MC102 – Aula23
Recursão III - QuickSort e MergeSort

Instituto de Computação – Unicamp

28 de Maio de 2014

Introdução

Vamos usar a técnica de recursão para resolver o problema de ordenação.

- Problema:
  - Temos um vetor v de inteiros de tamanho n.
  - Devemos deixar v ordenado em ordem crescente de valores.
- Veremos algoritmos baseados na técnica **dividir-e-conquistar** que usam recursão.

Introdução

Vamos usar a técnica de recursão para resolver o problema de ordenação.

- Problema:
  - Temos um vetor v de inteiros de tamanho n.
  - Devemos deixar v ordenado em ordem crescente de valores.
- Veremos algoritmos baseados na técnica **dividir-e-conquistar** que usam recursão.

Dividir e Conquistar

- Temos que resolver um problema P de tamanho n.
- **Dividir:** Quebramos P em sub-problemas menores.
- Resolvemos os sub-problemas de forma recursiva.
- **Conquistar:** Unimos as soluções dos sub-problemas para obter solução do problema maior P.

Dividir e Conquistar

- Temos que resolver um problema P de tamanho n.
- **Dividir:** Quebramos P em sub-problemas menores.
- Resolvemos os sub-problemas de forma recursiva.
- **Conquistar:** Unimos as soluções dos sub-problemas para obter solução do problema maior P.

Dividir e Conquistar

- Temos que resolver um problema P de tamanho n.
- **Dividir:** Quebramos P em sub-problemas menores.
- Resolvemos os sub-problemas de forma recursiva.
- **Conquistar:** Unimos as soluções dos sub-problemas para obter solução do problema maior P.

Dividir e Conquistar

- Temos que resolver um problema P de tamanho n.
- **Dividir:** Quebramos P em sub-problemas menores.
- Resolvemos os sub-problemas de forma recursiva.
- **Conquistar:** Unimos as soluções dos sub-problemas para obter solução do problema maior P.

Quick-Sort

- Vamos supor que devemos ordenar um vetor de uma posição *ini* até *fim*.
- **Dividir:**
  - Escolha em elemento especial do vetor chamado *pivô*.
  - Particione o vetor em uma posição *pos* tal que todos elementos de *ini* até $pos - 1$ são menores ou iguais do que o *pivô*, e todos elementos de *pos* até *fim* são maiores ou iguais ao *pivô*.
- Resolvemos o problema de ordenação de forma recursiva para estes dois sub-vetores (um de *ini* até $pos - 1$ e o outro de *pos* até *fim*).
- **Conquistar:** Nada a fazer já que o vetor estará ordenado devido a como foi feito a fase de divisão.

Quick-Sort

- Vamos supor que devemos ordenar um vetor de uma posição *ini* até *fim*.
- **Dividir:**
  - Escolha em elemento especial do vetor chamado *pivô*.
  - Particione o vetor em uma posição *pos* tal que todos elementos de *ini* até $pos - 1$ são menores ou iguais do que o *pivô*, e todos elementos de *pos* até *fim* são maiores ou iguais ao *pivô*.
- Resolvemos o problema de ordenação de forma recursiva para estes dois sub-vetores (um de *ini* até $pos - 1$ e o outro de *pos* até *fim*).
- **Conquistar:** Nada a fazer já que o vetor estará ordenado devido a como foi feito a fase de divisão.

Quick-Sort

- Vamos supor que devemos ordenar um vetor de uma posição *ini* até *fim*.
- **Dividir:**
  - Escolha em elemento especial do vetor chamado *pivô*.
  - Particione o vetor em uma posição *pos* tal que todos elementos de *ini* até $pos - 1$ são menores ou iguais do que o *pivô*, e todos elementos de *pos* até *fim* são maiores ou iguais ao *pivô*.
- Resolvemos o problema de ordenação de forma recursiva para estes dois sub-vetores (um de *ini* até $pos - 1$ e o outro de *pos* até *fim*).
- **Conquistar:** Nada a fazer já que o vetor estará ordenado devido a como foi feito a fase de divisão.

Quick-Sort

- Vamos supor que devemos ordenar um vetor de uma posição *ini* até *fim*.
- **Dividir:**
  - Escolha em elemento especial do vetor chamado *pivô*.
  - Particione o vetor em uma posição *pos* tal que todos elementos de *ini* até $pos - 1$ são menores ou iguais do que o *pivô*, e todos elementos de *pos* até *fim* são maiores ou iguais ao *pivô*.
- Resolvemos o problema de ordenação de forma recursiva para estes dois sub-vetores (um de *ini* até $pos - 1$ e o outro de *pos* até *fim*).
- **Conquistar:** Nada a fazer já que o vetor estará ordenado devido a como foi feito a fase de divisão.

Quick-Sort

- Vamos supor que devemos ordenar um vetor de uma posição *ini* até *fim*.
- **Dividir:**
  - Escolha em elemento especial do vetor chamado *pivô*.
  - Particione o vetor em uma posição *pos* tal que todos elementos de *ini* até $pos - 1$ são menores ou iguais do que o *pivô*, e todos elementos de *pos* até *fim* são maiores ou iguais ao *pivô*.
- Resolvemos o problema de ordenação de forma recursiva para estes dois sub-vetores (um de *ini* até $pos - 1$ e o outro de *pos* até *fim*).
- **Conquistar:** Nada a fazer já que o vetor estará ordenado devido a como foi feito a fase de divisão.

