PCC104 - Projeto e Análise de Algoritmos

Marco Antonio M. Carvalho

Departamento de Computação
Instituto de Ciências Exatas e Biológicas
Universidade Federal de Ouro Preto

Conteúdo

Fonte

Este material é baseado nos livros

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.
- S. Halim. *Competitive Programming*. 3rd Edition, 2013.
- Ian Parberry and William Gasarch. *Problems on Algorithms*. Second Edition, 2002.
- Ian Parberry *Lecture Notes on Algorithm Analysis and Complexity Theory*. Fourth Edition, 2001.

Licença

Divisão e Conquista

Ian Parberry e William Gasarch, *Problems on Algorithms*

"*Divide-and-Conquer is perhaps the most commonly used algorithm design technique in computer science.*

Faced with a big problem P, divide it into smaller subproblems, solve these subproblems, and combine their solutions into a solution for P.

But how do you solve the smaller problems?

Simply divide each of the small problems into smaller problems, and keep doing this until the problems become so small that it is trivial to solve them.

Sound like recursion? Not surprisingly, a recursive procedure is usually the easiest way of implementing divide-and-conquer"

Definição

Divisão e Conquista (ou Dividir e Conquistar, ou ainda, D&C) é um paradigma de solução de problemas no qual tentamos simplificar a solução do problema original dividindo-o em subproblemas menores e resolvendo-os (ou "conquistando-os") separadamente.

O processo:

- **Dividir** o problema original em **subproblemas** – normalmente com a metade (ou algo próximo disto) do tamanho do problema original, porém com a mesma estrutura;
- **Conquistar**, ou determinar a solução dos subproblemas, comumente, de maneira recursiva – que agora se tornam mais "fáceis";
- Se necessário, **combinar** as soluções dos subproblemas para produzir a solução completa para o problema original.

Observação

Conforme descrito anteriormente, podemos resolver subproblemas de estrutura igual de maneira recursiva.

Eventualmente, é necessário resolver alguns subproblemas diferentes do problema original em termos de estrutura.

Consideramos estes subproblemas como parte do processo de **combinar** as soluções.

Utilização

Existem quatro condições que indicam se o paradigma D&C pode ser aplicado com sucesso:

- Deve ser possível dividir o problema em subproblemas;
- A combinação de resultados deve ser eficiente;
- Os subproblemas devem possuir tamanhos parecidos dentro de um mesmo nível;
- A solução dos subproblemas são operações repetidas ou correlacionadas.

Utilização

Classicamente, o paradigma D&C é utilizado em vários algoritmos de ordenação:

- *Quick Sort*;
- *Merge Sort*;
- *Heap Sort*.

Outro caso de sucesso é a Busca Binária, embora este algoritmo não resolva todos os subproblemas nem combine os resultados.

A maneira como os dados são organizados em estruturas como Árvore Binária de Busca, *Heap* e Árvore de Segmentos também possui o espírito do D&C.

D&CGenerico(x)
Entrada: (sub)problema x
se x *é o caso base* **então**
  retorna *resolve*(x);
senão
  Divida x em n subproblemas $x_0, x_1, \ldots, x_{n-1}$;
  para $i \leftarrow 0$ ***até*** $n-1$ **faça**
    $y_i \leftarrow$ **D&CGenerico**(x_i);
  fim
  Combine $y_0, y_1, \ldots, y_{n-1}$ em y;
  retorna y;
fim

Recorrências e D&C

As recorrências são intimamente relacionadas com o D&C, dado que caracterizam naturalmente o tempo de execução destes algoritmos.

Por exemplo, para o *Merge Sort* temos que:

- $T(1) = 1$;
- $T(n) = 2T(n/2) + n$, para $n > 1$.

Dividir ao Meio?

Um algoritmo D&C pode dividir um problema em partes de diferentes tamanhos, como 2/3 e 1/3.

