UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE CIÊNCIAS MATEMÁTICAS E DE COMPUTAÇÃO
Departamento de Ciências de Computação

SCC-501 - Capítulo 3
Análise de Algoritmos - Parte 2

João Luís Garcia Rosa[1]

[1]Departamento de Ciências de Computação
Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo - São Carlos
http://www.icmc.usp.br/~joaoluis

2011

Sumário

Sumário

Exercício

- Analise a sub-rotina recursiva abaixo:

```
1 sub-rotina fatorial(n: numérico)
2 início
3   declare aux numérico;
4   se n = 1
5     então aux ← 1
6     senão aux ← n * fatorial(n − 1);
7   fatorial ← aux;
8 fim
```

Regras para o cálculo

- Sub-rotinas recursivas:
  - Se a recursão é um "disfarce" da repetição (e, portanto, a recursão está mal empregada, em geral), basta analisá-la como tal,
  - O exemplo anterior é obviamente $\mathcal{O}(n)$.
- Eliminando a recursão:

```
1 sub-rotina fatorial(n: numérico)
2 início
3   declare aux numérico;
4   aux ← 1
5   enquanto n > 1 faça
6     aux ← aux * n;
7     n ← n − 1;
8   fatorial ← aux;
9 fim
```

Regras para o cálculo

- Sub-rotinas recursivas:
  - Em muitos casos (incluindo casos em que a recursividade é bem empregada), é difícil transformá-la em repetição:
    - Nesses casos, para fazer a análise do algoritmo, pode ser necessário se recorrer à análise de recorrência.
    - **Recorrência**: equação ou desigualdade que descreve uma função em termos de seu valor em entradas menores.
    - Caso típico: algoritmos de dividir-e-conquistar, ou seja, algoritmos que desmembram o problema em vários subproblemas que são semelhantes ao problema original, mas menores em tamanho, resolvem os subproblemas recursivamente e depois combinam essas soluções com o objetivo de criar uma solução para o problema original.

Regras para o cálculo

- Exemplo de uso de recorrência:
  - Números de Fibonacci:
    - 0,1,1,2,3,5,8,13...
    - $fib(0) = 0$, $fib(1) = 1$, $fib(i) = fib(i-1) + fib(i-2)$.
  - A rotina:

```
1 sub-rotina fib(n: numérico)
2 início
3 declare aux numérico;
4 se n = 1
5   então aux ← 1
6   senão aux ← fib(n − 1) + fib(n − 2);
7 fib ← aux;
8 fim
```

Regras para o cálculo

- Seja $T(n)$ o tempo de execução da função:
  - **Caso 1**: Se $n = 0$ ou $n = 1$, o tempo de execução é constante, que é o tempo de testar o valor de n no comando *se*, mais atribuir o valor 1 à variável *aux*, mais atribuir o valor de *aux* ao nome da função; ou seja, $T(0) = T(1) = 3$.
  - **Caso 2**: Se $n > 2$, o tempo consiste em testar o valor de n no comando *se*, mais o trabalho a ser executado no *senão* (que é uma soma, uma atribuição e 2 chamadas recursivas), mais a atribuição de *aux* ao nome da função; ou seja, a recorrência $T(n) = T(n-1) + T(n-2) + 4$, para $n > 2$.

Regras para o cálculo

- Muitas vezes, a recorrência pode ser resolvida com base na prática e experiência do analista,
- Alguns métodos para resolver recorrências:
  - Método da substituição,
  - Método mestre,
  - Método da árvore de recursão.

Sumário

Resolução de recorrências

- Método da substituição:
  - Supõe-se (aleatoriamente ou com base na experiência) um limite superior para a função e verifica-se se ela não extrapola este limite:
    - Uso de indução matemática.
  - O nome do método vem da “substituição” da resposta adequada pelo palpite,
  - Pode-se “apertar” o palpite para achar funções mais exatas.

Resolução de recorrências

- Método mestre:
  - Fornece limites para recorrências da forma $T(n) = aT(\frac{n}{b}) + f(n)$, em que $a \geq 1$, $b > 1$ e $f(n)$ é uma função dada,
  - Envolve a memorização de alguns casos básicos que podem ser aplicados para muitas recorrências simples.