Quick-Sort: Particionamento

Dado um valor p como pivô, como fazer o particionamento?

- Podemos "varrer" o vetor do início para o fim até encontrarmos um elemento maior que o pivô.
- Varremos o vetor do fim para o início até encontrarmos um elemento menor ou igual ao pivô.
- Trocamos então estes elementos de posição e continuamos com o processo até termos verificado todas as posições do vetor.

Quick-Sort: Particionamento

Dado um valor p como pivô, como fazer o particionamento?

- Podemos "varrer"o vetor do início para o fim até encontrarmos um elemento maior que o pivô.
- Varremos o vetor do fim para o início até encontrarmos um elemento menor ou igual ao pivô.
- Trocamos então estes elementos de posição e continuamos com o processo até termos verificado todas as posições do vetor.

Quick-Sort: Particionamento

Dado um valor p como pivô, como fazer o particionamento?

- Podemos "varrer"o vetor do início para o fim até encontrarmos um elemento maior que o pivô.
- Varremos o vetor do fim para o início até encontrarmos um elemento menor ou igual ao pivô.
- Trocamos então estes elementos de posição e continuamos com o processo até termos verificado todas as posições do vetor.

Quick-Sort: Particionamento

Dado um valor p como pivô, como fazer o particionamento?

- Podemos "varrer"o vetor do início para o fim até encontrarmos um elemento maior que o pivô.
- Varremos o vetor do fim para o início até encontrarmos um elemento menor ou igual ao pivô.
- Trocamos então estes elementos de posição e continuamos com o processo até termos verificado todas as posições do vetor.

Quick-Sort: Particionamento

A função retorna a posição de partição. Ela considera sempre o último elemento como o pivô.

```
int particiona(int v[], int ini, int fim){
  int pivo = v[fim], aux;

  while(ini<fim){
    while( (ini < fim) && (v[ini] <= pivo) ) //para quando encontrar elemento
      ini++;                                 //maior que o pivô

    while( (ini < fim) && (v[fim] > pivo) ) //para quando encontrar elemento
      fim--;                                //menor ou igual ao pivô

    aux = v[ini]; //troca estes elementos de posição
    v[ini] = v[fim];
    v[fim] = aux;
  }

  //O laço para quando ini==fim, ou seja checamos o vetor inteiro
  return ini;
}
```

Quick-Sort: Particionamento

A função retorna a posição de partição. Ela considera sempre o último elemento como o pivô.

```
int particiona(int v[], int ini, int fim){
  int pivo = v[fim], aux;

  while(ini<fim){
    while( (ini < fim) && (v[ini] <= pivo) ) //para quando encontrar elemento
      ini++;                                 //maior que o pivô

    while( (ini < fim) && (v[fim] > pivo) ) //para quando encontrar elemento
      fim--;                                //menor ou igual ao pivô

    aux = v[ini]; //troca estes elementos de posição
    v[ini] = v[fim];
    v[fim] = aux;
  }

  //O laço para quando ini==fim, ou seja checamos o vetor inteiro
  return ini;
}
```

Quick-Sort: Particionamento

A função retorna a posição de partição. Ela considera sempre o último elemento como o pivô.

```
int particiona(int v[], int ini, int fim){
  int pivo = v[fim], aux;

  while(ini<fim){
    while( (ini < fim) && (v[ini] <= pivo) ) //para quando encontrar elemento
      ini++;                                 //maior que o pivô

    while( (ini < fim) && (v[fim] > pivo) ) //para quando encontrar elemento
      fim--;                                //menor ou igual ao pivô

    aux = v[ini]; //troca estes elementos de posição
    v[ini] = v[fim];
    v[fim] = aux;
  }

  //O laço para quando ini==fim, ou seja checamos o vetor inteiro
  return ini;
}
```

Quick-Sort: Particionamento

Exemplo: (1,9,3,7,6,2,3,8,5) e pivô=5.

- (1,9,3,7,6,2,3,8,5) → (1,5,3,7,6,2,3,8,9)
- (1,5,3,7,6,2,3,8,9) → (1,5,3,3,6,2,7,8,9)
- (1,5,3,3,6,2,7,8,9) → (1,5,3,3,2,6,7,8,9)
- (1,5,3,3,2,6,7,8,9) → Retorna posição 5.

Quick-Sort: Particionamento

Exemplo: (1,9,3,7,6,2,3,8,5) e pivô=5.

- (1,9,3,7,6,2,3,8,5) $\rightarrow$ (1,5,3,7,6,2,3,8,9)
- (1,5,3,7,6,2,3,8,9) $\rightarrow$ (1,5,3,3,6,2,7,8,9)
- (1,5,3,3,6,2,7,8,9) $\rightarrow$ (1,5,3,3,2,6,7,8,9)
- (1,5,3,3,2,6,7,8,9) $\rightarrow$ Retorna posição 5.

Quick-Sort: Particionamento

Exemplo: (1,9,3,7,6,2,3,8,5) e pivô=5.

- (1,9,3,7,6,2,3,8,5) $\rightarrow$ (1,5,3,7,6,2,3,8,9)
- (1,5,3,7,6,2,3,8,9) $\rightarrow$ (1,5,3,3,6,2,7,8,9)
- (1,5,3,3,6,2,7,8,9) $\rightarrow$ (1,5,3,3,2,6,7,8,9)
- (1,5,3,3,2,6,7,8,9) $\rightarrow$ Retorna posição 5.