Supondo que o processo de combinar soluções seja linear, teríamos $T(n) = T(2n/3) + T(n/3) + n$.

Dividir ao Meio? (cont.)

Ainda, não necessariamente o tamanho do subproblema será uma fração do problema original.

Por exemplo, a versão recursiva de busca linear cria a cada passo um único subproblema que difere em tamanho do problema original por apenas um elemento.

Supondo um tempo constante adicional às chamadas recursivas, temos que $T(n) = T(n-1) + 1$.

Teorema Mestre

Para resolver a maior parte das recorrências associadas com algoritmos D&C podemos utilizar o **Teorema Mestre**, dada a estrutura das mesmas.

Maximum-Subarray Problem

O *Maximum-Subarray Problem* é um conhecido problema computacional que pede que seja encontrada a subsequência contígua de números cuja soma seja máxima em uma determinada sequência de números.

Consideremos a sequência -2, 1, -3, 4, -1, 2, 1, -5, 4. A subsequência contígua de números de soma máxima é 4, -1, 2, 1, cuja soma é 6.

Este problema foi proposto originalmente em 1977, por *Ulf Grenander* como um modelo simplificado para estimativa de máxima verossimillhança entre padrões em imagens digitalizadas.

Investindo em Ações

Para ilustrar, suponha que estamos interessados em investir em ações.

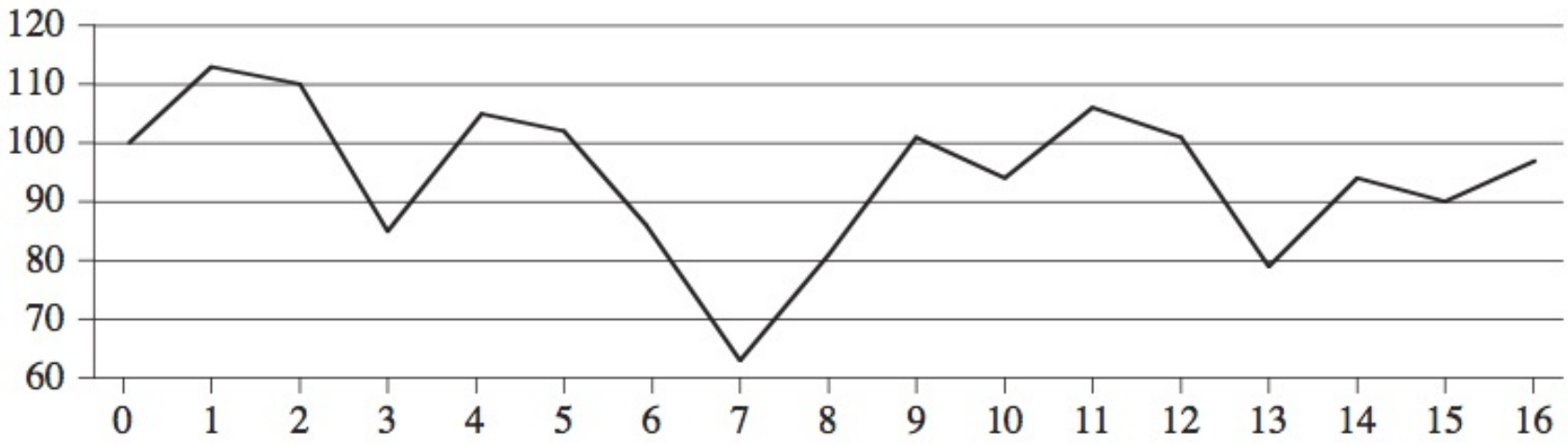
Seguiremos a estratégia "comprar na baixa, vender na alta" e, óbvio ululante, queremos maximizar nossos lucros.

Consideremos algumas simplificações:

- Só podemos comprar e vender uma única ação;
- A compra e a venda só podem ser realizadas após o fechamento do dia, quando o valor da ação não pode mais variar;
- Em compensação, possuímos o dom de "prever" o valor da ação nos dias seguintes.