Resolução de recorrências

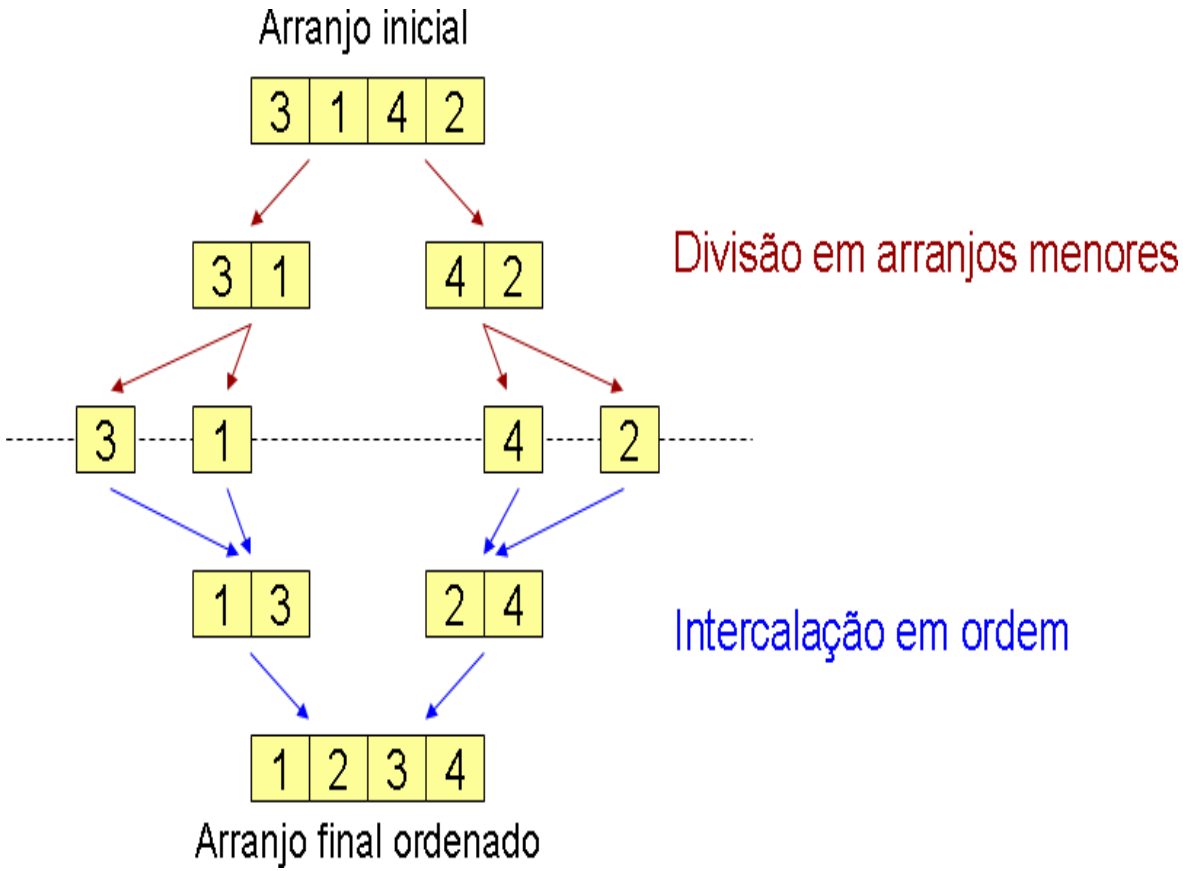
- Método da árvore de recursão:
  - Traça-se uma árvore que, nível a nível, representa as recursões sendo chamadas,
  - Em seguida, em cada nível/nó da árvore, são acumulados os tempos necessários para o processamento:
    - No final, tem-se a estimativa de tempo do problema.
  - Este método pode ser utilizado para se fazer uma suposição mais informada no método da substituição.

Resolução de recorrências

- Método da árvore de recursão:
  - **Exemplo**: algoritmo de ordenação de arranjos por intercalação:
    - Passo 1: divide-se um arranjo não ordenado em dois subarranjos,
    - Passo 2: se os subarranjos não são unitários, cada subarranjo é submetido ao passo 1 anterior; caso contrário, eles são ordenados por intercalação dos elementos e isso é propagado para os subarranjos anteriores.