Quick-Sort: Particionamento

Exemplo: (1,9,3,7,6,2,3,8,5) e pivô=5.

- (1,9,3,7,6,2,3,8,5) $\rightarrow$ (1,5,3,7,6,2,3,8,9)
- (1,5,3,7,6,2,3,8,9) $\rightarrow$ (1,5,3,3,6,2,7,8,9)
- (1,5,3,3,6,2,7,8,9) $\rightarrow$ (1,5,3,3,2,6,7,8,9)
- (1,5,3,3,2,6,7,8,9) $\rightarrow$ Retorna posição 5.

Quick-Sort: Particionamento

Exemplo: (1,9,3,7,6,2,3,8,5) e pivô=5.

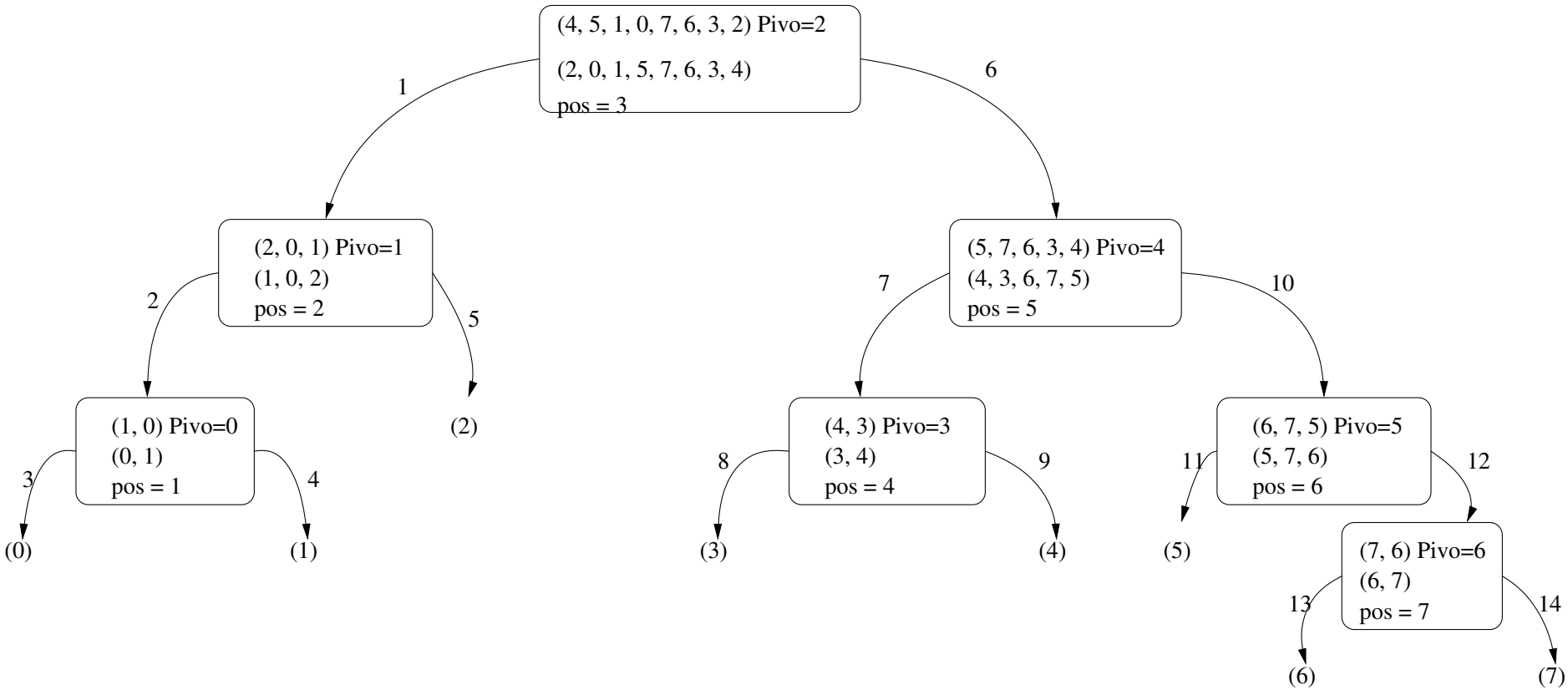
- (1,9,3,7,6,2,3,8,5) $\rightarrow$ (1,5,3,7,6,2,3,8,9)
- (1,5,3,7,6,2,3,8,9) $\rightarrow$ (1,5,3,3,6,2,7,8,9)
- (1,5,3,3,6,2,7,8,9) $\rightarrow$ (1,5,3,3,2,6,7,8,9)
- (1,5,3,3,2,6,7,8,9) $\rightarrow$ Retorna posição 5.

Quick-Sort

```
void quickSort(int v[], int ini, int fim){
  if(ini < fim){ //só faz ordenação se tiver pelo
                 //menos 2 elementos
    int pos = particiona(v, ini, fim);
    quickSort(v,ini, pos-1);
    quickSort(v,pos, fim);
  }
}
```

Quick-Sort

Abaixo temos um exemplo da árvore de recursão com ordem das chamadas recursivas.



