Capítulo 6

Divisão e Conquista

" A nossa experiência mostra que a legibilidade é o único e melhor critério para medir a qualidade de um programa: se um programa é fácil de ler, ele é provavelmente um bom programa; se ele é difícil de ler, provavelmente ele não é bom"

- Kernighan e Plauger -

Sumário:

7.1- Conceitos

A estratégia de Divisão e Conquista ou **Recursão em árvore**, foi inventada por Napoleão Bonaparte na batalha de Austerliz em 1805. Nessa batalha, o exército francês estava em inferioridade numérica, e necessitava de derrotar os exércitos russos e austríacos que estavam concentrados no campo de batalha como se fossem um único exército. Napoleão mandou o seu exército atacar no centro, tendo criado um caos e perplexidade no imigo, dividindo o seu exército em dois e derrotando-os separadamente. O método de desenvolvimento de algoritmos recursivos por divisão e conquista, reflete esta estratégia militar.

Dada uma amostra de um problema de tamanho n, o método de divisão e conquista, consiste em dividir essa amostra em k subamostras disjuntas ($1 \leq k \leq n$) que correspondem a k subproblemas distintos. Resolver esses subproblemas de forma separada, e combinar as soluções parciais para encontrar a solução original.

Como as subamostras são do tipo da amostra original, faz sentido utilizar uma solução recursiva, para proceder a divisão de cada subamostra, até obter-se uma subamostra tão pequena, cuja solução é trivial.

Em termos gerais, esse algoritmo baseia-se no seguinte esquema:

se o problema for trivial **então**
Resolva directamente;
senão
início
Fragmente o problema em duas amostras aproximadamente iguais;
Aplique o método à cada um das amostras;
Combine as soluções parciais para resolver o problema original;
fim;

7.2 - Exemplos

Um exemplo clássico da aplicação deste método, é a sequencia de Fibonacci. Esta sequência, foi descoberta pelo matemático italiano Leonardo Fibonacci (1170-1250), também conhecido como Leonardo de Pisa, quanto estudava o crescimento de uma população de coelhos.

O problema consistia em saber, quantos casais de coelhos poderão ser obtidos na n-ésima geração, se em cada mês, cada casal reproduzir um novo casal, que se torna fértil a partir do 2º mês. Vamos considerar que não ocorrerão mortes e temos um único casal de coelhos.

Leonard de Pisa, descobriu que esse crescimento era descrito pela seguinte sequência:

0,1, 1, 2, 3, 5, 8,13, 21 ,…

que mais tarde recebeu o seu nome em sua homenagem.

Pela composição da sequencia, facilmente se constacta que o caso base, acontece quando n = 0 e n = 1. Para esses casos:

$$\text{fibo }(0) = 0 \quad \text{e} \quad \text{fibo }(1) = 1;$$

Vamos determinar o passo recursivo, com a definição de uma fórmula recorrente que expressa n-ésimo termo de Fibonacci como combinação de termos de Fibonacci anteriores. Sabemos que:

$$\begin{aligned} n = 2 \quad \text{fibo }(2) &= 1 \\ &= 1 + 0 \\ &= \text{fibo}(0) + \text{fibo}(1) \end{aligned}$$

$$\begin{aligned} n = 3 \quad \text{fibo }(3) &= 2 \\ &= 1 + 1 \\ &= \text{fibo}(2) + \text{fibo}(1) \end{aligned}$$

n = 4   fibo (4) = 3
= 2 + 1
= fibo(3) + fibo(2)

Logo, podemos concluir que para qualquer que seja n > 1, temos:

fibo(n) = fibo(n-1) + fibo(n-2)

Logo, estamos em condições de desenvolver uma função recursiva para calcular o n-ésimo termo fibonacci, para qualquer que seja n ≥ 2.

```
int fibo (int n)
 {
   if ( n <= 1)
      return n;
    return fibo ( n-1) + fibo (n-2);
 }
```

Para valores de n muito pequenos, esta função resolve o problema num intervalo de tempo aceitável, mas para valores muito grandes o tempo de processamento não é aceitável, porque são efectuadas muitas chamadas recursivas. Esse elevado número de chamadas recursivas deve-se a recursão dupla existente no passo da recursão.