Investindo em Ações

Informações sobre o preço das ações após o fechamento em um período de 17 dias.
O eixo vertical indica o preço e o eixo horizontal indica o dia.
Na tabela, a última linha indica a mudança do valor em relação ao dia anterior.



Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Change		13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

Compra na Baixa, Venda na Alta

Podemos ser levados a pensar que a maximização do lucro está ligada a estratégia de comprar pelo preço mais baixo e vender pelo mais alto, desde que possível.

Assim, bastaria encontrar estes dois picos e, a partir do maior preço, andar para a esquerda (ou do menor preço para a direita) tentando encontrar o intervalo que obtivesse a maior diferença.

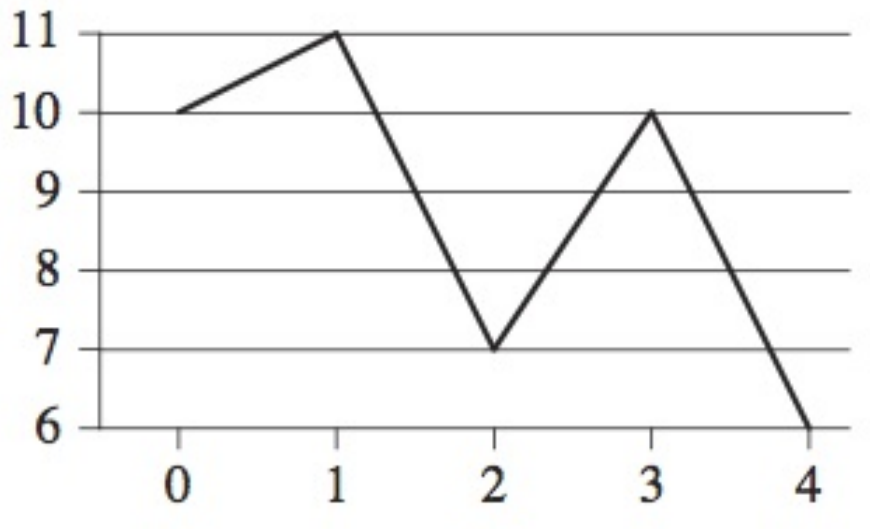
No exemplo anterior, não é possível comprar as ações pelo menor preço possível (depois do dia 7, \$63) e vender pelo maior preço possível, (depois do dia 1, \$113).

O slide a seguir nos mostra outro contra-exemplo para esta estratégia gulosa.

Contra-Exemplo

Um exemplo que nos mostra que o lucro máximo não necessariamente começa no menor preço (dia 4, \$6) ou termina no maior preço (dia 1, \$11).

Com efeito, o lucro de \$3 é obtido pela compra após o dia 2 (\$7) e venda após o dia 3 (\$10).



Day	0	1	2	3	4
Price	10	11	7	10	6
Change		1	−4	3	−4

Força Bruta

Um algoritmo força bruta para este problema pode ser facilmente projetado: testamos cada par possível de datas para compra e venda, tal que a data da compra anteceda a data da venda.

Análise

Existem $\binom{n}{2}$ pares de datas conforme descrito, o que é $\Theta(n^2)$.

Desta forma, nosso melhor algoritmo seria avaliar cada par de datas em tempo constante, o que resultaria em $\Omega(n^2)$.

Dá para fazer melhor?