Quick-Sort

- Se o Quick-Sort particionar o vetor de tal forma que cada partição tenha mais ou menos o mesmo tamanho ele é muito eficiente.
- Porém se a partição for muito desigual ($n-1$ de um lado e 1 de outro) ele é ineficiente.
- Quando um vetor já está ordenado ou quase-ordenado, ocorre este caso ruim. Por que?

Quick-Sort

- Se o Quick-Sort particionar o vetor de tal forma que cada partição tenha mais ou menos o mesmo tamanho ele é muito eficiente.
- Porém se a partição for muito desigual ($n-1$ de um lado e 1 de outro) ele é ineficiente.
- Quando um vetor já está ordenado ou quase-ordenado, ocorre este caso ruim. Por que?

Quick-Sort

- Se o Quick-Sort particionar o vetor de tal forma que cada partição tenha mais ou menos o mesmo tamanho ele é muito eficiente.
- Porém se a partição for muito desigual ($n-1$ de um lado e 1 de outro) ele é ineficiente.
- Quando um vetor já está ordenado ou quase-ordenado, ocorre este caso ruim. Por que?

Quick-Sort: Tratando o pior caso

- Podemos implementar o Quick-Sort de tal forma a diminuirmos a chance de ocorrência do pior caso.
- Ao invés de escolhermos o pivô como um elemento de uma posição fixa, podemos escolher como pivô o elemento de uma posição aleatória.
- Podemos usar a função *rand* em *stdlib.h* que retorna um número de forma aleatória entre 0 e *RAND_MAX*.

Quick-Sort: Tratando o pior caso

- Podemos implementar o Quick-Sort de tal forma a diminuirmos a chance de ocorrência do pior caso.
- Ao invés de escolhermos o pivô como um elemento de uma posição fixa, podemos escolher como pivô o elemento de uma posição aleatória.
- Podemos usar a função *rand* em *stdlib.h* que retorna um número de forma aleatória entre 0 e *RAND_MAX*.

Quick-Sort: Tratando o pior caso

- Podemos implementar o Quick-Sort de tal forma a diminuirmos a chance de ocorrência do pior caso.
- Ao invés de escolhermos o pivô como um elemento de uma posição fixa, podemos escolher como pivô o elemento de uma posição aleatória.
- Podemos usar a função *rand* em *stdlib.h* que retorna um número de forma aleatória entre 0 e *RAND_MAX*.

Random-Quick-Sort

- A única diferença é que escolhemos um elemento aleatório.
- Tal elemento é trocado com o que está no fim (será o pivô).

```
void randomQuickSort(int v[],int ini, int fim){
  int j, aux;
  j = rand()%(fim-ini+1);
  aux = v[fim];
  v[fim] = v[ini+j];
  v[ini+j] = aux;

  if(ini <fim){
    int pos = particiona(v, ini, fim);
    randomQuickSort(v, ini, pos-1);
    randomQuickSort(v, pos, fim);
  }
}
```

Random-Quick-Sort

- A única diferença é que escolhemos um elemento aleatório.
- Tal elemento é trocado com o que está no fim (será o pivô).

```
void randomQuickSort(int v[],int ini, int fim){
  int j, aux;
  j = rand()%(fim-ini+1);
  aux = v[fim];
  v[fim] = v[ini+j];
  v[ini+j] = aux;

  if(ini <fim){
    int pos = particiona(v, ini, fim);
    randomQuickSort(v, ini, pos-1);
    randomQuickSort(v, pos, fim);
  }
}
```

Random-Quick-Sort

- A única diferença é que escolhemos um elemento aleatório.
- Tal elemento é trocado com o que está no fim (será o pivô).

```
void randomQuickSort(int v[],int ini, int fim){
  int j, aux;
  j = rand()%(fim-ini+1);
  aux = v[fim];
  v[fim] = v[ini+j];
  v[ini+j] = aux;

  if(ini <fim){
    int pos = particiona(v, ini, fim);
    randomQuickSort(v, ini, pos-1);
    randomQuickSort(v, pos, fim);
  }
}
```

Random-Quick-Sort

- A chance de ocorrer um caso ruim para o Random-Quick-Sort é desprezível.

Quick-Sort

1. Aplique o algoritmo de particionamento sobre o vetor $(13, 19, 9, 5, 12, 21, 7, 4, 11, 2, 6, 6)$ com pivô igual a 6.
2. Qual o valor retornado pelo algoritmo de particionamento se todos os elementos do vetor tiverem valores iguais?
3. Faça uma execução passo-a-passo do Quick-Sort com o vetor $(4, 3, 6, 7, 9, 10, 5, 8)$.
4. Modifique o algoritmo QuickSort para ordenar vetores em ordem decrescente.

Merge-Sort: Ordenação por intercalação

- O Merge-Sort é um algoritmo baseado na técnica dividir-e-conquistar.
- Neste caso temos que ordenar um vetor de tamanho n.
  - **Dividir:** Dividimos o vetor de tamanho n em dois sub-vetores de tamanho aproximadamente iguais (um de tamanho $\lceil n/2 \rceil$ e outro de tamanho $\lfloor n/2 \rfloor$).
  - Resolvemos o problema de ordenação de forma recursiva para estes dois sub-vetores.
  - **Conquistar:** Com os dois sub-vetores ordenados, construímos um vetor de tamanho n ordenado.