Ordenação por intercalação

- Exemplo com arranjo de 4 elementos:



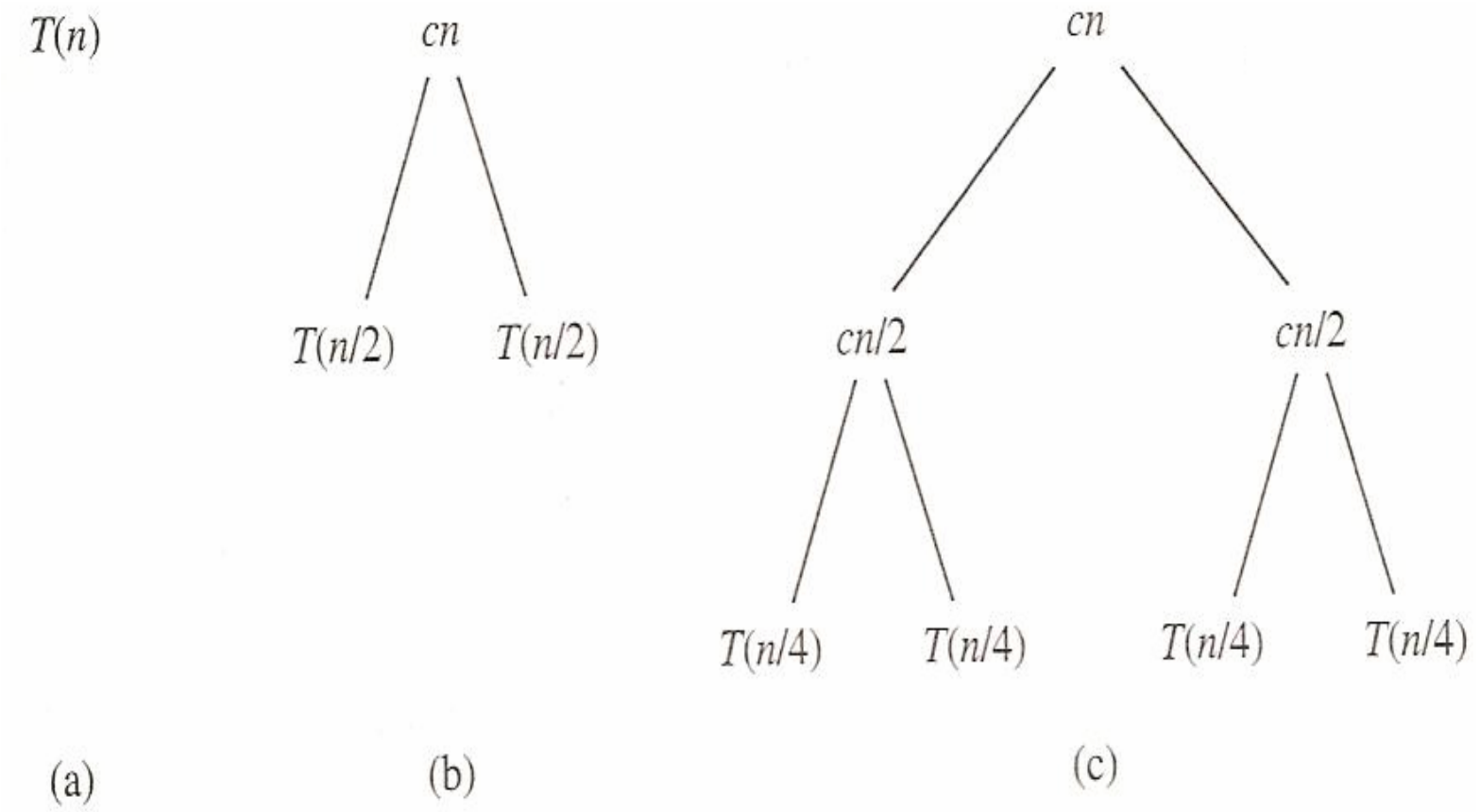
- Implemente a(s) sub-rotina(s) e calcule sua complexidade.

