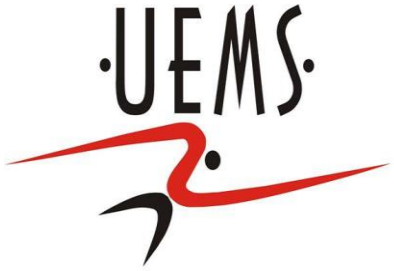


Programação de Computadores I – Recursão

Profa. Mercedes Gonzales Márquez

Recursão

- Muitos problemas computacionais têm a seguinte característica dita “recursiva” : Cada instância do problema contém uma instância menor do mesmo problema.

N! = N x N-1 X N-2 x N-3 x ... X 1

N!=N x (N-1)!

Exemplo:

7!= 7 x 6 x 5 x 4 x 3 x 2 x 1

6! = 6 x 5 x 4 x 3 x 2 x 1

7!= 7 x 6!

- A solução recursiva está diretamente ligada ao conceito de indução matemática.

Recursão

- Na indução matemática define-se a solução do problema para casos básicos e usa-se como hipótese que a solução do problema de tamanho n pode ser obtida a partir da sua solução de tamanho menor, por exemplo, n-1.
- Assim sendo, a solução de um problema é dita recursiva quando ela é escrita em função de si própria para instâncias menores do problema.

Recursão

Toda recursão é composta por:

- **Caso base**

– Uma ou mais instâncias do problema que podem ser solucionadas facilmente (solução trivial)

- **Chamadas Recursivas**

– O objeto define-se em termos de si próprio, procurando convergir para o caso base.

Por exemplo, o fatorial de um número n pode ser calculado a partir do fatorial do número n−1.

Recursão

- Esquematicamente, os algoritmos recursivos tem a seguinte forma:

  Se <condição para o caso base> então

  resolução direta para o caso base

  Senão

  uma ou mais chamadas recursivas

  Fim se

- Sem a condição de parada, expressa no caso base, uma recursão iria se repetir indefinidamente.

Recursão

- Implementação iterativa do cálculo de fatorial de um número.

Implementação Iterativa

```
long fat(long n){
long r = 1;
for(int i = n; i >0; i--)
        r = r * i;
return r;
}
```

Mas, podemos usar a propriedade recursiva do fatorial

N! = N x N-1 X N-2 x N-3 x ... X 1

N!=N x (N-1)!

Exemplo:

7!= 7 x 6 x 5 x 4 x 3 x 2 x 1

6! = 6 x 5 x 4 x 3 x 2 x 1

7!= 7 x 6!

Recursão

- A solução do problema também pode ser expressa de forma recursiva
  - Se n = 1 então n! = 1.
  - Se n > 1 então n! = n x (n-1)!
- Aplicamos o princípio da indução assim:

- Sabemos a solução para um caso base: n = 1.

- Definimos a solução do problema geral n! em termos do mesmo problema, só que para um caso menor: ((n-1)!).

Recursão

Implementação Recursiva

```
long fat(long n){
   if(n <= 1) //Caso base
       return 1;
   else //Sabendo o fatorial de (n-1)
   //calculamos o fatorial de n
      return (n* fat(n-1));
}
```

Sequência de Fibonacci – Definição Recursiva

- A série de Fibonacci é a seguinte:

  1; 1; 2; 3; 5; 8; 13; 21; ...

- Queremos determinar qual e o n-ésimo (fibo(n)) termo da série.
- Vejamos a solução iterativa deste problema.

Sequência de Fibonacci – Solução Iterativa

1; 1; 2; 3; 5; 8; 13; 21; ...

```
int fibonacci (int n){
    int i, fibo, primeiro=1, segundo=1;
    if (n<=2)
        return (1);
    else{
        for (i=3;i<=n;i++){
            fibo=primeiro+segundo;
            primeiro=segundo;
            segundo=fibo;
        }
        return (fibo);
    }
}
```

```
int main(){
    int n;
    do {
        printf ("Informe um numero inteiro
maior que zero: ");
        scanf ("%d", &n);
    }while (n<1);
    printf ("O termo %d da sequencia de
fibonacci eh %d\n", n, fibonacci(n));
}
```

Sequência de Fibonacci – Definição Recursiva

Identifiquemos a propriedade recursiva da sequência de Fibonacci

fibo(1) = 1 fibo(2) = 1

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

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

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

Assim:

- No caso base temos:

  - Se n = 1 ou n = 2 então fibo(n) = 1.