Merge-Sort: Ordenação por intercalação

- O Merge-Sort é um algoritmo baseado na técnica dividir-e-conquistar.
- Neste caso temos que ordenar um vetor de tamanho n.
  - **Dividir:** Dividimos o vetor de tamanho n em dois sub-vetores de tamanho aproximadamente iguais (um de tamanho $\lceil n/2 \rceil$ e outro de tamanho $\lfloor n/2 \rfloor$).
  - Resolvemos o problema de ordenação de forma recursiva para estes dois sub-vetores.
  - **Conquistar:** Com os dois sub-vetores ordenados, construímos um vetor de tamanho n ordenado.

Merge-Sort: Ordenação por intercalação

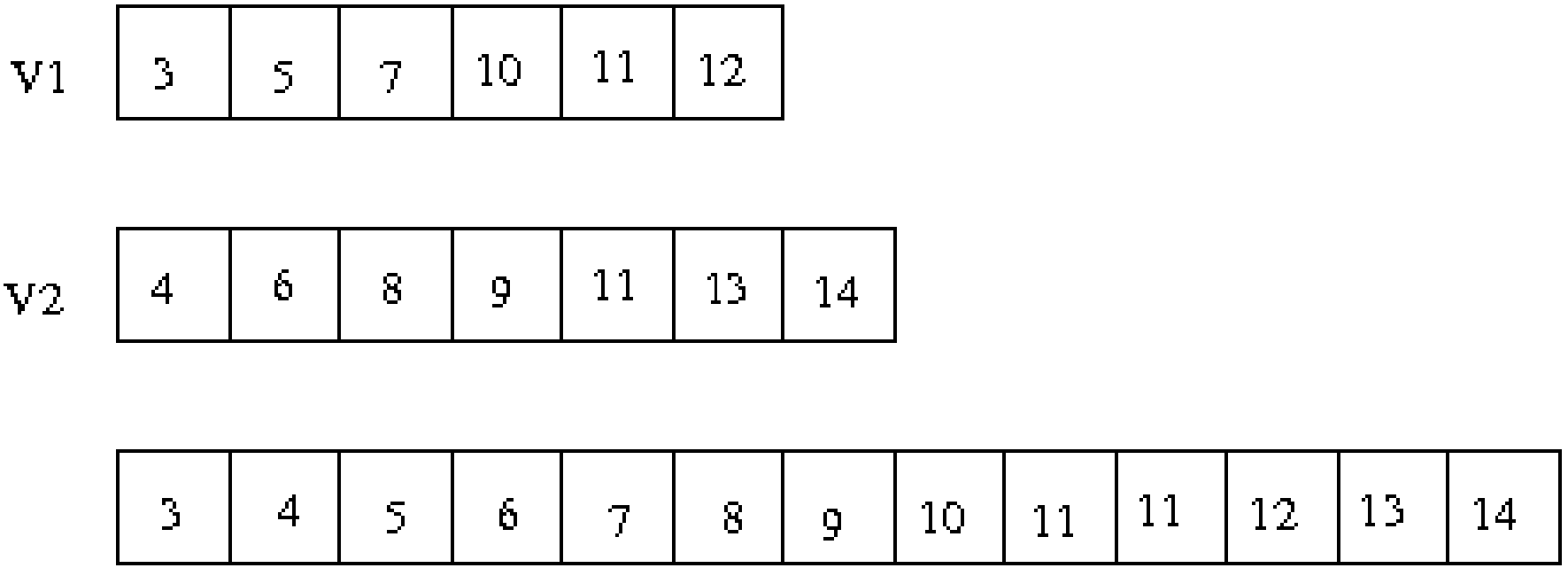
- O Merge-Sort é um algoritmo baseado na técnica dividir-e-conquistar.
- Neste caso temos que ordenar um vetor de tamanho n.
  - **Dividir:** Dividimos o vetor de tamanho n em dois sub-vetores de tamanho aproximadamente iguais (um de tamanho $\lceil n/2 \rceil$ e outro de tamanho $\lfloor n/2 \rfloor$).
  - Resolvemos o problema de ordenação de forma recursiva para estes dois sub-vetores.
  - **Conquistar:** Com os dois sub-vetores ordenados, construímos um vetor de tamanho n ordenado.

Merge-Sort: Ordenação por intercalação

- O Merge-Sort é um algoritmo baseado na técnica dividir-e-conquistar.
- Neste caso temos que ordenar um vetor de tamanho n.
  - **Dividir:** Dividimos o vetor de tamanho n em dois sub-vetores de tamanho aproximadamente iguais (um de tamanho $\lceil n/2 \rceil$ e outro de tamanho $\lfloor n/2 \rfloor$).
  - Resolvemos o problema de ordenação de forma recursiva para estes dois sub-vetores.
  - **Conquistar:** Com os dois sub-vetores ordenados, construímos um vetor de tamanho n ordenado.

Merge-Sort: Ordenação por intercalação

Conquistar: Dados dois vetores v_1 e v_2 ordenados, como obter um outro vetor ordenado contendo os elementos de v_1 e v_2?



Merge (Fusão)

- A idéia é executar um laço em que cada iteração testamos quem é o menor elemento dentre $v_1[i]$ e $v_2[j]$ e copiamos este elemento para o novo vetor.
- Durante a execução deste laço podemos chegar em um ponto em que todos elementos de um dos vetores (v_1 ou v_2) tenham sido inteiramente copiados. Neste caso terminamos o laço e copiamos os elementos restantes do outro vetor.

