1) / 2$
  - Base:
    - Se $n = 1$, então $1 = 1(1+1) / 2$
  - Hipótese de indução:
    - Suponha $1 + 2 + 3 + \ldots + n = n(n+1) / 2$
  - Passo (mostrar que é verdade para n+1):

    $1 + 2 + \ldots + n + n+1 = (1 + 2 + \ldots + n) + (n+1)$

    $= n(n+1)/2 + n+1 = [n(n+1) + 2(n+1)]/2$

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

Exemplo de Indução: Fórmula Série Geométrica

- Prove $a^0 + a^1 + \ldots + a^n = (a^{n+1} - 1)/(a - 1)$ for all $a \neq 1$
  - Base: mostrar que $a^0 = (a^{0+1} - 1)/(a - 1)$

    ?
  - Hipótese de Indução:
    - ?
  - Passo (mostrar que é verdade para n+1):

    ?

Exemplo de Indução: Fórmula Série Geométrica

- Prove $a^0 + a^1 + \ldots + a^n = (a^{n+1} - 1)/(a - 1)$ for all $a \neq 1$
  - Base: mostrar que $a^0 = (a^{0+1} - 1)/(a - 1)$

    $a^0 = 1 = (a^1 - 1)/(a - 1)$
  - Hipótese de Indução:
    - Suponha $a^0 + a^1 + \ldots + a^n = (a^{n+1} - 1)/(a - 1)$
  - Passo (mostrar que é verdade para n+1):

    $a^0 + a^1 + \ldots + a^{n+1} = a^0 + a^1 + \ldots + a^n + a^{n+1}$

    $= (a^{n+1} - 1)/(a - 1) + a^{n+1} = (a^{n+1+1} - 1)/(a - 1)$

Como mostrar que um algoritmo recursivo está correto?

- Por indução:
  - Base: mostrar que funciona para exemplos pequenos
  - Hipótese de indução: assume que a solução é correta para todos os subproblemas
  - Passo de indução: mostrar que se a hipótese de indução é correta então o algoritmo é correto para o problema original

Verificar correção do merge sort

MERGE-SORT $A[1\,..\,n]$

1. Se $n = 1$, então fim.
2. Ordena recursivamente $A[\,1\,..\,\lceil n/2 \rceil\,]$ e $A[\,\lceil n/2 \rceil + 1\,..\,n\,]$.
3. "***Merge***" de 2 listas ordenadas

Prova:

1. Caso base: Se n = 1, o algoritmo retorna a resposta correta pois A[1..1] já está ordenado.
2. Hipótese de indução: assume que o algoritmo ordena corretamente $A[1..\lceil n/2 \rceil]$ e $A[\lceil n/2 \rceil+1..n]$.
3. Passo: se $A[1..\lceil n/2 \rceil]$ e $A[\lceil n/2 \rceil+1..n]$ são ordenados corretamente, então todo o vetor $A[1..\lceil n/2 \rceil]$ e $A[\lceil n/2 \rceil+1..n]$ está ordenado depois do "merge"

Como analisar a eficiência no tempo de um algoritmo?

- Colocar a execução em função de n como uma função do tempo dos subproblemas menores

Análise do merge sort

$T(n)$	**MERGE-SORT** $A[1\,..\,n]$
$\Theta(1)$	1 Se $n = 1$, então fim.
$2T(n/2)$	2 Ordena recursivamente $A[\,1\,..\lceil n/2\rceil\,]$ e $A[\,\lceil n/2\rceil+1\,..\,n\,]$.
$f(n)$	3 "***Merge***" de 2 listas ordenadas

Deveria ser $T(\lceil n/2\rceil) + T(\lfloor n/2\rfloor)$, mas não faz diferença assintoticamente