  - Sabendo casos anteriores podemos computar fibo(n) como: fibo(n) = fibo(n-1) + fibo(n-2).

Sequência de Fibonacci – Definição Recursiva

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

1, 1, 2, 3, 5, 8, 13, 21, 34, ...

Exemplo para n=5

fibo(5)= 5

Recursão

Exemplo : Faça um programa que encontre o valor do elemento máximo de um vetor v de n elementos distintos.

Solução Iterativa

```
int maximo(int n, int v[]){
    int i, max=v[0];
    for (i=1;i<n;i++)
        if (v[i]>max)
            max=v[i];
     return(max);
}
```

Recursão

Denotamos por m(n) o máximo dos elementos das posições 0 ate n-1 do vetor. Então:

- Se n = 1 então o máximo é igual a v[0].
- Se n > 1 então determinamos o máximo dos n-1 elementos (m(n-1)) e se ele for maior que o elemento v[n-1] então o máximo é m(n-1) senão o máximo será v[n-1].

Solução Recursiva

```
int maximo(int n, int v[]){
  if (n == 1)
        return v[0];
  else{
        x = maximo (n-1, v);
        if (x > v[n-1]) return x;
        else return v[n-1];
  }
```

Recursão

Exemplo : Faça um programa que faça a versão recursiva da função busca iterativa.

Solução Iterativa

```
int busca (int x, int n, int v[]) {
    int k;
    k = n-1;
    while (k >= 0 && v[k] != x)
        k -= 1;
    return k;
 }
```

Recursão

Exemplo 3: Soma de elementos de um vetor :

Faça um programa que preencha um vetor de 10 elementos inteiros, imprima o seu conteúdo, e o resultado do somatório dos seus elementos, calculado por uma função recursiva.

Solução Iterativa

```
int soma(int n, int v[]){
    int i, soma=0;
    for (i=0;i<n;i++)
        soma+=v[i];
     return(soma);
}
```

Recursão

Identifiquemos a propriedade recursiva da soma.

Denotamos por S(n) a soma dos elementos das posições 0 ate n-1 do vetor. Então:

- Se n = 1 então a soma e igual a v[0].
- Se n > 1 então a soma e igual a v[n-1] + S(n-1), ou seja,

S(n) = v[n-1] + S(n-1)

Solução Recursiva

```
int soma(int n, int v[]){
   if (n == 1)
         return (v[0]);
   else
         return (v[n-1]+soma(n-1, v));
}
```

Recursão

Exemplo 4. Escreva uma função recursiva que recebe como parâmetros um número real **X** e um inteiro **n** e retorna o valor de $\mathbf{X^n}$.

- **Obs.:n** pode ser negativo.

Identifiquemos a propriedade recursiva da potência

Se n>0 $x^n = x*x^{n-1}$

Se n<0 $x^n = 1/(x*x^{|n|-1})$

Exemplo $2^{-3} = 1/\ 2^3 = 1/\ (2*2^2) = 1/\ 8$

```
float Potencia(float x,int n){
if  ( n ==0 ) return (1);
else
   if ( n >0 ) return (x* Potencia(x, n-1));
   else return (1/(x* Potencia(x, abs(n) -1)));
}
```

Recursão

- Exemplo 5. A seguinte função é a solucao recursiva do problema de contar o número de algarismos (ou dígitos) que um número inteiro n, tem.

Exemplo o número 353636 tem 6 algarismos e o número 454 tem 3 algarismos.

