# Ordenação por separação

## Aula 7

Diego Padilha Rubert

Faculdade de Computação
Universidade Federal de Mato Grosso do Sul

Algoritmos e Programação II

# Conteúdo da aula

# Introdução e motivação

- Operação básica em Computação
- Métodos mais eficientes de ordenação
- Ordenação com recursão
- Dividir para conquistar

# Introdução e motivação

- Operação básica em Computação
- Métodos mais eficientes de ordenação
- Ordenação com recursão
- Dividir para conquistar

# Introdução e motivação

- Operação básica em Computação
- Métodos mais eficientes de ordenação
- Ordenação com recursão
- Dividir para conquistar

# Introdução e motivação

- Operação básica em Computação
- Métodos mais eficientes de ordenação
- Ordenação com recursão
- Dividir para conquistar

# Problema da separação

## Problema

Rearranjar o vetor $v[p..r]$ de modo que tenhamos

$$v[p..q] \leqslant v[q+1..r]$$

para algum $q$ em $p..r-1$. A expressão $v[p..q] \leqslant v[q+1..r]$ significa que $v[i] \leqslant v[j]$ para todo $i$, $p \leqslant i \leqslant q$, e todo $j$, $q < j \leqslant r$.

# Problema da separação

```
int separa(int p, int r, int v[MAX])
{
   int x, i, j;

   x = v[p];
   i = p - 1;
   j = r + 1;
   while (i < j) {
      do {
         j--;
      } while (v[j] > x);
      do {
         i++;
      } while (v[i] < x);
      if (i < j)
         troca(&v[i], &v[j]);
   }
   return j;
}
```

# Problema da separação

- No início de cada iteração valem os seguintes invariantes:
  - $v[p..r]$ é uma permutação do vetor original,
  - $v[p..i] \leqslant x \leqslant v[j..r]$ e
  - $p \leqslant i \leqslant j \leqslant r$.
- Tempo de execução de pior caso é proporcional a $r - p + 1$, isto é, proporcional ao número de elementos do vetor.

# Problema da separação

- No início de cada iteração valem os seguintes invariantes:
  - $v[p..r]$ é uma permutação do vetor original,
  - $v[p..i] \leqslant x \leqslant v[j..r]$ e
  - $p \leqslant i \leqslant j \leqslant r$.
- Tempo de execução de pior caso é proporcional a $r - p + 1$, isto é, proporcional ao número de elementos do vetor.

# Problema da separação

- No início de cada iteração valem os seguintes invariantes:
  - $v[p..r]$ é uma permutação do vetor original,
  - $v[p..i] \leqslant x \leqslant v[j..r]$ e
  - $p \leqslant i \leqslant j \leqslant r$.
- Tempo de execução de pior caso é proporcional a $r - p + 1$, isto é, proporcional ao número de elementos do vetor.

# Problema da separação

- No início de cada iteração valem os seguintes invariantes:
  - $v[p..r]$ é uma permutação do vetor original,
  - $v[p..i] \leqslant x \leqslant v[j..r]$ e
  - $p \leqslant i \leqslant j \leqslant r$.
- Tempo de execução de pior caso é proporcional a $r - p + 1$, isto é, proporcional ao número de elementos do vetor.

# Ordenação por separação

- Separamos os elementos “pequenos” dos elementos “grandes” de um vetor $v$ com $r - p$ elementos
- Ordenamos recursivamente esses dois trechos de $v$
- “Colamos” esses dois trechos para obter o vetor $v$ original ordenado

# Ordenação por separação

- Separamos os elementos “pequenos” dos elementos “grandes” de um vetor $v$ com $r - p$ elementos
- Ordenamos recursivamente esses dois trechos de $v$
- “Colamos” esses dois trechos para obter o vetor $v$ original ordenado

# Ordenação por separação

- Separamos os elementos “pequenos” dos elementos “grandes” de um vetor $v$ com $r - p$ elementos
- Ordenamos recursivamente esses dois trechos de $v$
- “Colamos” esses dois trechos para obter o vetor $v$ original ordenado