Análise do merge sort

1 Divide: Trivial.
2 Conquistar: Ordena recursivamente 2 subarrays.
3 Combinar: Faz o merge de 2 subarrays ordenados

$$T(n) = 2\,T(n/2) + f(n) + \Theta(1)$$

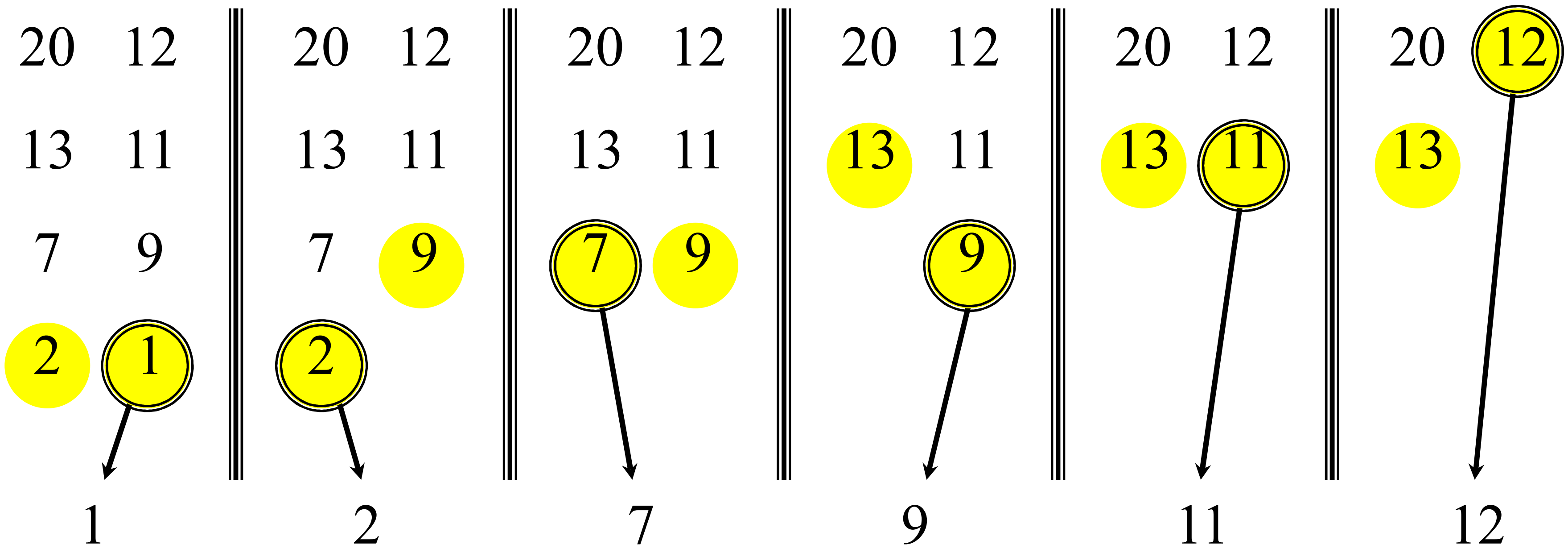
 # subproblemas

Dividir e combinar

Tamanho do subproblema

1. Qual é o tempo do caso base? Constante
2. O que é $f(n)$?
3. Qual é a ordem de crescimento de $T(n)$?

Merge de duas listas ordenadas



$\Theta(n)$ tempo total para o merge de *n* elementos (tempo linear)

Recorrência do merge sort

$$T(n) = \begin{cases} \Theta(1) \text{ se } n = 1; \\ 2T(n/2) + \Theta(n) \text{ se } n > 1. \end{cases}$$

- Podemos com freqüência omitir o caso base quando $T(n) = \Theta(1)$ para um n pequeno o suficiente mas só quando não afeta o resultado da solução assintótica

- Qual a solução de $T(n)$? $O(n)$, $O(n^2)$ ou $O(n^3)$ …?

Pesquisa Binária

Para Encontrar um elemento em um vetor ordenado

1. Verificar elemento do meio
2. Se ==, foi encontrado
3. Senão: se menor procura na metade da esquerda,
4. Senão procura na metade da direita

Exemplo: encontrar 9

3	5	7	8	9	12	15

Pesquisa Binária

Para Encontrar um elemento em um vetor ordenado

1. Verificar elemento do meio
2. Se ==, foi encontrado
3. Senão: se menor procura na metade da esquerda,
4. Senão procura na metade da direita

Exemplo: encontrar 9

3	5	7	8	9	12	15

Pesquisa Binária

Para Encontrar um elemento em um vetor ordenado

1. Verificar elemento do meio
2. Se ==, foi encontrado
3. Senão: se menor procura na metade da esquerda,
4. Senão procura na metade da direita