Usemos divisão inteira

353636 / 10 = 35363

35363 / 10 = 3536

3536 / 10 = 353

353 / 10 = 35

35 / 10 = 3

Solução iterativa

```
int digitos(int n){
   int cont
   cont=1;
   while (abs(n)>9) {
         n = n/10;
         cont ++;
   }
   return (cont)
}
```

Recursão

Observe a propriedade recursiva da determinação do número de algarismos. Chamaremos de digitos ao número de algarismos

digitos(n) = 1 + digitos(n/10)

Exemplo:

digitos**(6549)=1 + digitos (654)**

1+digitos(65)

1+digitos(6)

1

Solução Recursiva

```
int digitos(int n){
    if (abs( n )<10)
        return (1);
    else
        return (1+ digitos(n/10));
}
```

Recursão

- Exemplo 6. Escreva uma função recursiva que calcule a soma dos dígitos de um inteiro positivo n. A soma dos dígitos de 1932, por exemplo, é 1+9+3+2=15.

Identificando a propriedade recursiva da soma de algarismos de um número inteiro positivo.

somdig(1932) = 1 + 9 + 3 + 2 = 15

1932 = 2 + 1 + 9 + 3

somdig(1932) = 2 + somdig(193)

Generalizando, então

somdig (n) = n%10 + somdig(n/10)

Recursão

somdig(52364) = 4+somdig(5236)

6+somdig(523)

3+somdig(52)

2+somdig(5)

5

```
int somdig(int n){
    if (n <10)
        return (n);
    else
        return (n%10+ somdig(n/10));
}
```

Recursão

- Exemplo 7. Escreva uma função recursiva que determine o inverso de um número inteiro positivo n, dado o número de dígitos de n. O inverso é obtido tomando os dígitos do número de direita para esquerda. Por exemplo, o inverso do número 132 é 231.

Identifiquemos a propriedade recursiva da determinação do inverso de um número

Inverso(7869) = 9000 + 600 + 80 + 7

Inverso (7869) = 9000 + inverso(786)

Inverso (n, digitos) = n%10*10^(digitos-1) + inverso (n/10, digitos-1)

Recursão

inverso(7869,4)=9000+inverso(786,3)

600+inverso(78,2)

80 +inverso(7,1)

7

```
int inverso(int n, int ndig){
   if (n <10)
        return (n);
   else
        return ((n%10)*pow(10,ndig-1)+ inverso(n/10,ndig-1));
}
```

Recursão

- Exemplo 8. Verificar se um número é capicua utilizando uma função recursiva que considere como entrada o número inteiro positivo n e o número de algarismos de n. Um número capicua é aquele número que coincide com seu inverso, de acordo a definição do exemplo 7.

Exemplo: 67876, 7890987 e 8

Identificamos a propriedade recursiva:

Verificamos se o último algarismo coincide com o primeiro, se assim for, eliminamos esses algarismos e aplicamos a recursão para o número com menos dois algarismos.

Capicua (67876, 5) = capicua (787,3) = capicua (8) => 1

Capicua (63626634, 8) => 0

Recursão

Capicua (n, ndig) =
Se (n%10 == n/pow(10, ndig-1))
    capicua (n%pow(10,ndig-1)/10, ndig-2)

```
int capicua(int n, int ndig){
   if (n <10)
        return (1);
   else
        if ((n%10)!=n/(int)pow(10,ndig-1)) return (0);
        capicua((n%(int)pow(10,ndig-1)/10),ndig-2);
}
```

capicua(36863,5)=capicua(686,3)=capicua(8,1) => 1
capicua(6464746)=capicua(46474)=capicua(647)=> 0

Recursão

Exemplo 9- Faça um algoritmo que realize uma busca binária em um vetor ordenado de elementos.