Resolução de recorrências

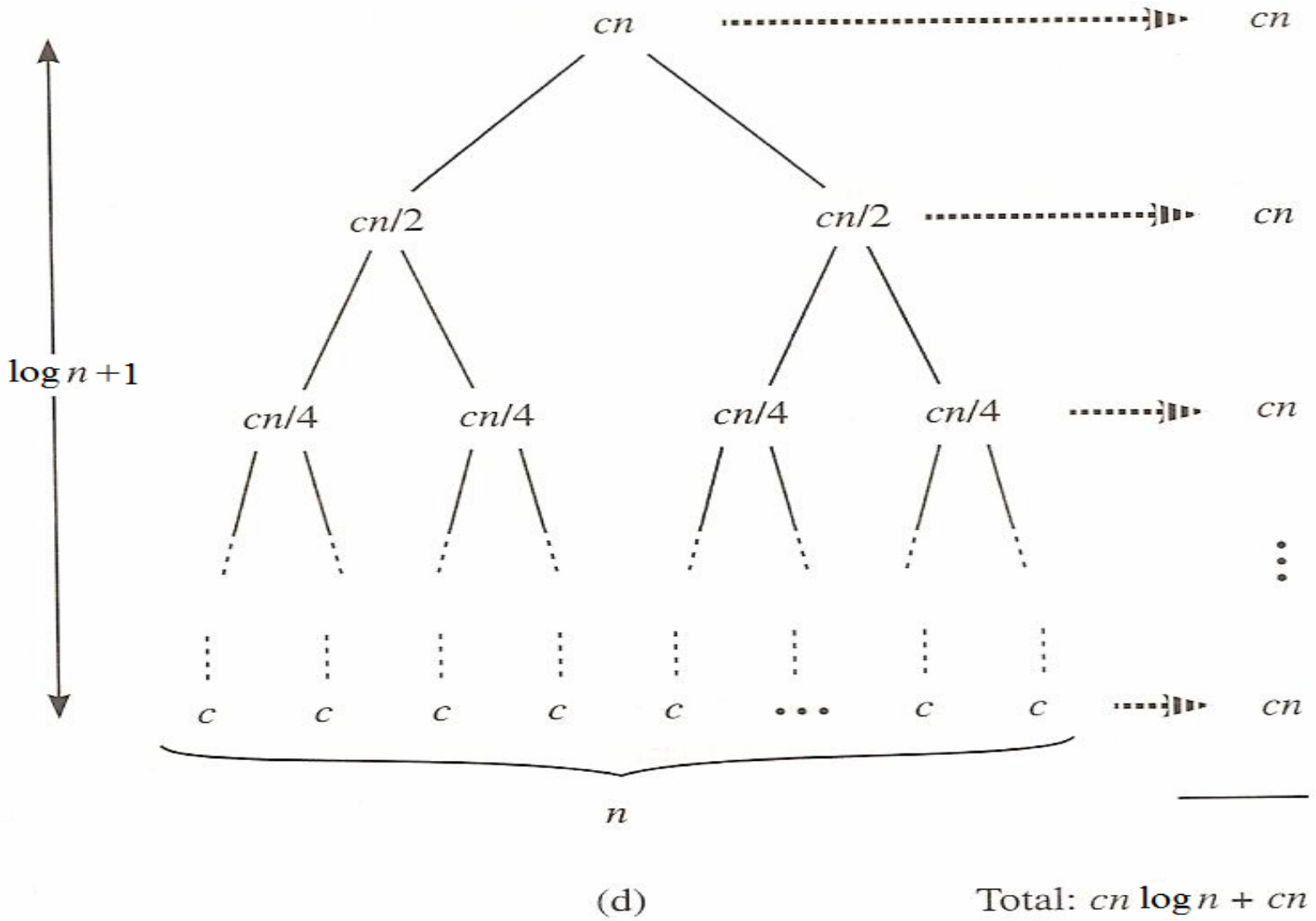
- Método da árvore de recursão:
  - Considere o tempo do algoritmo (que envolve recorrência):

$$T(n) = c, \text{ se } n = 1$$
$$T(n) = 2T(\tfrac{n}{2}) + cn, \text{ se } n > 1$$

Resolução de recorrências



Resolução de recorrências



Resolução de recorrências

- Tem-se que:
  - Na parte (a), há $T(n)$ ainda não expandido,
  - Na parte (b), $T(n)$ foi dividido em árvores equivalentes representando a recorrência com custos divididos ($T(\frac{n}{2})$ cada uma), sendo cn o custo no nível superior da recursão (fora da recursão e, portanto, associado ao nó raiz),
  - ...
  - No fim, nota-se que o tamanho da árvore corresponde a $(log\ n) + 1$, o qual multiplica os valores obtidos em cada nível da árvore, os quais, nesse caso, são iguais:
    - Como resultado, tem-se $cn\ log\ n + cn$, ou seja, $\mathcal{O}(n\ log\ n)$.