Merge (Fusão)

- A idéia é executar um laço em que cada iteração testamos quem é o menor elemento dentre $v_1[i]$ e $v_2[j]$ e copiamos este elemento para o novo vetor.
- Durante a execução deste laço podemos chegar em um ponto em que todos elementos de um dos vetores (v_1 ou v_2) tenham sido inteiramente copiados. Neste caso terminamos o laço e copiamos os elementos restantes do outro vetor.

Merge (Fusão)

Retorna um vetor que é a fusão dos vetores passados por parâmetro:

```
int * merge(int a[], int ta,  int b[], int tb){
  int *c = malloc(sizeof(int)*(ta+tb) );
  int i=0,j=0,k=0; //índice de a, b, e c resp.

  while(i< ta && j< tb){
    if(a[i] <= b[j])
      c[k++] = a[i++];
    else
      c[k++] = b[j++];
  }

  while(i<ta) //copia resto de a
    c[k++] = a[i++];
  while(j<tb) //copia resto de b
    c[k++] = b[j++];
  return c;
}
```

Merge (Fusão)

Retorna um vetor que é a fusão dos vetores passados por parâmetro:

```
int * merge(int a[], int ta,  int b[], int tb){
  int *c = malloc(sizeof(int)*(ta+tb) );
  int i=0,j=0,k=0; //índice de a, b, e c resp.

  while(i< ta && j< tb){
    if(a[i] <= b[j])
      c[k++] = a[i++];
    else
      c[k++] = b[j++];
  }

  while(i<ta) //copia resto de a
    c[k++] = a[i++];
  while(j<tb) //copia resto de b
    c[k++] = b[j++];
  return c;
}
```

Merge (Fusão)

Retorna um vetor que é a fusão dos vetores passados por parâmetro:

```
int * merge(int a[], int ta,  int b[], int tb){
  int *c = malloc(sizeof(int)*(ta+tb) );
  int i=0,j=0,k=0; //índice de a, b, e c resp.

  while(i< ta && j< tb){
    if(a[i] <= b[j])
      c[k++] = a[i++];
    else
      c[k++] = b[j++];
  }

  while(i<ta) //copia resto de a
    c[k++] = a[i++];
  while(j<tb) //copia resto de b
    c[k++] = b[j++];
  return c;
}
```

Merge (Fusão)

- A função descrita recebe dois vetores ordenados e devolve um terceiro contendo todos os elementos.
- Porém no merge-sort faremos a intercalação de pedaços de um mesmo vetor.
- Teremos posições *ini*, *meio*, *fim* de um vetor e devemos fazer a intercalação dos dois sub-vetores (ini até meio e meio+1 até fim).
  - Para isso a função utiliza um vetor auxiliar, que receberá o resultado da intercalação, e que no final é copiado para o vetor a ser ordenado.

Merge (Fusão)

- A função descrita recebe dois vetores ordenados e devolve um terceiro contendo todos os elementos.
- Porém no merge-sort faremos a intercalação de pedaços de um mesmo vetor.
- Teremos posições *ini*, *meio*, *fim* de um vetor e devemos fazer a intercalação dos dois sub-vetores (ini até meio e meio+1 até fim).
  - Para isso a função utiliza um vetor auxiliar, que receberá o resultado da intercalação, e que no final é copiado para o vetor a ser ordenado.

Merge (Fusão)

- A função descrita recebe dois vetores ordenados e devolve um terceiro contendo todos os elementos.
- Porém no merge-sort faremos a intercalação de pedaços de um mesmo vetor.
- Teremos posições *ini*, *meio*, *fim* de um vetor e devemos fazer a intercalação dos dois sub-vetores (ini até meio e meio+1 até fim).
  - Para isso a função utiliza um vetor auxiliar, que receberá o resultado da intercalação, e que no final é copiado para o vetor a ser ordenado.

Merge (Fusão)

Faz intercalação de pedaços de v. No fim v estará ordenado entre as posições *ini* e *fim*:

```
void merge(int v[], int ini,  int meio, int fim, int aux[]){
  int i=ini,j=meio+1,k=0; //indices da metade inf, sup e aux respc.

  while(i<=meio && j<=fim){
    if(v[i] <= v[j])
      aux[k++] = v[i++];
    else
      aux[k++] = v[j++];
  }

  while(i<=meio)
    aux[k++] = v[i++];
  while(j<=fim)
    aux[k++] = v[j++];
  for(i=ini, k=0 ; i<= fim; i++, k++) //copia aux para v
      v[i]=aux[k];
}
```

Merge (Fusão)

Faz intercalação de pedaços de v. No fim v estará ordenado entre as posições *ini* e *fim*:

```
void merge(int v[], int ini,  int meio, int fim, int aux[]){
  int i=ini,j=meio+1,k=0; //indices da metade inf, sup e aux respc.

  while(i<=meio && j<=fim){
    if(v[i] <= v[j])
      aux[k++] = v[i++];
    else
      aux[k++] = v[j++];
  }

  while(i<=meio)
    aux[k++] = v[i++];
  while(j<=fim)
    aux[k++] = v[j++];
  for(i=ini, k=0 ; i<= fim; i++, k++) //copia aux para v
      v[i]=aux[k];
}
```