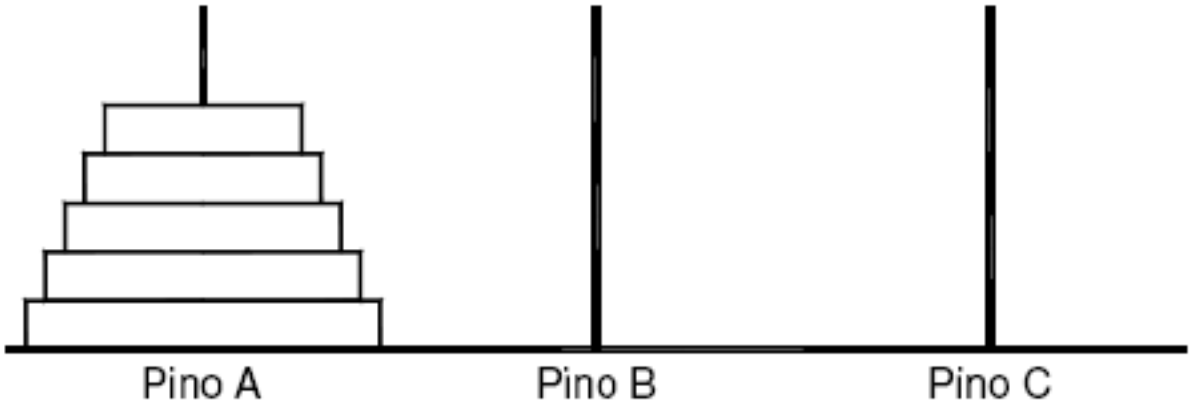
- Divide seu vetor em duas metades
- Três condições
  1. Se o item for igual ao item que está na metade do vetor, o item foi encontrado
  2. Se for menor, procure na primeira metade
  3. Se for maior procure na segunda metade

Busca Binária

```
Void Busca_Binária(int x, int inicio, int fim){
int meio
        meio=(inicio + fim)/ 2;
        if (fim < inicio) printf ("Elemento Não Encontrado");
        else
            if  (v[meio] ==x) printf ("Elemento está na posição %d",meio);
            else
                if (v[meio] < x){
                        inicio=meio +1;
                        Busca_Binaria (x, inicio, fim);
                }else{
                        fim=meio - 1;
                        Busca_Binaria (x, inicio, fim);
                }
}
```

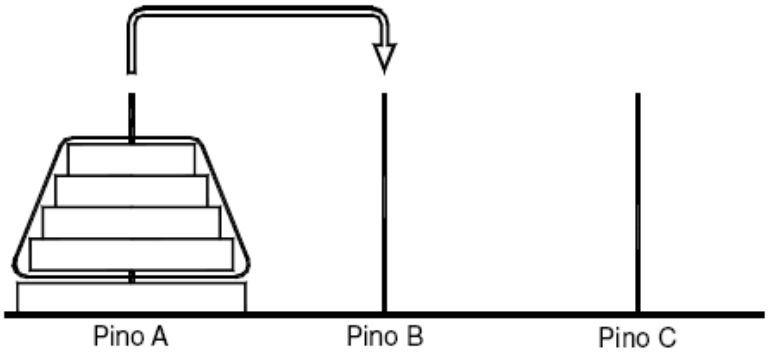
Recursão

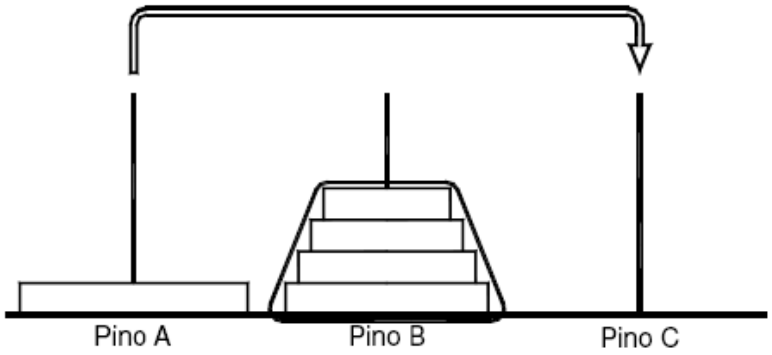
Exemplo 10-Problema da Torre de Hanói
O problema ou quebra-cabeça conhecido como torre de Hanói consiste em transferir, com o menor número de movimentos, a torre composta por *N* discos do pino **A** (origem) para o pino **C** (destino), utilizando o pino **B** como auxiliar. Somente um disco pode ser movimentado de cada vez e um disco não pode ser colocado sobre outro disco de menor diâmetro.