Resolução de recorrências

- Alguns dizem que a expressão correta é "$f(n)$ é $\mathcal{O}(g(n))$":
  - Seria considerado redundante e inadequado dizer "$f(n) \leq \mathcal{O}(g(n))$" ou (ainda pior) "$f(n) = \mathcal{O}(g(n))$",
  - Não é incorreto (embora não seja usual) dizer "$f(n) \in \mathcal{O}(g(n))$", já que o operador *Big-oh* representa todo um conjunto de funções.

Sumário

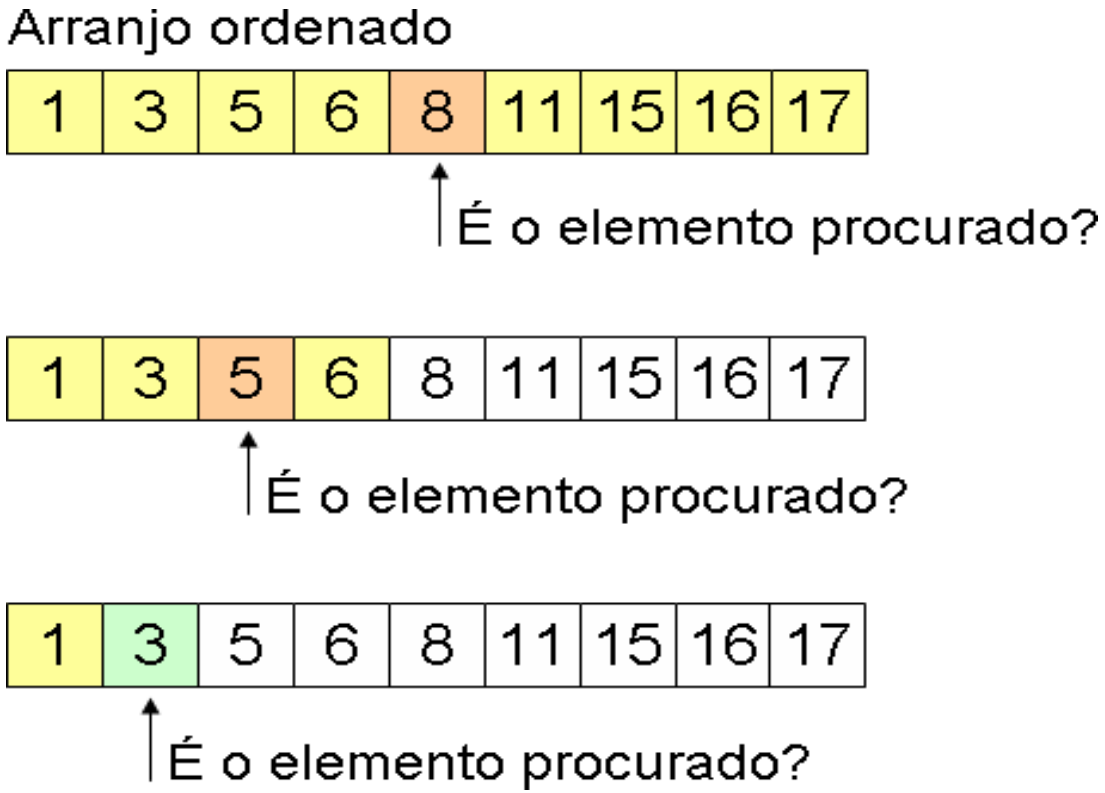
Cuidado!

- A análise assintótica é uma ferramenta fundamental ao projeto, análise ou escolha de um algoritmo específico para uma dada aplicação,
- No entanto, deve-se ter sempre em mente que essa análise "esconde" fatores assintoticamente irrelevantes, mas que em alguns casos podem ser relevantes na prática, particularmente se o problema de interesse se limitar a entradas (relativamente) pequenas:
  - Por exemplo, um algoritmo com tempo de execução da ordem de $10^{100}n$ é $\mathcal{O}(n)$, assintoticamente melhor do que outro com tempo 10 $n \log n$, o que nos faria, em princípio, preferir o primeiro,
  - No entanto, 10^{100} é o número estimado por alguns astrônomos como um limite superior para a quantidade de átomos existente no universo observável!