# Ordenação por separação

```
/* Recebe um vetor v[p..r] e o rearranja em ordem crescente */
void quicksort(int p, int r, int v[MAX])
{
   int q;

   if (p < r) {
      q = separa(p, r, v);
      quicksort(p, q, v);
      quicksort(q+1, r, v);
   }
}
```

# Ordenação por separação

- Para ordenar um vetor $v[0..n-1]$ basta chamar a função `quicksort` com os seguintes argumentos:

```
quicksort(0, n-1, v);
```

- Um exemplo de execução da função `quicksort` para um vetor de entrada $v[0..7] = \{4, 6, 7, 3, 5, 1, 2, 8\}$ e chamada

```
quicksort(0, 7, v);
```

# Ordenação por separação

- Para ordenar um vetor $v[0..n-1]$ basta chamar a função `quicksort` com os seguintes argumentos:

```
quicksort(0, n-1, v);
```

- Um exemplo de execução da função `quicksort` para um vetor de entrada $v[0..7] = \{4, 6, 7, 3, 5, 1, 2, 8\}$ e chamada

```
quicksort(0, 7, v);
```

# Ordenação por separação

- Desempenho da função `quicksort` quando queremos ordenar um vetor $v[0..n-1]$
  - Proporcional ao número de comparações realizadas entre os elementos do vetor
  - Se o índice devolvido pela função `separa` sempre tiver valor mais ou menos médio de $p$ e $r$, então o número de comparações será aproximadamente $n \log_2 n$
  - Caso contrário, o número de comparações será da ordem de $n^2$
  - Portanto, o consumo de tempo de pior caso da ordenação por separação não é melhor que o dos métodos elementares
  - Felizmente, o pior caso para a ordenação por separação é raro. Dessa forma, o consumo de tempo *médio* da função `quicksort` é proporcional a $n \log_2 n$

# Ordenação por separação

- Desempenho da função `quicksort` quando queremos ordenar um vetor $v[0..n-1]$
  - Proporcional ao número de comparações realizadas entre os elementos do vetor
  - Se o índice devolvido pela função `separa` sempre tiver valor mais ou menos médio de $p$ e $r$, então o número de comparações será aproximadamente $n \log_2 n$
  - Caso contrário, o número de comparações será da ordem de $n^2$
  - Portanto, o consumo de tempo de pior caso da ordenação por separação não é melhor que o dos métodos elementares
  - Felizmente, o pior caso para a ordenação por separação é raro. Dessa forma, o consumo de tempo *médio* da função `quicksort` é proporcional a $n \log_2 n$

# Ordenação por separação

- Desempenho da função `quicksort` quando queremos ordenar um vetor $v[0..n-1]$
  - Proporcional ao número de comparações realizadas entre os elementos do vetor
  - Se o índice devolvido pela função `separa` sempre tiver valor mais ou menos médio de $p$ e $r$, então o número de comparações será aproximadamente $n \log_2 n$
  - Caso contrário, o número de comparações será da ordem de $n^2$
  - Portanto, o consumo de tempo de pior caso da ordenação por separação não é melhor que o dos métodos elementares
  - Felizmente, o pior caso para a ordenação por separação é raro. Dessa forma, o consumo de tempo *médio* da função `quicksort` é proporcional a $n \log_2 n$

# Ordenação por separação

- Desempenho da função `quicksort` quando queremos ordenar um vetor $v[0..n-1]$
  - Proporcional ao número de comparações realizadas entre os elementos do vetor
  - Se o índice devolvido pela função `separa` sempre tiver valor mais ou menos médio de $p$ e $r$, então o número de comparações será aproximadamente $n \log_2 n$
  - Caso contrário, o número de comparações será da ordem de $n^2$
  - Portanto, o consumo de tempo de pior caso da ordenação por separação não é melhor que o dos métodos elementares
  - Felizmente, o pior caso para a ordenação por separação é raro. Dessa forma, o consumo de tempo *médio* da função `quicksort` é proporcional a $n \log_2 n$

