Fundamentos de programação

aula 11: Programando recursivamente

1 Introdução

Lembre da esperteza que nós vimos na aula 06

- Para ordenar uma lista de números, nós podemos
  - ordenar a primeira metade
  - ordenar a segunda metade
  - e depois combinar as duas metades ordenadas

Agora, com a técnica das caixinhas, esse programa pode ser escrito da seguinte maneira

```
#include <stdlib.h>
#include <stdio.h>

#define  N  10

int L[N], A[N];

void combina_metades()
 {
   int  i=0, j=N/2, k=0;

   while ( i < N/2  ||  j < N )
    {
      if ( j == N  ||  ( i < N/2  &&  L[i] < L[j] ) )
       {
         A[k] = L[i];  i++;  k++; }
      else
         A[k] = L[j];  j++;  j++; }
    }
   for(k=0; k<N; k++)  L[k] = A[k];
 }

void ordena_metades (int in, int fin)     // a ordenação da bolha
 {                                        // mas poderia ser qq outra
    int i,j,aux;

    for (i=in; i<fin; i++)
      for (j=fin; j>in; j--)
          if (L[j] < L[j-1] )
          {
              aux = L[j];  L[j] = L[j-1];  L[j-1] = aux;
 }         }

int main()
 {
    ordena_metades(0,N/2-1);
    ordena_metades(N/2,N-1);
    combina_metades();
 }
```

Na aula 06, nós vimos que esse programa realiza a metade das comparações realizadas pelos outros programas que foram apresentados naquela aula.

E de fato, repetindo o teste de velocidade que nós fizemos na aula passada, nós verificamos que esse programa executa (aproximadamente) duas vezes mais rápido que a ordenação da bolha

```
ordenação da bolha:  2156845
 ordenação esperta:   982504
```

Legal.

Mas nós podemos ser mais espertos.

A observação chave é que, se a ordenação esperta é duas vezes mais rápida que a ordenação da bolha, então não faz sentido ordenar as duas metades usando o método da bolha.

Quer dizer, nós deveríamos utilizar o método esperto para fazer isso.

Mas, como é isso na prática?

Bom, aqui existem duas opções.

A primeira delas consiste em modificar o programa principal da seguinte maneira

```
int main()
 {
    // ETAPA 1: aplicar o método esperto na 1a metade
    ordena_metades(0,N/4-1);
    ordena_metades(N/4,N/2-1);
    combina_metades(0,N/2);

    // ETAPA 2: aplicar o método esperto na 2a metade
    ordena_metades(N/2,3*N/4-1);
    ordena_metades(3*N/4,N-1);
    combina_metades(N/2,N/2);

    // ETAPA 3: combinar as duas metades ordenadas
    combina_metades(0,N);
 }
```

(Note que nós precisamos adaptar a função `combina_metades()` para ela poder trabalhar apenas em uma parte da lista.)

De fato essa ideia funciona, e ela é duas vezes mais rápida que o método esperto

```
    ordenação da bolha:  2179460
     ordenação esperta:  1018383
ordenação mais esperta:   474463
```

1.1 Ordenação recursiva

Mas nós podemos ser ainda mais espertos.

Quer dizer, a segunda maneira de fazer as coisas é baseada em mais uma esperteza.

A observação chave dessa vez é que a versão original do programa principal já implementava a ordenação esperta

```
int main()
 {
    ordena_metades(0,N/2-1);
    ordena_metades(N/2,N-1);
    combina_metades();
 }
```

Daí que, se nós queremos ordenar as metades de maneira esperta, basta fazer uma chamada à função `main`

```
void ordena_metades (int in, int fin)     // versão esperta
 {
     . . .
    main();
 }
```

Bom, fazer uma chamada à função `main` pode parecer uma coisa meio estranha ...

Mas a linguagem C nos permite fazer isso.

O problema é que a função `main`() não é realmente uma função como outra qualquer, e isso acaba dando alguns problemas.

Mas isso não é nada que alguns poucos truques sujos de programação não possam resolver.

E daí, quando a gente bota a coisa para funcionar, o resultado que a gente obtém é radical:

```
          ordenação da bolha:  2074409
           ordenação esperta:  1103090
    ordenação mais esperta:     460097
       ordenação recursiva:       4069
```

2 Programação recursiva

Mas, o que aconteceu?

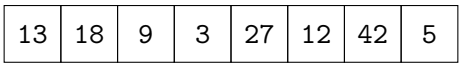
Bom, o que aconteceu foi que nós acabamos encontrando um dos algoritmos de ordenação mais eficientes que existem.

E no final das contas, a sua ideia é bem simples

- Para ordenar uma lista de números, nós podemos
  - ordenar a primeira metade
  - ordenar a segunda metade
  - e depois combinar as duas metades ordenadas

  sendo que a ordenação das duas metades também é feita por esse método.

Para ver como a coisa funciona na prática, considere a seguinte lista de tamanho 8



Para ordenar essa lista, nós vamos ordenar as suas duas metades

E para ordenar essas 2 listas, nós vamos ordenar as suas metades

E para ordenar essas 4 listas, nós vamos ordenar as suas metades

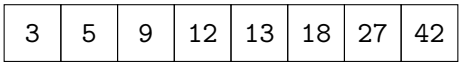
Mas uma lista de tamanho 1 já está ordenada.

Então agora nós começamos as etapas de combinação.