Merge (Fusão)

Faz intercalação de pedaços de v. No fim v estará ordenado entre as posições *ini* e *fim*:

```
void merge(int v[], int ini,  int meio, int fim, int aux[]){
  int i=ini,j=meio+1,k=0; //indices da metade inf, sup e aux respc.

  while(i<=meio && j<=fim){
    if(v[i] <= v[j])
      aux[k++] = v[i++];
    else
      aux[k++] = v[j++];
  }

  while(i<=meio)
    aux[k++] = v[i++];
  while(j<=fim)
    aux[k++] = v[j++];
  for(i=ini, k=0 ; i<= fim; i++, k++) //copia aux para v
      v[i]=aux[k];
}
```

Merge-Sort

- O merge-sort resolve de forma recursiva dois sub-problemas, cada um contendo uma metade do vetor original.
- Com a resposta das chamadas recursivas podemos chamar a função *merge* para obter um vetor ordenado.

Merge-Sort

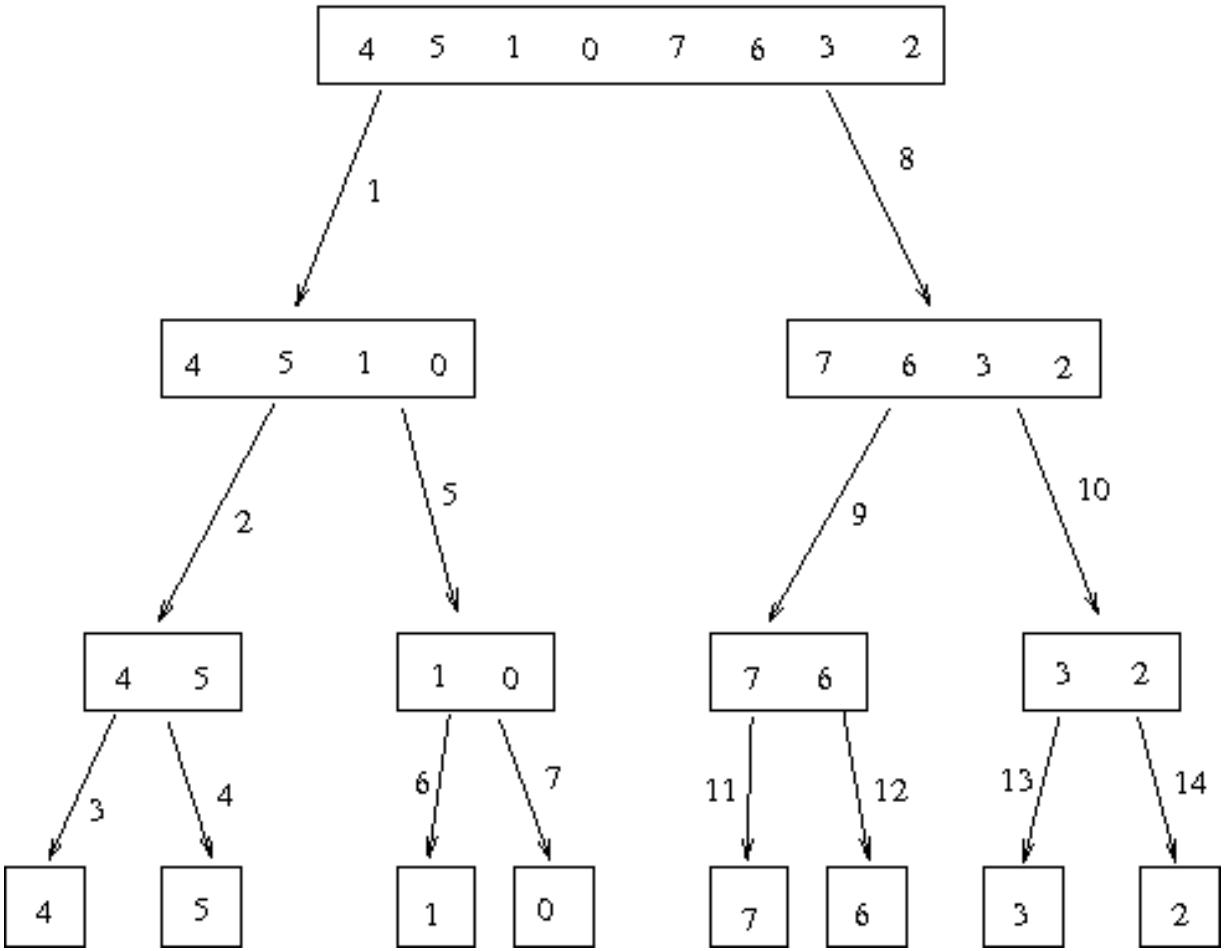
- O merge-sort resolve de forma recursiva dois sub-problemas, cada um contendo uma metade do vetor original.
- Com a resposta das chamadas recursivas podemos chamar a função *merge* para obter um vetor ordenado.