Vamos mostrar, através de uma simulação, a quantidade de chamadas recursivas que essa função executa. Para tornar essa simulação perceptível, vamos inserir no código, algumas mensagens e declarar uma variável auxiliar que irá mostrar o valor do termo de Fibonacci gerado.

```
int fibo (int n)
 {
   int f;
   printf (" Entrar em fibo(%d) \n ", n);
   if ( n <= 1 )
        f = n;
   else
        f =  fibo (n-1) + fibo (n-2);
    printf ( " sair de fib(%d), Retornar = %d \n ", n, f);
    return F;
 }
```

Suponhamos, sem perda da generalidade que n = 4. Embora esse número inteiro é muito pequeno, veja a quantidade de chamadas recursivas e de mensagens que a função realiza.

Entrar em Fibo (4)
Entrar em Fibo (3)
Entrar em Fibo (2)
Entrar em Fibo(1)
Sair de Fibo (1), Retorna = 1
Entrar em Fibo (0)
Sair de Fibo (0), Retorna = 0
Sair de Fibo (2), Retorna = 1
Entrar em Fibo (1)
Sair de Fibo (1), Retorna = 1
Sair de Fibo (3), Retorna = 2
Entrar em Fibo (2)
Entrar em Fibo (1)
Sair de Fibo (1), Retorno = 1
Entrar em Fibo (0)
Sair de Fibo (0) , Retorno = 0
Sair de Fibo (2) , Retorno = 1
Sair de Fibo (4) , Retorno = 3

Ao contrário do processo recursivo linear estudado no capítulo anterior, estamos perante a existência de múltiplas fases de expansão e de contracção, que são originadas pela dupla recursão existente na função fibo().

A título ilustractivo, mostramos a árvore de recursão gerada por este processo para n = 4.

```
                      fibo(4)
                   /          \
              fibo(3)          fibo(2)
             /      \          /      \
        fibo(2)   fibo(1) fibo(1)    fibo(0)
         /    \       |       |          |
   fibo(1) fibo(0)    1       1          0
      |        |
      1        0
```

7.3 - Operações Recursivas com Vectores

Seja T = (v_0, t_1, t_2, …, $t_{nElem-1}$) um conjunto de números inteiros armazenado num vector, tal que 0< nElem <= M, sendo M a dimensão do vector Esse conjunto pode ser decomposto em dois subconjuntos disjuntos com a mesma dimensão. Um subconjunto T_i = (t_0, t_1, …, t_m) e outro subconjunto T_j = (t_{m+1}, t_{m+2}, …, $t_{nElem-1}$) onde m é o ponto médio. Como os subconjuntos não são unitários, podemos aplicar novamente o processo de decomposição até obtenção de subconjuntos unitários.

Facilmente se constacta, que esse processo de decomposição é finito, e retrata a estratégia de divisão e conquista.

7.3.1- Imprimir os Elementos de um Vector

Seja T = ($t_0,t_1,\ldots,t_{nElem-1}$) um conjunto de números inteiros armazenados num vector , tal que 0 < nElem <= M, sendo M a dimensão do vector. A resolução deste problema consiste em dividir o conjunto T em dois subconjuntos com comprimentos aproximadamente iguais:

T1= ($t_0,t_1,\ldots,t_k$) e T2=($t_{k+1},t_{k+2},\ldots,t_{nElem-1}$) onde k = (nElem - 1) / 2

Resolvemos o problema de forma separada para os subconjuntos T1 e T2, imprimindo desse modo todos os seus elementos. Então o nosso problema consiste em saber como imprimir os elementos de cada subconjunto. Se aplicarmos o método de dividir cada subconjunto ao meio até chegarmos a um conjunto unitário temos uma parte do problema resolvido. A outra parte é trivial, ela consiste em imprimir o conteúdo do conjunto unitário. Então a solução do problema original consiste na união da impressão de todos os subconjuntos unitários gerados pelo processo recursivo.

```
void imprimirElementos (float vet[ ], int i, int f )
 {
    if ( i == f )
       printf ( " %f", vet[i] );
    else
       {
          k = ( i + f )/2;
          imprimirElementos (vet , i , k);
          imprimirElementos (vet , k+1 , f);
       }
 }
```