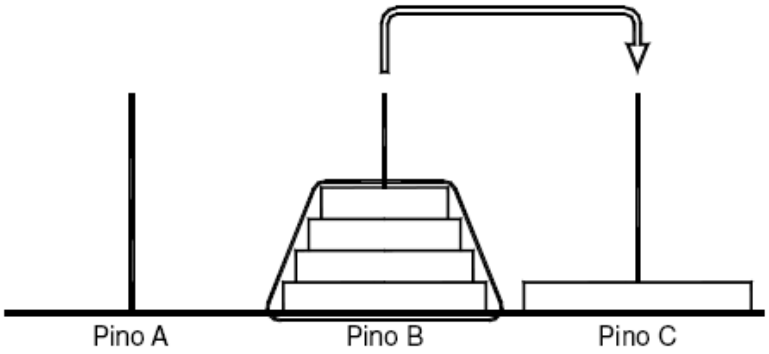


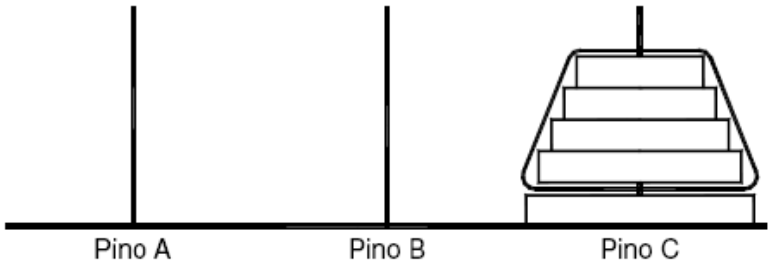
Recursão

Solução: Transferir a torre com *N*-1 discos de **A** para **B**, mover o maior disco de **A** para **C** e transferir a torre com *N*-1 de **B** para **C**. Embora não seja possível transferir a torre com *N*-1 de uma só vez, o problema torna-se mais simples: mover um disco e transferir duas torres com *N*-2 discos. Assim, cada transferência de torre implica em mover um disco e transferir de duas torres com um disco a menos e isso deve ser feito até que torre consista de um único disco.









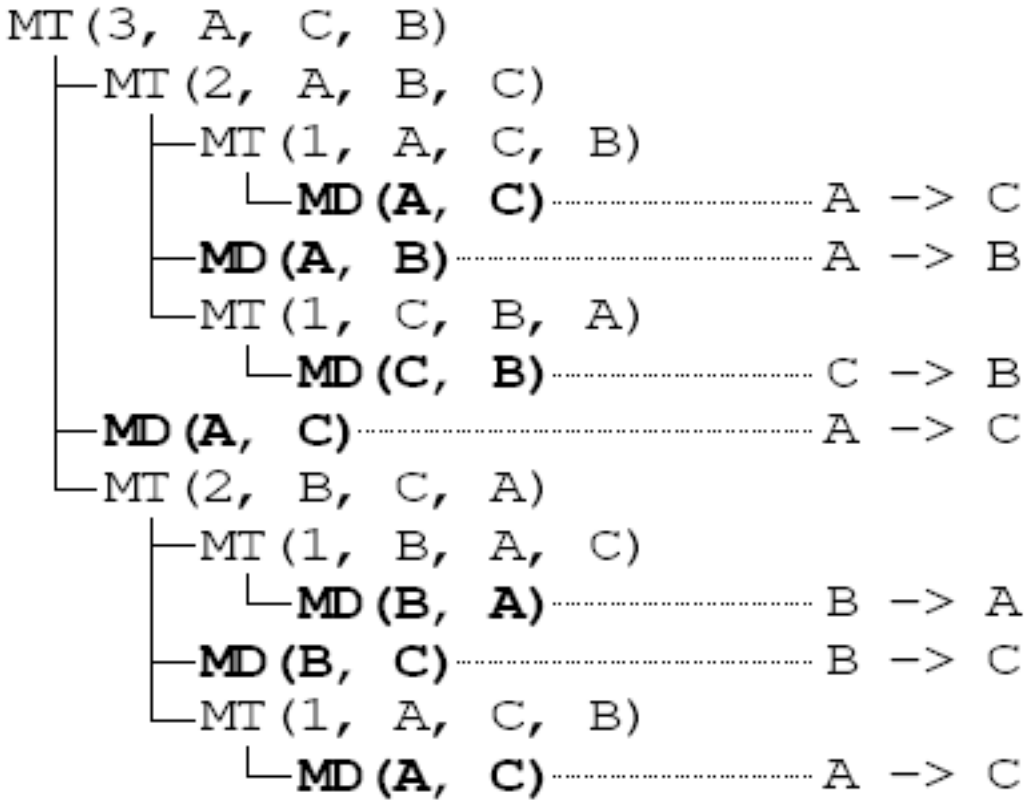
Recursão

```
vazio MoveTorre(inteiro:n, literal: Orig, Dest, Aux)
início
   se n = 1 então
      MoveDisco(Orig, Dest)
   senão
      MoveTorre(n - 1, Orig, Aux, Dest)
      MoveDisco(Orig, Dest)
      MoveTorre(n - 1, Aux, Dest, Orig)
   fim se
fim
vazio MoveDisco(literal:Orig, Dest)
início
Escreva(“Movimento: ”, Orig, “ -> ”, Dest)
fim
```