Exemplo: encontrar 9

3	5	7	8	9	12	15

Pesquisa Binária

Para Encontrar um elemento em um vetor ordenado

1. Verificar elemento do meio
2. Se ==, foi encontrado
3. Senão: se menor procura na metade da esquerda,
4. Senão procura na metade da direita

Exemplo: encontrar 9

3	5	7	8	9	12	15

Pesquisa Binária

Para Encontrar um elemento em um vetor ordenado

1. Verificar elemento do meio
2. Se ==, foi encontrado
3. Senão: se menor procura na metade da esquerda,
4. Senão procura na metade da direita

Exemplo: encontrar 9

3	5	7	8	9	12	15

Pesquisa Binária

Para Encontrar um elemento em um vetor ordenado

1. Verificar elemento do meio
2. Se ==, foi encontrado
3. Senão: se menor procura na metade da esquerda,
4. Senão procura na metade da direita

Exemplo: encontrar 9

3 5 7 8 9 12 15

Pesquisa Binária

```
// inicialmente chamada com low = 0, high = N – 1
   BinarySearch_Right(A[0..N-1], value, low, high)
{
   if (high < low) return -1
   mid = low +((high – low) / 2)
   if (A[mid] > value)
       return BinarySearch_Right(A, value, low, mid-1)
       else return BinarySearch_Right(A, value, mid+1, high)
}
```

Qual é a relação de recorrência para o tempo?

Recorrência para Pesquisa Binária

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(1)$$

$$T(1) = \Theta(1)$$

Cálculo do Fatorial

Fatorial (n)

```
if (n == 1) return 1;
return n * Fatorial (n-1);
```

Cálculo da Recorrência do Fatorial

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

$$T(1) = \Theta(1)$$

Formas de Recorrência

$$T(n) = T(n-1) + 1$$

$$T(n) = T(n-1) + n$$

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

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

Como resolver a recorrência para conseguir uma forma fechada. Ex. $T(n) = \Theta(n^2)$ or $T(n) = \Theta(nlgn)$, ou pelo menos um limite como $T(n) = O(n^2)$?

Resolvendo a Recorrência

- O tempo de execução de vários algoritmos podem ser expressados com uma das seguintes formas recursivas

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

ou

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

As duas podem ser difíceis de se resolver.
Lidaremos aqui com as relativamente mais fáceis

Formando Relações de Recorrência

```
void conta (int n) {
   if (n > 0) {
       printf("%d ",n);
       conta(n-1);
   }
}
```

- O caso base ocorre quando n==0. Assim o T(0) é igual a uma constante *a*
- Quando n>0 a função imprime o valor e se chama recursivamente com parâmetro n-1
- A relação de recorrência é dada por:

$T(0) = a$ para uma constante a

$T(n) = b + T(n-1)$ para uma constante b

Formando Relações de Recorrência

- Em geral, T(*n*) é a soma de várias escolhas de T(*m*), que é o custo dos problemas subproblemas recursivos adicionados do custo do trabalho feito fora da chamada recursiva
- $$\mathbf{T(\boldsymbol{n}) = \boldsymbol{a}T(f(\boldsymbol{n})) + \boldsymbol{b}T(g(\boldsymbol{n})) + \ldots + c(\boldsymbol{n})}$$
- Onde a e b são o número de subproblemas e f(n) e g(n) são os tamanhos dos subproblemas.
- c(n) é o custo do trabalho feito fora da chamada recursiva (c(n) pode ser uma constante)

Formando Relações de Recorrência

```
int g(int n) {
   if (n == 1)
      return 2;
   else
      return 3 * g(n / 2) + g( n / 2) + 5;
}
```

- No caso base n == 1 faz uma comparação e retorna. T(1) é uma constante **c**.
- Quando **n > 1** faz duas chamadas recursivas com parâmetros n **/ 2,** e algumas operações (constante **b**)
- A relação de recorrência :

$T(1) = c$	para uma constante c
$T(n) = b + 2T(n / 2)$	para uma constante b

Formando Relações de Recorrência

```
long fibonacci (int n) {
  if( n == 1 || n == 2)
      return 1;
  else
      return fibonacci(n - 1) + fibonacci(n - 2);
}
```

- A relação de recorrência é:

$$T(n) = c \qquad \text{se } n = 1 \text{ ou } n = 2$$

$$T(n) = T(n-1) + T(n-2) + b \qquad \text{se } n > 2$$