Merge-Sort

```
void mergeSort(int v[],int ini, int fim, int aux[]){
  int meio = (fim+ini)/2;
  if(ini < fim){
    mergeSort(v,ini,meio,aux);
    mergeSort(v,meio+1,fim,aux);
    merge(v,ini,meio,fim,aux);
  }
}
```

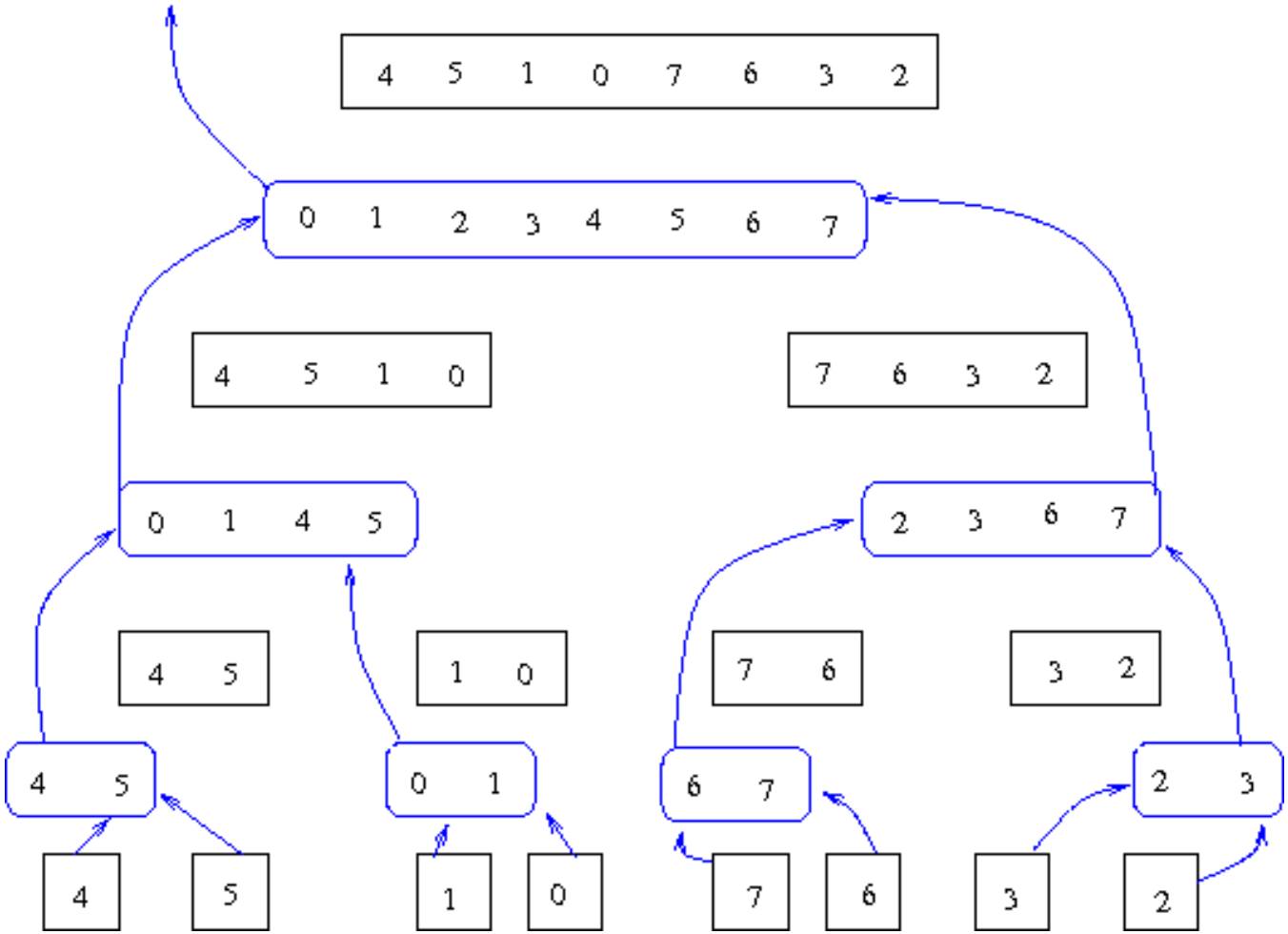
Merge-Sort

Abaixo temos um exemplo com a ordem de execução das chamadas recursivas.



Merge-Sort

Abaixo temos o retorno do exemplo anterior.



Merge-Sort: Exemplo de uso

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

void merge(int v[], int ini,  int meio, int fim, int aux[]);
void mergeSort(int v[],int ini, int fim, int aux[]);


int main(){
   int v[]={12,90, 47, -9, 78, 45, 78, 3323, 1, 2, 34, 20};
   int aux[12];
   int i;
   mergeSort(v, 0, 11, aux);
   for(i=0; i<12; i++)
     printf("\n %d",v[i]);
}
```

Exercícios

1. Mostre passo a passo a execução da função merge considerando dois sub-vetores: $(3, 5, 7, 10, 11, 12)$ e $(4, 6, 8, 9, 11, 13, 14)$.
2. Faça uma execução Passo-a-Passo do Merge-Sort para o vetor: $(30, 45, 21, 20, 6, 715, 100, 65, 33)$.
3. Reescreva o algoritmo Merge-Sort para que este passe a ordenar um vetor em ordem decrescente.
4. Considere o seguinte problema: Temos como entrada um vetor de inteiros v (não necessariamente ordenado), e um inteiro x. Desenvolva um algoritmo que determina se há dois números em v cuja soma seja x. Tente fazer o algoritmo o mais eficiente possível. Utilize um dos algoritmos de ordenação na sua solução.