7.3.2- Determinar o Número de Elementos de um Vector

Seja T = ($t_0,t_1,\ldots,t_{nElem-1}$) um conjunto de números inteiros armazenados num vector tal que 0 < nElem <= M. A resolução deste problema consiste em dividir o conjunto V em dois subconjuntos com comprimentos aproximadamente iguais:

T1= ($t_0,t_1,\ldots,t_k$) e T2= ($t_{k+1},t_{k+2},\ldots,t_{nElem-1}$) onde k = (nElem - 1) / 2

Resolvemos o problema de forma separada para os subconjuntos V1 e V2, obtendo, desse modo, os totais de elementos de cada subconjunto. A solução original consiste na soma das soluções parciais.

```
int totalElementos (float vet[ ], int i, int f )
{
  int total1, total2, k;
  if ( i == f )
     return 1;
  else
   {
     k = ( i + f ) /2;
     total1= totalElementos (vet , i , k);
     total2= totalElementos (vet , k+1 , f);
     return total1 + total2;
   }
}
```

Vamos optimizar esta função,

```
int totalElementos ( float vet[ ], int i, int f )
{
   int k;
   if ( i == f )
     return 1;
   else
    {
     k = ( i + f ) / 2;
     return totalElementos (vet , i , k) + totalElementos (vet , k+1, f) ;
    }
 }
```

7.3.2- Determinar o Elemento máximo de um Vector

Seja T = ($v_0, v_1, \ldots, v_{nElem-1}$) um conjunto de números inteiros armazenados num vector, tal que 0 < nElem <= M. A resolução deste problema consiste em dividir o conjunto V em dois subconjuntos com comprimentos aproximadamente iguais:

T1= ($t_0, t_1, \ldots, t_k$) e T2 =($t_{k+1}, t_{k+2}, \ldots, t_n$) onde k = (n - 1) / 2

Resolvemos o problema de forma separada para os subconjuntos T1 e T2, obtendo, desse modo, os elementos máximos de cada subconjunto. Sejam max1 e max2 esses elementos. A solução original consiste na comparação dos elementos máximos das soluções parciais.

```
int maiorElemento (float vet[ ], int i, int f )
{
  int max1, max2, k;
  if  ( i == f )
    return vet[I];
  else
    {
      k = ( i + f ) / 2;
      max1 = maximo (vet , i , k);
      max2 =  maximo (vet , k+1 , f);
       if ( max1 > max2 )
          return max1;
       else
          return max2;
    }
}
```

Mais uma vez enfatizamos que a estratégia de divisão e conquista produz algoritmos mais elegantes e mais fáceis de serem lidos e compreendidos, do que algoritmos iterativos. Contudo esses algoritmos não são mais eficientes porque de memória adicional, para fazer face as chamadas recursivas, tornando desse modo, a execução do algoritmo mais lenta.

7.4 - Recomendações Bibliográficas

Para o leitor aprofundar os seus conhecimentos sobre a estratégia de divisão e conquista e programação dinâmica, recomendamos os seguintes livros:

Cormen T. H.;- *Introduction to algorithms,* Cambridge, Massachusetts, MIT press, 2009.

Levitin A.;- *Introduction to the Design and Analysis of Algorithms,* 3rd Edition, Pearson, 2011.

Robert E. S.;- *Programming Abstraction in C*: a Second Course in Computer Science, Addison.Wesley, 1998.

Routo T. ; - *Desenvolvimento de Algoritmos e Estruturas de Dados*, McGraw_hill, Brasil, 1991.

Sedgewick R. ;- *Algoritms in C*, Massachusetts, Addison-Wesley, Reading, Massachusetts,1990.