Mudando a Perspectiva

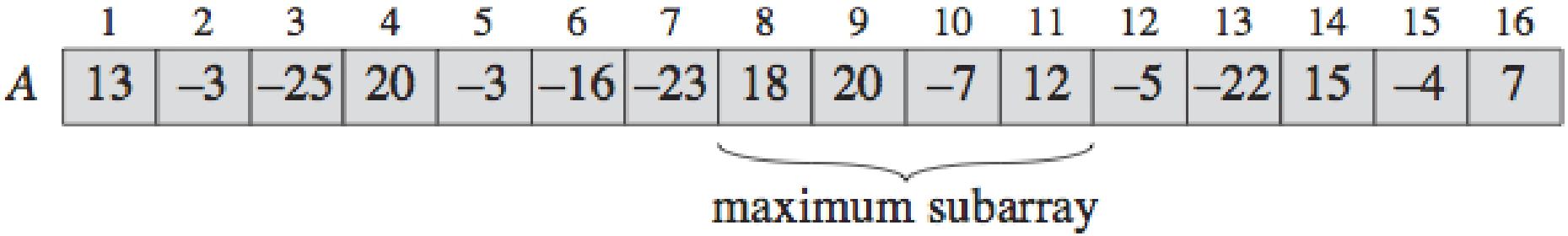
Para tentarmos diminuir a complexidade do algoritmo, vamos considerar a entrada de uma maneira diferente.

Podemos nos concentrar apenas na diferença de preços entre os dias, ao invés de nos concentrarmos nos preços diários.

Desta forma, a diferença no dia i é a diferença entre o preço da ação entre os dias i-1 e i.

Com esta transformação, estaremos interessados em determinar uma subsequência contígua e não vazia de elementos que possua a maior soma, ou seja, estamos lidando com o *Maximum-Subarray Problem*.

Divisão e Conquista



A subsequência de soma máxima é $A[8, .., 11]$, cuja soma é 43.

Sendo assim, compraríamos as ações depois do dia 7 e as venderíamos depois do dia 11, com lucro de $43.

Análise

À primeira vista, a transformação parece não ajudar, já que existem $\binom{n-1}{2} = \Theta(n^2)$ subsequências em um período de n dias.

Entretanto, pode ser mostrado que, embora o custo de computar uma subsequência seja linear em seu comprimento, quando computamos todas as $\Theta(n^2)$ subsequências, podemos organizar a computação tal que ela é feita em tempo $O(1)$, utilizando os valores computados anteriormente.

Assim, o tempo do algoritmo de força bruta seria limitado por $\Theta(n^2)$.

Projeto – *Maximum-Subarray Problem*

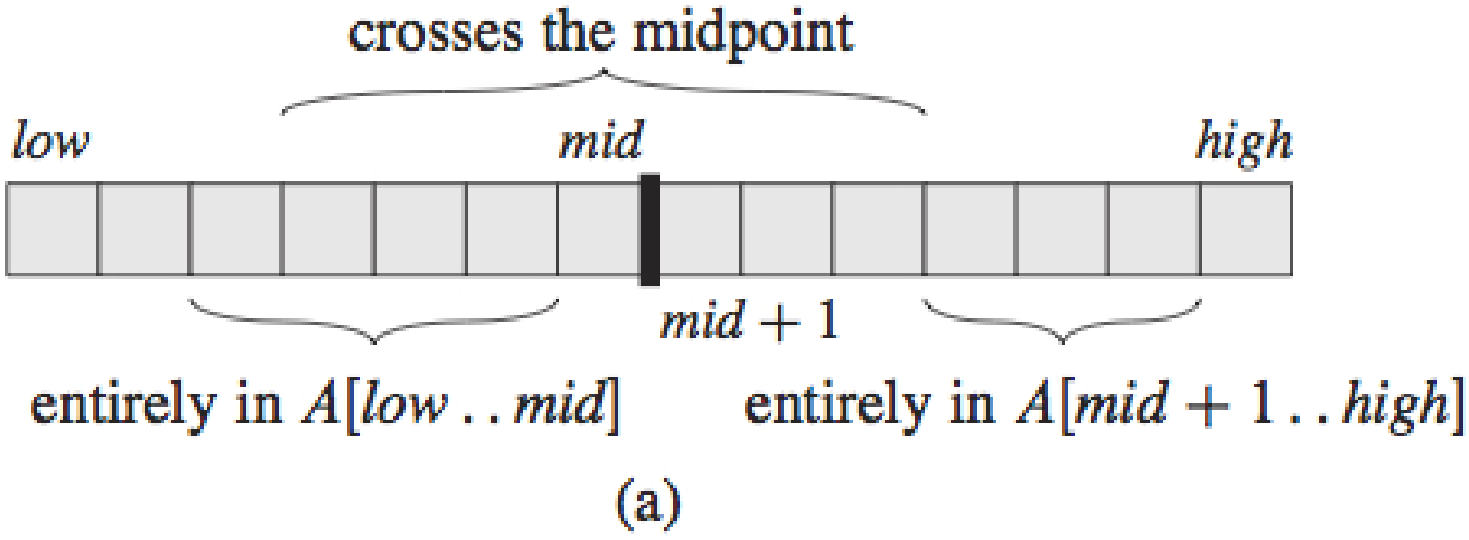
O paradigma D&C sugere que, se temos uma sequência A[*inicio*, .., *fim*], então a dividamos em duas partes, se possível, de mesmo tamanho.