Recursão

Uma chamada a MoveTorre(3, ‘A’, ‘C’, ‘B’) teria a seguinte saída:

Movimento: A -> C
Movimento: A -> B
Movimento: C -> B
Movimento: A -> C
Movimento: B -> A
Movimento: B -> C
Movimento: A -> C



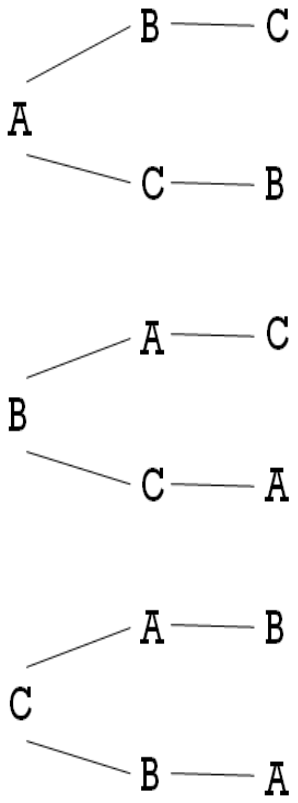
Recursão

Implemente o procedimento MoveTorre.

Recursão

Exemplo 11- Gerar todas as possíveis permutações de um conjunto de N símbolos. Por exemplo, existem 2 permutações no conjunto de 2 símbolos A e B: AB, BA.
E para um conjunto de 3 símbolos A, B e C, existem 6 permutações: ABC, ACB, BAC, BCA, CBA e CAB**.**

O conjunto de permutações de *N* símbolos pode ser gerada recursivamente a partir do conjunto de permutações de (*N* - 1) símbolos.
Vamos pegar cada símbolo e vamos prefixá-lo em cada permutação dos (N-1) símbolos restantes.
Seja N=3 e sejam os símbolos A,B,C
Vou prefixar A nas permutações de B e C: BC,CB
Obtendo ABC e ACB
Vou prefixar B nas permutações de A e C: AC,CA
Obtendo BAC e BCA
Vou prefixar C nas permutações de A e B: AB,BA
Obtendo CAB e CBA



Recursão

Note que o número total de permutações de n símbolos é dado por **P(n) = n!**

Ou seja:

Para 2 símbolos P(2)=2!=2 permutações

Para 3 símbolos P(3) = 3! = 3 x 2 x 1 = 6 permutações.

Para 4 símbolos temos: P(4) = 4! = 4 x 3 x 2 x 1 = 24 permutações.

O recurso básico para a elaboração do algoritmo consiste em montar uma **árvore de recursão** para uma “instância” de solução para o problema, por exemplo 3 símbolos.