Combinando as listas ordenadas de tamanho 1, nós obtemos

Combinando as listas ordenadas de tamanho 2, nós obtemos

Finalmente, combinando as listas ordenadas de tamanho 4, nós obtemos

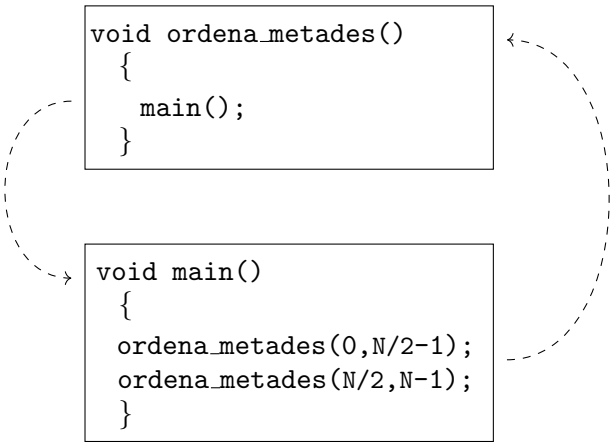


Pronto.

Só o que sobrou foram as operações de combinação de listas ordenadas.

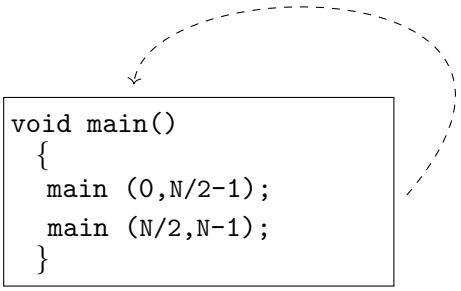
E isso é extremamente rápido.

A ideia que nós tivemos para implementar esse método foi a seguinte



Mas, prestando atenção, a gente vê que a função `ordena_metades`() não está fazendo mais nada.

Daí que, a gente pode tirar essa função da jogada, para obter algo assim



Isto é, uma função que faz uma chamada a ela mesma.

O que a linguagem C também permite fazer.

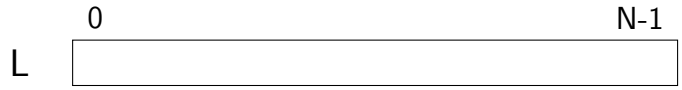
Na prática, é conveniente fazer isso por meio de uma outra função que não a `main`().

E a coisa fica assim

```
void combina_metades (int i, int f)
 {
     ( ... omitido ... )
 }

void ordenacao_recursiva (int i, int f)
 {
   int meio;

   if ( i == f )  return;

   meio = (i+f)/2;
   ordenacao_recursiva (i,meio);
   ordenacao_recursiva (meio+1,f);
   combina_metades(i,f);
 }

int main()
 {
    ordenacao_recursiva(0,N-1);
 }
```

3 Outros exemplos

Imagine que nós queremos escrever uma função que imprime a lista



Pensando de maneira recursiva, a coisa seria feita assim

- Primeiro nós imprimimos o 1^o elemento
  e depois nós imprimimos o resto da lista (fazendo uma chamada recursiva)

Abaixo nós temos o programa em C que implementa essa ideia

```
#include <stdlib.h>
#include <stdio.h>

#define  N  10

int L = { 13, 18, 9, 3, 27, 12, 42, 5, 55, 2 };

void imprime_rec (int i)
 {
    if ( i == N )  { printf("\n"); return;  }

    printf ("%d ", L[i]);
    i++;
    imprime_rec(i);
 }

int main()
 {
    imprime_rec(0);
 }
```

Legal, não é?

Quer dizer, nesse exemplo a chamada recursiva está fazendo o papel de um laço de repetição.

Vejamos outros exemplos.

Exemplos

a. **Busca em uma lista desordenada**

Imagine agora que nós queremos saber se o número `x` está na lista `L` ou não.

Pensando recursivamente, a coisa seria feita assim

- Verifique se `x` é o 1^o elemento
  se não for, procure no resto da lista (fazendo uma chamada recursiva)

Abaixo nós temos o programa em C que implementa essa ideia.

```
#include <stdlib.h>
#include <stdio.h>

#define  N  10

int L = { 13, 18, 9, 3, 27, 12, 42, 5, 55, 2 };

void busca_rec (int x, int i)
 {
    if ( i == N )  {  printf ("O número x não está na lista\n");
                      return;  }

    if ( x == L[i] )
     {
       printf ("O número x foi encontrado na posição %d\n", i);
     }
    else
     {
       i++;
       busca_rec(x,i);
 }   }

int main()
 {
    int x = 42;

    busca_rec (x,0);
 }
```

◇

b. **Busca binária**

Imagine agora que a lista está ordenada.

Daí, a ideia é começar a busca pelo meio da lista.

E pensando recursivamente, a coisa ficaria assim

- Verifique se x é o elemento do meio

  se for, então acabou;

  se x é menor que o elemento do meio, então procure na metade esquerda;

  se x é maior que o elemento do meio, então procure na metade direita

Esse procedimento é conhecido como a *busca binária.*

E abaixo nós temos a função em C que implementa esse procedimento

```
void busca_binaria (int i, int f)
 {
    int k;

    if ( i > f )  {  printf("O número x não está na lista\n");
                     return;  }
    k = (i+f)/2;
    if ( x == L[k] )  {  printf("O número x foi encontrado na posição %d\n", i);
                         return;  }
    if ( x < L[k] )  busca_binaria(i,k-1);
    if ( x > L[k] )  busca_binaria(k+1,f);
 }
```

◇