Assim, teríamos subproblemas A[*inicio*, .., *meio*] e A[*meio*+1, .., *fim*].

Qualquer subsequência $A[i, .., j]$ deve cair em exatamente uma das situações abaixo:

- Completamente na primeira metade A[*inicio*, .., *meio*], tal que *inicio* $\leq i \leq j \leq$ *meio*;
- Completamente na segunda metade A[*meio*+1, .., *fim*], tal que *meio* $< i \leq j \leq$ *fim*; ou
- Cruzando o ponto médio, tal que *inicio* $\leq i \leq$ *meio* $< j \leq$ *fim*.

Sendo assim, a subsequência de soma máxima deve possuir a soma maior do que todas as subsequências em A[*inicio*, .., *meio*], em A[*meio*+1, .., *fim*] e também cruzando o ponto médio.



Possíveis situações para a subsequência.

Projeto – *Maximum-Subarray Problem*

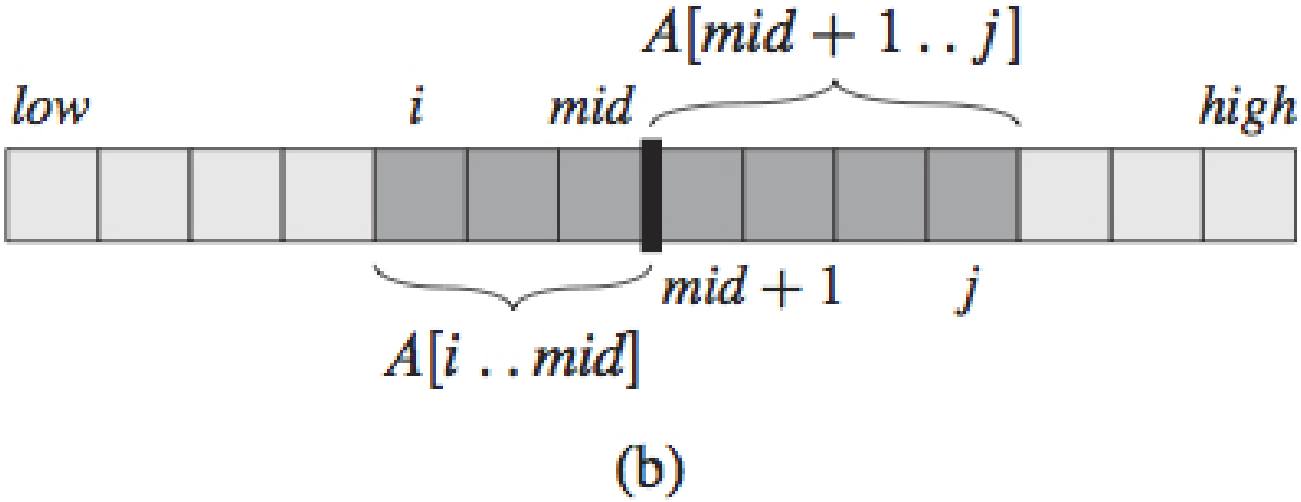
Podemos determinar as subsequências de soma máxima de $A[inicio, .., meio]$ e $A[meio + 1, .., fim]$ recursivamente, uma vez que estes dois subproblemas são cópias exatas do problema original.