# Ordenação por separação

- Desempenho da função `quicksort` quando queremos ordenar um vetor $v[0..n-1]$
  - Proporcional ao número de comparações realizadas entre os elementos do vetor
  - Se o índice devolvido pela função `separa` sempre tiver valor mais ou menos médio de $p$ e $r$, então o número de comparações será aproximadamente $n \log_2 n$
  - Caso contrário, o número de comparações será da ordem de $n^2$
  - Portanto, o consumo de tempo de pior caso da ordenação por separação não é melhor que o dos métodos elementares
  - Felizmente, o pior caso para a ordenação por separação é raro. Dessa forma, o consumo de tempo *médio* da função `quicksort` é proporcional a $n \log_2 n$

# Ordenação por separação

- Desempenho da função `quicksort` quando queremos ordenar um vetor $v[0..n-1]$
  - Proporcional ao número de comparações realizadas entre os elementos do vetor
  - Se o índice devolvido pela função `separa` sempre tiver valor mais ou menos médio de $p$ e $r$, então o número de comparações será aproximadamente $n \log_2 n$
  - Caso contrário, o número de comparações será da ordem de $n^2$
  - Portanto, o consumo de tempo de pior caso da ordenação por separação não é melhor que o dos métodos elementares
  - Felizmente, o pior caso para a ordenação por separação é raro. Dessa forma, o consumo de tempo *médio* da função `quicksort` é proporcional a $n \log_2 n$

# Exercícios

1. Ilustre a operação da função `separa` sobre o vetor $v$ que contém os elementos do conjunto $\{13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21\}$.
2. Qual o valor de $q$ a função `separa` devolve quando todos os elementos no vetor $v[p..r]$ têm o mesmo valor?
3. A função `separa` produz o resultado correto quando $p = r$?
4. Escreva uma função que rearranje um vetor $v[p..r]$ de números inteiros de modo que os elementos negativos e nulos fiquem à esquerda e os positivos fiquem à direita. Em outras palavras, rearranje o vetor de modo que tenhamos $v[p..q-1] \leqslant 0$ e $v[q..r] > 0$ para algum $q$ em $p..r+1$. Procure escrever uma função eficiente que não use um vetor auxiliar.

# Exercícios

5. Digamos que um vetor $v[p..r]$ está **arrumado** se existe $q$ em $p..r$ que satisfaz

$$v[p..q-1] \leqslant v[q] < v[q+1..r] .$$

   Escreva uma função que decida se $v[p..r]$ está arrumado. Em caso afirmativo, sua função deve devolver o valor de $q$.

6. Que acontece se trocarmos a expressão `if (p < r)` pela expressão `if (p != r)` no corpo da função `quicksort`?

7. Compare as funções `quicksort` e `mergesort`. Discuta as semelhanças e diferenças.

8. Como você modificaria a função `quicksort` para ordenar elementos em ordem decrescente?

# Exercícios

9. Os bancos frequentemente gravam transações sobre uma conta corrente na ordem das datas das transações, mas muitas pessoas preferem receber seus extratos bancários em listagens ordenadas pelo número do cheque emitido. As pessoas em geral emitem cheques na ordem da numeração dos cheques, e comerciantes usualmente descontam estes cheques com rapidez razoável. O problema de converter uma lista ordenada por data de transação em uma lista ordenada por número de cheques é portanto o problema de ordenar uma entrada quase já ordenada. Argumente que, neste caso, o método de ordenação por inserção provavelmente se comportará melhor do que o método de ordenação por separação.

# Exercícios

10. Forneça um argumento cuidadoso para mostrar que a função `separa` é correta. Prove o seguinte:
    (a) Os índices $i$ e $j$ nunca referenciam um elemento de $v$ fora do intervalo $[p..r]$;
    (b) O índice $j$ não é igual a $r$ quando `separa` termina (ou seja, a partição é sempre não trivial);
    (c) Todo elemento de $v[p..j]$ é menor ou igual a todo elemento de $v[j+1..r]$ quando a função `separa` termina.
11. Escreva uma versão da função `quicksort` que coloque um vetor de cadeias de caracteres em ordem lexicográfica.