Sumário

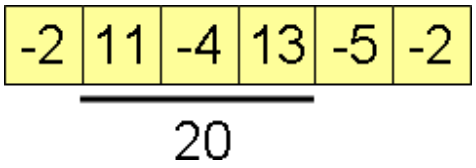
Análise de algoritmos recursivos

- Muitas vezes, temos que resolver recorrências:
  - **Exemplo**: pesquisa binária pelo número 3:



Análise de algoritmos recursivos

- Implemente o algoritmo da busca binária em um arranjo ordenado,
- Teste e analise o algoritmo,
- Problema da maior soma de subseqüência em um arranjo:



- Faça um algoritmo para resolver o problema e analise-o,
- Algoritmo de Euclides para calcular o máximo divisor comum para 2 números,
- Algoritmo para calcular x^n.

Sumário

Estratégia

- Dada uma função para computar n entradas, a estratégia *dividir-e-conquistar* separa as entradas em k subconjuntos distintos, $1 < k \leq n$, levando a k subproblemas,
- Estes subproblemas devem ser resolvidos, e então um método deve ser encontrado para combinar subsoluções em uma solução do todo,
- Se os subproblemas ainda forem muito grandes, então a estratégia dividir-e-conquistar pode ser reaplicada,
- Os subproblemas resultantes são do *mesmo* tipo do problema original.

Estratégia

- A reaplicação do princípio é naturalmente expressa por um algoritmo recursivo,
- Subproblemas do mesmo tipo cada vez menores são gerados até que finalmente subproblemas que são pequenos o suficiente para serem resolvidos sem a separação são produzidos.

Estratégia

- Considere a estratégia dividir-e-conquistar com a separação da entrada em dois subproblemas do mesmo tipo,
- Pode-se escrever uma abstração de controle que espelha a forma como um algoritmo baseado na estratégia parecerá,
- Por *abstração de controle* entende-se um procedimento cujo fluxo de controle é claro mas cujas operações primárias são especificadas por outros procedimentos cujos significados precisos são indefinidos,
- DAC é inicialmente chamado como DAC(P), onde P é o problema a ser resolvido.

DAC(P)

- PEQUENO(P) é uma função booleana que determina se o tamanho da entrada é pequeno o suficiente para que a resposta possa ser computada sem dividir a entrada,
- Se isso for verdade, a função S é chamada,
- De outra forma, o problema P é dividido em subproblemas menores,
- Estes subproblemas p_1,p_2,...p_k, são resolvidos por aplicações recursivas de DAC,
- COMBINE é uma função que determina a solução para P usando as soluções para os k subproblemas,

DAC(P)

```
Algoritmo DAC(P)
{
 Se PEQUENO(P) então retorne S(P);
 senão
 {
   Divida P em instâncias menores p_1, p_2, ... p_k,  k ≥ 1;
   Aplique DAC a cada um destes subproblemas;
   retorne COMBINE(DAC(p_1), DAC(p_2), ..., DAC(p_k));
 }
}
```

DAC(P)

- Se o tamanho de P é n e os tamanhos dos k subproblemas são n_1, n_2, ..., n_k, respectivamente, então o tempo de computação de DAC(P) é descrito pela **relação (equação) de recorrência**:

$$T(n) = \begin{cases} g(n) & n \text{ pequeno} \\ T(n_1) + T(n_2) + ... + T(n_k) + f(n) & n \text{ grande} \end{cases}$$

- onde $T(n)$ é o tempo para DAC em qualquer entrada de tamanho n e $g(n)$ é o tempo para computar a resposta para entradas pequenas,
- A função $f(n)$ é o tempo para dividir P e combinar as soluções para os subproblemas.

Recorrências

- Para algoritmos baseados em dividir-e-conquistar que produzem subproblemas do mesmo tipo do problema original, é muito natural descrever estes algoritmos usando **recursão**.
- A complexidade de muitos algoritmos dividir-e-conquistar é dada por recorrências da forma:

$$T(n) = \begin{cases} T(1) & n = 1 \\ aT(n/b) + f(n) & n > 1 \end{cases}$$

- onde a e b são constantes conhecidas. Assume-se que $T(1)$ é conhecido e n é uma potência de b (isto é, $n = b^k$).