O que nos faltaria é determinar a subsequência de soma máxima que cruza o ponto médio, depois, determinar dos três qual é o maior.

Podemos facilmente determinar subsequência de soma máxima que cruza o ponto médio, em tempo linear no tamanho $A[inicio, .., fim]$.

Podemos, por exemplo, determinar as subsequências de forma $A[i, .., meio]$ e $A[meio + 1, .., j]$ e depois combiná-las.

O algoritmo a seguir recebe como entrada a sequência A e os índices do início, meio e fim, retornando os índices que delimitam a subsequência de soma máxima que cruze o ponto médio entre dois subproblemas.



(b)

Exemplo de subsequência que cruza o ponto médio.

Divisão e Conquista

```
1  FindMaxCrossingSubarray(A, inicio, meio, fim)
   Entrada: Sequência A, índices inicio, meio e fim
2  somaEsquerda← −∞; soma← 0;
3  para i ← meio até inicio faça
4  |   soma← soma + A[i];
5  |   se soma > somaEsquerda então
6  |   |   somaEsquerda← soma;
7  |   |   maxEsquerda← i;
8  |   fim
9  fim
10 somaDireita← −∞; soma← 0;
11 para j ← meio + 1 até fim faça
12 |   soma← soma + A[j];
13 |   se soma > somaDireita então
14 |   |   somaDireita← soma;
15 |   |   maxDireita← j;
16 |   fim
17 fim
18 retorna maxEsquerda, maxDireita, somaEsquerda+somaDireita;
```

Projeto – *Maximum-Subarray Problem*

O algoritmo apresentado primeiro determina a subsequência de soma máxima do meio para o início, depois, do meio para o fim.

Ambas subsequências possuem o elemento do meio, desta forma, as combinamos para gerar uma única subsequência de soma máxima que cruza o ponto médio.

A cada iteração, os dois laços possuem custo constante.

Desta forma, o algoritmo é linear no número de elementos, ou seja, $\Theta(n)$.

De posse deste algoritmo, podemos projetar um algoritmo D&C para o *Maximum-Subarray Problem*.

Divisão e Conquista

FindMaxSubarray(A, inicio, fim)
Entrada: Sequência A, índices $inicio$ e fim
se *fim* = *inicio* **então**
  retorna *inicio, fim,* A*[inicio];//um único elemento*
senão
  meio $\leftarrow \lfloor (inicio + fim)/2 \rfloor$;
  (inicioEsquerda, fimEsquerda, somaEsquerda)$\leftarrow$ **FindMaxSubarray**(A, inicio, meio);
  (inicioDireita, fimDireita, somaDireita)$\leftarrow$ **FindMaxSubarray**(A, meio+1, fim);
  (inicioCruzado, fimCruzado, somaCruzado)$\leftarrow$ **FindMaxCrossingSubarray**(A, inicio, meio, fim);
  se *somaEsquerda* $\geq$ *somaDireita* **e** *somaEsquerda* $\geq$ *somaCruzado* **então**
    retorna *inicioEsquerda, fimEsquerda, somaEsquerda;*
  senão
    se *somaDireita* $\geq$ *somaEsquerda* **e** *somaDireita* $\geq$ *somaCruzado* **então**
      retorna *inicioDireita, fimDireita, somaDireita;*
    senão
      retorna *inicioCruzado, fimCruzado, somaCruzado;*
    fim
  fim
fim

Projeto – FindMaxSubarray

As chamadas recursivas de **FindMaxCrossingSubarray** retornam uma tupla que delimita os elementos da subsequência de soma máxima, além do valor da soma máxima.