Recursão

N=4, ABCD

Prefixar A nas permutações de BCD

Para achar as permutações de BCD

Prefixar B nas permutações de CD

BCD e BDC

Prefixar C nas permutações de BD

CBD e CDB

Prefixar D nas permutações de BC

DBC e DCB

Obtenho ABCD,ABDC,ACBD,ACDB,ADBC,ADCB

Recursão

Prefixar B nas permutações de ACD

Para achar as permutações de ACD

Prefixar A nas permutações de CD

ACD e ADC

Prefixar C nas permutações de AD

CAD e CDA

Prefixar D nas permutações de AC

DAC e DCA

Obtenho BACD,BADC,BCAD,BCDA,BDAC,BDCA

Recursão

Prefixar C nas permutações de BAD

Para achar as permutações de BAD

Prefixar B nas permutações de AD

BAD e BDA

Prefixar A nas permutações de BD

ABD e ADB

Prefixar D nas permutações de AB

DAB e DBA

Obtenho CBAD,CBDA,CABD,CADB,CDAB,CDBA

Recursão

Prefixar D nas permutações de BCA

Para achar as permutações de BCA

Prefixar B nas permutações de CA

BCA e BAC

Prefixar C nas permutações de BA

CBA e CAB

Prefixar A nas permutações de BC

ACB e ABC

Obtenho DBCA,DBAC, DCBA, DCAB, DACB, DABC

Recursão

- Na chamada recursiva :

  Para cada posição i de vetor de caracteres, entre k e o fim do vetor:

  1. Troque os caracteres das posições i e k.
  2. Gere todas as permutações do nível k+1, ou seja, todas as permutações de n-1 caracteres.
  3. Restaure o vetor original trocando as posições i e k.

Recursão

```
#include <stdio.h>
#include <string.h>
void permutate(char string[], int k, int n){
    int i;
    char temp;
    if(k == (n-1)){
        printf ("%s\n",string);
    }else{
        for(i = k;i < n; i++){
            temp = string[i];
            string[i] = string[k];
            string[k] = temp;
            permutate(string, (k+1), n);
            temp = string[k];
            string[k] = string[i];
            string[i] = temp;
         }
    }
}
int main (){
    char str[10];
    strcpy(str,"ABC");
    permutate(str,0,3);
    printf ("\n");
}
```

Pe(ABC,0)=

i= 0 i=1 i=2

Pe(ABC,1) Pe(BAC,1) Pe(CBA,1)

i=1 i=2 i=1 i=2 i=1 i=2

Pe(ABC,2) Pe(ACB,2) Pe(BAC,2) Pe(BCA) Pe(CBA,2) Pe(CAB)

ABC, ACB, BAC, BCA, CBA, CAB

Recursão

Exemplo 12-

O algoritmo Merge Sort tem como objetivo a ordenação de um vetor A de n elementos reais. A sua estratégia é baseada no paradigma dividir para conquistar.

1. Passo Dividir

Se um dado vetor *A* tem um ou nenhum elemento, simplesmente retorne pois ele já está ordenado. Caso contrário, divida $A[p .. r]$ em dois subvetores $A[p .. q]$ e $A[q + 1 .. r]$, cada um contendo a metade dos elementos de $A[p .. r]$. q é o índice do médio de $A[p .. r]$.

2. Passo Conquistar

Ordene recursivamente os dois subvetores $A[p .. q]$ e $A[q + 1 .. r]$.