Método da substituição

- Um dos métodos para resolver tal relação de recorrência é chamado de *método da substituição*,
- Este método repetidamente faz substituições para cada ocorrência da função T do lado direito até que todas as ocorrências desapareçam,
- **Exemplo**: Considere o caso no qual $a = 2$ e $b = 2$. Seja $T(1) = 2$ e $f(n) = n$. Tem-se:

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2[2T(n/4) + n/2] + n \\ &= 4T(n/4) + 2n \\ &= 4[2T(n/8) + n/4] + 2n \\ &= 8T(n/8) + 3n \\ &\dots \end{aligned}$$

Método da substituição

- Em geral, nota-se que $T(n) = 2^i T(n/2^i) + in$, para qualquer $log\ n \geq i \geq 1$,
- Em particular, então, $T(n) = 2^{log\ n} T(n/2^{log\ n}) + n\ log\ n$, correspondente a escolha de $i = log\ n$. Portanto, $T(n) = nT(1) + n\ log\ n = n\ log\ n + 2n$.

Sumário

Estratégia

- Outro Exemplo: Seja $T(n) = T(n-1) + t_2$. Pode-se escrever $T(n-1) = T(n-1-1) + t_2$, desde que $n > 1$.
- Como $T(n-1)$ aparece no lado direito da primeira equação, pode-se substituir o lado direito inteiro da última equação,
- Repetindo o processo, chega-se a:

$$\begin{aligned} T(n) &= T(n-1) + t_2 \\ &= (T(n-2) + t_2) + t_2 \\ &= T(n-2) + 2t_2 \\ &= (T(n-3) + t_2) + 2t_2 \\ &= T(n-3) + 3t_2 \\ &\dots \end{aligned}$$

Estratégia

- O próximo passo requer certa intuição. Pode-se tentar obter o padrão emergente. Neste caso, é óbvio: $T(n) = T(n-k) + kt_2$, onde $1 \leq k \leq n$.
- Se houver dúvidas sobre nossa intuição, sempre pode-se provar por indução:
  - **Caso Base**: Para $k = 1$, a fórmula é correta: $T(n) = T(n-1) + t_2$.
  - **Hipótese Indutiva**: Assuma que $T(n) = T(n-k) + kt_2$ para $k = 1, 2, \ldots$. Assim, $T(n) = T(n-l) + lt_2$.
- Note que usando a relação de recorrência original pode-se escrever $T(n-l) = T(n-l-1) + t_2$, para $l \leq n$.

Estratégia

- Logo:

$$\begin{aligned} T(n) &= T(n-l-1)+t_2+lt_2 \\ &= T(n-(l+1))+(l+1)t_2 \end{aligned}$$

- Portanto, por indução em l, a fórmula está correta para todo $0 \leq k \leq n$.
- Portanto, mostrou-se que $T(n) = T(n-k)+kt_2$, para $1 \leq k \leq n$. Agora, se n é conhecido, pode-se repetir o processo até que se tenha $T(0)$ do lado direito.
- O fato de que n é desconhecido não deve ser impeditivo: consegue-se $T(0)$ do lado direito quando $n-k=0$. Isto é, $k=n$. Fazendo $k=n$ tem-se

$$\begin{aligned} T(n) &= T(n-k)+kt_2 \\ &= T(0)+nt_2 \\ &= t_1+nt_2 \end{aligned}$$

Sumário

Big-Oh

- Como as relações de recorrência podem ser usadas para ajudar a determinar o tempo de execução (big-Oh) de funções recursivas,
- Uma função com, características similares: qual é a complexidade assintótica da função FACACOISA mostrada abaixo? Por que?
- Assuma que a função COMBINE roda no tempo $\mathcal{O}(n)$ quando $|left - right| = n$, i.e., quando COMBINE é usada para combinar n elementos no vetor a.

Big-Oh

```
void FacaCoisa(int a[], int left, int right)
// póscondição:  a[left] <= ...  <= a[right]
{
 int mid = (left+right)/2;
 if (left < right)
 {
   FacaCoisa(a,left,mid);
   FacaCoisa(a,mid+1,right);
   Combine(a,left,mid,right);
 }
}
```

- Esta função é uma implementação do algoritmo de ordenação *merge sort*.
- A complexidade do *merge sort* é $\mathcal{O}(n \log n)$ para um vetor de n elementos.