# Exercícios

12. Considere a seguinte solução do problema da separação, devido a N. Lomuto. Para separar $v[p..r]$, esta versão incrementa duas regiões, $v[p..i]$ e $v[i+1..j]$, tal que todo elemento na primeira região é menor ou igual a $x = v[r]$ e todo elemento na segunda região é maior que $x$.

```
int separa_Lomuto(int p, int r, int v[MAX])
{
   int x, i, j;

   x = v[r];
   i = p - 1;
   for (j = p; j <= r; j++)
      if (v[j] <= x) {
         i++;
         troca(&v[i], &v[j]);
      }
   if (i < r)
      return i;
   else
      return i - 1;
}
```

# Exercícios

13.

(a) Ilustre a operação da função `separa_Lomuto` sobre o vetor $v$ que contém os elementos $\{13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21\}$.
(b) Argumente que `separa_Lomuto` está correta.
(c) Qual o número máximo de vezes que um elemento pode ser movido pelas funções `separa` e `separa_Lomuto`? Justifique sua resposta.
(d) Argumente que `separa_Lomuto`, assim como `separa`, tem tempo de execução proporcional a $n$ sobre um vetor de $n = r - p + 1$ elementos.
(e) Como a troca da função `separa` pela função `separa_Lomuto` afeta o tempo de execução do método da ordenação por separação quando todos os valores de entrada são iguais?

# Exercícios

14. A função `quicksort` contém duas chamadas recursivas para ela própria. Depois da chamada da `separa`, o sub-vetor esquerdo é ordenado recursivamente e então o sub-vetor direito é ordenado recursivamente. A segunda chamada recursiva no corpo da função `quicksort` não é realmente necessária; ela pode ser evitada usando uma estrutura de controle iterativa. Essa técnica, chamada **recursão de cauda**, é fornecida automaticamente por bons compiladores. Considere a seguinte versão da ordenação por separação , que simula a recursão de cauda.

```
void quicksort2(int p, int r, int v[MAX])
{
   while (p < r) {
      q = separa(p, r, v);
      quicksort2(p, q, v);
      p = q + 1;
   }
}
```

# Exercícios

14. (*continuação*)
Ilustre a operação da função `quicksort2` sobre o vetor $v$ que contém os elementos $\{21, 7, 5, 11, 6, 42, 13, 2\}$. Use a chamada `quicksort2(0, n-1, v)`. Argumente que a função `quicksort2` ordena corretamente o vetor $v$.

# Exercícios

15. Um famoso programador propôs o seguinte método de ordenação de um vetor $v[0..n-1]$ de números inteiros:

```
void silly_sort(int i, int j, int v[MAX])
{
   int k;

   if (v[i] > v[j])
      troca(&v[i], &v[j]);
   if (i + 1 < j) {
      k = (j - i + 1) / 3;
      silly_sort(i, j-k, v);
      silly_sort(i+k, j, v);
      silly_sort(i, j-k, v);
   }
}
```

Ilustre a operação desse novo método de ordenação sobre o vetor $v$ que contém os elementos $\{21, 7, 5, 11, 6, 42, 13, 2\}$. Use a chamada `silly_sort(0, n-1, v)`. Argumente que a função `silly_sort` ordena corretamente o vetor $v$.

# Exercícios

16. Veja animações dos métodos de ordenação que já vimos nas seguintes páginas:
    - Sort Animation de R. Mohammadi;
    - Sorting Algorithms de J. Harrison;
    - Sorting Algorithms de P. Morin;
    - Sorting Algorithms Animations de D. R. Martin.
17. Familiarize-se com a função `qsort` da biblioteca `stdlib` da linguagem C.