O caso base é termos apenas um elemento, que por si só é uma subsequência de soma máxima.

O passo recursivo determina o meio do subproblema e o divide para determinar as subsequências de soma máxima do lado esquerdo e do lado direito.

O passo de combinação determina a subsequência de soma máxima que cruza o ponto médio do problema e depois determina dentre as três qual possui a maior soma

Exercício

1. Execute o algoritmo **FindMaxSubarray** para a instância $A = [20, -7, 12, -5, -22, 15, -4, 7]$.
2. Determine e resolva a recorrência e determine a complexidade de **FindMaxSubarray**.

Análise – FindMaxSubarray

Para simplificarmos a análise do algoritmo, consideramos que o tamanho do problema é uma potência de 2, de forma que o tamanho de todos os problemas é inteiro e par.

Seja $T(n)$ o tempo de execução de **FindMaxSubarray**:

- $T(1) = 1$, uma vez que o caso base possui apenas operações de tempo constante;
- No passo recursivo, dividimos o problema em duas partes iguais, então necessitamos de $T(\frac{n}{2})$ de tempo para resolver cada um deles;
- Como resolvemos 2 subproblemas em tempo linear, temos $2T(\frac{n}{2})$;
- Como vimos, **FindMaxCrossingSubarray** possui tempo $\Theta(n)$;
- Determinar a subsequência de maior soma, dentre as três calculadas é feito em tempo constante ($\Theta(1)$).

Análise – FindMaxSubarray

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

Recorrência – FindMaxSubarray

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1, \\ 2T(\frac{n}{2}) + \Theta(n) & \text{se } n > 1 \end{cases}$$

Maximum-Subarray Problem – Comparação dos Algoritmos

n	$O(n^3)$	$O(n^2)$	$O(n\ log\ n)$	$O(n)$
10^0	3,10 μs	3,10 μs	3,20 μs	3,10 μs
10^1	133,10 μs	39,00 μs	73,40 μs	9,40 μs
10^2	37,00 ms	2,63 ms	3,08 ms	93,00 μs
10^3	32,30 s	243,00 ms	247,18 ms	930 μs
10^4	6,53 h	23,00 s	22,88 s	7,8 ms
10^5	272,01 d	51,67 m	20,95 m	78 ms
10^6	745,18 a	3,59 d	29,55 h	750 ms

Valores calculados por extrapolação.

Os quatro algoritmos foram implementados na linguagem de programação Python. A execução foi realizada em uma máquina com processador Intel(R) Corel(TM)2 DUO CPU P8600 @ 2.40GHz 1,58 GHz, Memória RAM de 1,89 GB e sistema operacional Microsoft Windows XP Professional Versão 2002 Service Pack 3.

Divisão e Conquista

Vantagens

Torna problemas difíceis mais fáceis pela diminuição do tamanho.

Tendência a complexidade logarítmica.

Paralelismo facilitado no processo de "conquista".

Em computação aritmética, traz resultados mais precisos em termos de controle de arredondamento.

Desvantagens

O número de chamadas recursivas pode ser um inconveniente para o desempenho.

Eventual dificuldade para selecionar o caso base.

Redundância de resolução de subproblemas repetidos, o que pode ser resolvido através do uso de **memoização**.

Comentários Finais

Começamos a ter uma idéia de quão poderoso pode ser o paradigma D&C.

Vimos como a D&C pode nos permitir projetar algoritmos assintoticamente mais rápidos do que Força Bruta.

Para alguns problemas, os melhores algoritmos existentes são D&C.

Em outros casos, podemos fazer ainda melhor. Com efeito o melhor algoritmo para o *Maximum-Subarray Problem* possui complexidade linear e não usa D&C.

Dúvidas?