A Relação de Recorrência

- Seja $T(n)$ o tempo para FACACOISA executar em um vetor de n elementos, i.e., quando $|left - right| = n$.
- Veja que o tempo para executar um vetor de um elemento é $\mathcal{O}(1)$, tempo constante.
- Tem-se então o seguinte relacionamento:

$$T(n) = \begin{cases} 2T(n/2) + \mathcal{O}(n) & \text{o } \mathcal{O}(n) \text{ é para COMBINE} \\ \mathcal{O}(1) \end{cases}$$

- Este relacionamento é chamado de *relação de recorrência* porque a função $T(...)$ ocorre em ambos os lados de "=".
- Esta relação de recorrência descreve completamente descreve a função FACACOISA, tal que se se resolver a relação de recorrência pode-se saber a complexidade de FACACOISA já que $T(n)$ é o tempo para executar FACACOISA.

Caso base

- Quando se escreve uma relação de recorrência, deve-se escrever duas equações: uma para o caso geral e uma para o caso base.
- As equações referem-se à função recursiva a qual a recorrência se aplica.
- O caso base é normalmente uma operação $\mathcal{O}(1)$, apesar de poder ser diferente.
- Em algumas relações de recorrência o caso base envolve entrada de tamanho um, tal que escreve-se $T(1) = \mathcal{O}(1)$.
- Entretanto há casos em que o caso base tem tamanho zero.
- Em tais casos, a base poderia ser $T(0) = \mathcal{O}(1)$.

Resolvendo relações de recorrência

- Pode-se realmente resolver a relação de recorrência dada no slide anterior:
  - Escreve-se n em vez de $\mathcal{O}(n)$ na primeira linha para simplificar.

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2[2T(n/4) + n/2] + n \\ &= 4T(n/4) + 2n \\ &= 4[2T(n/8) + n/4] + 2n \\ &= 8T(n/8) + 3n \\ &\cdots \\ &= 2^k T(n/2^k) + kn \end{aligned}$$

Resolvendo relações de recorrência

- Note que a última linha é derivada observando um padrão – esta é a "sacada" – generalização de padrões matemáticos como parte do problema.
- Sabe-se que $T(1) = 1$ e esta é uma forma de terminar a derivação. Na verdade, deseja-se que $T(1)$ apareça do lado direito do sinal de "=".
- Isto significa que se quer $n/2^k = 1$ ou $n = 2^k$ ou $log\ n = k$.
- Continuando com a derivação anterior, tem-se o seguinte já que $k = log\ n$

$$\begin{aligned} &= 2^k T(n/2^k) + kn \\ &= 2^{log\ n} T(1) + (log\ n)n \\ &= n + n\ log\ n \qquad \text{[lembre-se que } T(1) = 1\text{]} \\ &= \mathcal{O}(n\ log\ n) \end{aligned}$$

Resolvendo relações de recorrência

- Resolveu-se a relação de recorrência e sua solução é o que se esperava.
- Para tornar isto uma prova formal, seria necessário usar indução para mostrar que $\mathcal{O}(n \log n)$ é a solução para a dada relação de recorrência,
- Mas o método "rápido" mostrado acima mostra como derivar a solução,
- A verificação subsequente que esta é a solução é deixado para algoritmos mais avançados.

Referências I


Astrachan, O. L.
Big-Oh for Recursive Functions: Recurrence Relations.
http://www.cs.duke.edu/~ola/ap/recurrence.html


Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C.
Algoritmos - Teoria e Prática.
Ed. Campus, Rio de Janeiro, Segunda Edição, 2002.


Horowitz, E., Sahni, S. Rajasekaran, S.
Computer Algorithms.
Computer Science Press, 1998.

Referências II

Pardo, T. A. S.
Análise de Algoritmos. SCE-181 Introdução à Ciência da Computação II.
Slides. Ciência de Computação. ICMC/USP, 2008.

Preiss, B. R.
Data Structures and Algorithms with Object-Oriented Design Patterns in C++. 1999.
http://www.brpreiss.com/books/opus4/html/page41.html\#SECTION003151000000000000000

Referências III

Rosa, J. L. G.
Análise de Algoritmos - parte 2 e Divisão e Conquista. SCC-201 Introdução à Ciência da Computação II (capítulo 3).
Slides. Ciência de Computação. ICMC/USP, 2009.