7.5 - Exercícios

7.5.1-Dada a formula recorrente:

Lucas(n) = 2 se n = 0
Lucas(n) = 1 se n = 1
Lucas (n) = Lucas(n-1) + Lucas(n-2) se n > 2

Desenvolva uma função que recebe como parâmetro um número inteiro não negativo e retorna o n-ésimo termo de Lucas:

7.5.2-Dada a formula recorrente:

$a^n = 1$ se n = 0
$a^n = (a^{n/2})^2$ se n é par
$a^n = a \times (a^{n/2})^2$ se n é impar

Desenvolva uma função que recebe como parâmetro um número inteiro não negativo e retorna a potência desse número.:

7.5.3- Dada a fórmula recorrente.

G(n) = 2 se n = 0
G(n) =1 se n = 1
G(n) =3 se n = 2
G(n)= G(n-1) + 5G(n-2) + 3G(n-3) se n > 3

Desenvolva uma função que recebe como parâmetro um número inteiro não negativo n, e retorne o valor de G(n).

7.5.4-Dada a formula recorrente:

Tribonacci(n) = 0 se n = 0
Tribonacci(n) = 1 se n = 1ou n = 2
Tribonacci(n) = tribonacci(n-1) + tribonacci(n-2) + tribonacci(n-3) se n > 3

Desenvolva uma função para calcular o n-ésimo termo de Tribonacci.

7.5.5-Dada a formula recorrente:

1 se n ≤ 1
n + p(n/3) + p((n+1)/3 + p((n+2)/3) se n > 1

Desenvolva uma função que recebe um número inteiro não negativo e calcula o valor de n.

7.5.6-Desenvolva uma função para graduar uma régua de n centímetros de tal forma que os números múltiplos de 10 têm um traço longo, enquanto os múltiplo de cinco um traço curto, os restantes contêm um ponto.

7.5.7- Dada a seguinte função:

```
int  ff (int n, int ind)
{
   int i;
   for (i = 0; i < ind; i++)
       printf (“ ");
   printf ("ff (%d, %d) \n", n, ind);
   if (n == 1)
      return 1;
   if (n % 2==0)
     return ff (n/2, ind + 1);
   return ff ((n-1)/2, ind +1) + ff((n+1)/2, ind +1);
 }
```

Suponha que n = 7 e que ind = 0.

7.5.8-Desenvolva uma função que entre outros parâmetros recebe um vector com números inteiros e um determinado valor. Determine o valor que mais se aproxima desse valor.

7.5.9- Desenvolva uma função que entre outros parâmetros recebe dois vectores do mesmo tipo. Verificar se esses vectores são iguais.

7.5.10-Desenvolva uma função que entre outros parâmetros recebe um vector do tipo caracter e um determinado carácter. Contar o número de vezes que esse carácter está contido no vector.

7.5.11-Desenvolva uma função que entre outros parâmetros recebe dois vectores ordendos em ordem crescente. Intercalar esses vectores num terceiro, sem repetir os seus elementos. Lembre-se que os vectores originais não possuem elementos repetidos.

7.5.12-Desenvolva uma função que entre outros parâmetros recebe um vector e uma determinada posição k. Separar o vector em dois de tal forma que os elementos que se encontram nas posições de 0 à k, vão para o primeiro vector e os restantes para o segundo.

7.5.13-Desenvolva uma função que entre outros parâmetros recebe um vector e uma determinada chave. Verificar se essa chave encontra-se no vector, retorna 0 se a chave não for encontrada e 1 no caso contrário.

7.5.14-Desenvolva uma função que entre outros parâmetros recebe um vector. Contar o número de zeros existente nesse vector.

7.5.15-Desenvolva uma função que entre outros parâmetros recebe um vector. Determinar o elemento máximo e o elemento mínimo.

7.5.16-Um camionista pretende sair de Luanda para Benguela pela estrada nacional. O tanque de combustível do seu camião está cheio, e contém gasóleo suficiente para n quilômetros. O Googlemaps instalado no seu camião fornece as distâncias entre os postos de gasolina existentes nessa estrada. O motorista deseja fazer o mínimo possível de paradas para abastecimento ao longo do caminho. Desenvolva um algoritmo eficiente para que o motorista possa determinar em quais postos de gasolina deve parar.