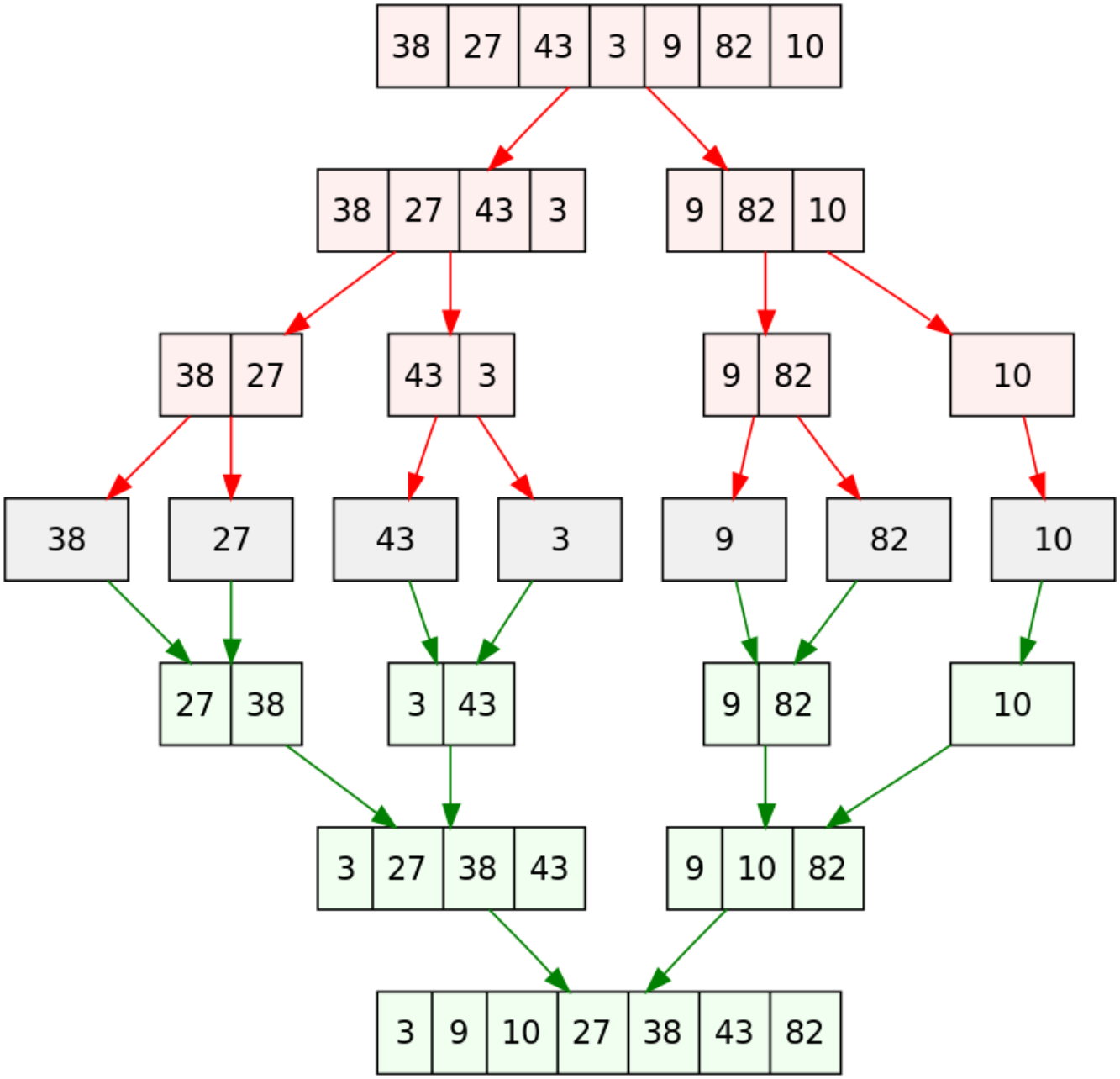
3. Passo combinar ou intercalar

Combine os elementos de volta em $A[p .. r]$ mesclando os subvetores ordenados $A[p .. q]$ e $A[q + 1 .. r]$ em uma sequência ordenada. Para realiza esta etapa vamos definir um procedimento Merge (A, p, q, r).

Recursão

```
void intercala( int p, int q, int r, int v[]) {
  int i, j, k, *w;
  w = malloc( (r-p) * sizeof (int));
  i = p; j = q; k = 0;
  while (i < q && j < r) {
        if (v[i] <= v[j]) w[k++] = v[i++];
        else w[k++] = v[j++]; }
  while (i < q) w[k++] = v[i++];
  while (j < r) w[k++] = v[j++];
  for (i = p; i < r; ++i) v[i] = w[i-p];
  free( w);
  }
 Void mergesort( int p, int r, int v[]) {
  if (p < r-1) {
    int q = (p + r)/2;
    mergesort( p, q, v);
    mergesort( q, r, v);
    intercala( p, q, r, v); }
    }
```

Recursão


38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82