Projeto e Análise de Algoritmos

Introdução

Lehilton Pedrosa

Primeiro Semestre de 2020

Agradecimentos

A maioria dos exemplos dos slides e exercícios adicionais foram ou criados e gentilmente cedidos pelo prof. Cid Carvalho de Souza e pela profa. Cândida Nunes da Silva (particularmente com modificações do prof. Orlando Lee); ou criados e gentilmente cedidos pelo prof. Flávio Keidi Miyazawa. Eu recriei ou reestruturei o conteúdo das unidades, possivelmente introduzindo erros, que devem ser reportados a mim.

O conjunto de slides de cada unidade do curso será disponibilizado como guia de estudos e deve ser usado unicamente para revisar as aulas. Para estudar e praticar, leia o livro-texto indicado e resolva os exercícios sugeridos.

Lehilton

Introdução à análise de algoritmos

Objetivos

O que veremos nesta disciplina?

1. Demonstrar correção de um algoritmo
   - como ter certeza de que a saída está correta
   - como convencer outras pessoas disso
2. Analisar a complexidade de um algoritmo
   - como estimar a quantidade de recursos utilizados
   - recursos podem ser tempo, memória, acesso a rede etc.

Objetivos

O que veremos nesta disciplina?

1. Demonstrar correção de um algoritmo
   - como ter certeza de que a saída está correta
   - como convencer outras pessoas disso

2. Analisar a complexidade de um algoritmo
   - como estimar a quantidade de recursos utilizados
   - recursos podem ser tempo, memória, acesso a rede etc.

Objetivos

O que veremos nesta disciplina?

1. Demonstrar correção de um algoritmo
   - como ter certeza de que a saída está correta
   - como convencer outras pessoas disso

2. Analisar a complexidade de um algoritmo
   - como estimar a quantidade de recursos utilizados
   - recursos podem ser tempo, memória, acesso a rede etc.

Objetivos

O que veremos nesta disciplina?

1. Demonstrar correção de um algoritmo
   - como ter certeza de que a saída está correta
   - como convencer outras pessoas disso

2. Analisar a complexidade de um algoritmo
   - como estimar a quantidade de recursos utilizados
   - recursos podem ser tempo, memória, acesso a rede etc.

Objetivos

O que veremos nesta disciplina?

1. Demonstrar correção de um algoritmo
   - como ter certeza de que a saída está correta
   - como convencer outras pessoas disso

2. Analisar a complexidade de um algoritmo
   - como estimar a quantidade de recursos utilizados
   - recursos podem ser tempo, memória, acesso a rede etc.

Objetivos

O que veremos nesta disciplina?

1. Demonstrar correção de um algoritmo
   - como ter certeza de que a saída está correta
   - como convencer outras pessoas disso

2. Analisar a complexidade de um algoritmo
   - como estimar a quantidade de recursos utilizados
   - recursos podem ser tempo, memória, acesso a rede etc.

Objetivos

O que veremos nesta disciplina?

1. Demonstrar correção de um algoritmo
   - como ter certeza de que a saída está correta
   - como convencer outras pessoas disso

2. Analisar a complexidade de um algoritmo
   - como estimar a quantidade de recursos utilizados
   - recursos podem ser tempo, memória, acesso a rede etc.

Objetivos

3. Utilizar técnicas conhecidas de projeto de algoritmos
   - divisão e conquista, programação dinâmica etc.
   - utilizar recursão adequadamente

4. Entender a dificuldade intrínseca de alguns problemas
   - inexistência de algoritmos eficientes
   - identificar os problemas intratáveis

Objetivos

3. Utilizar técnicas conhecidas de projeto de algoritmos
   - divisão e conquista, programação dinâmica etc.
   - utilizar recursão adequadamente

4. Entender a dificuldade intrínseca de alguns problemas
   - inexistência de algoritmos eficientes
   - identificar os problemas intratáveis

Objetivos

3. Utilizar técnicas conhecidas de projeto de algoritmos
   - divisão e conquista, programação dinâmica etc.
   - utilizar recursão adequadamente

4. Entender a dificuldade intrínseca de alguns problemas
   - inexistência de algoritmos eficientes
   - identificar os problemas intratáveis

Objetivos

3. Utilizar técnicas conhecidas de projeto de algoritmos
   - divisão e conquista, programação dinâmica etc.
   - utilizar recursão adequadamente

4. Entender a dificuldade intrínseca de alguns problemas
   - inexistência de algoritmos eficientes
   - identificar os problemas intratáveis

Objetivos

3. Utilizar técnicas conhecidas de projeto de algoritmos
   - divisão e conquista, programação dinâmica etc.
   - utilizar recursão adequadamente

4. Entender a dificuldade intrínseca de alguns problemas
   - inexistência de algoritmos eficientes
   - identificar os problemas intratáveis

Objetivos

3. Utilizar técnicas conhecidas de projeto de algoritmos
   - divisão e conquista, programação dinâmica etc.
   - utilizar recursão adequadamente

4. Entender a dificuldade intrínseca de alguns problemas
   - inexistência de algoritmos eficientes
   - identificar os problemas intratáveis

Problemas computacionais

Um problema computacional é uma relação entre um conjunto de instâncias e um conjunto de soluções:

- uma instância é um conjunto de valores conhecidos
- uma solução é um conjunto de valores a computar
- cada instância corresponde a uma ou mais soluções

Problemas computacionais

Um problema computacional é uma relação entre um conjunto de instâncias e um conjunto de soluções:

- uma instância é um conjunto de valores conhecidos
- uma solução é um conjunto de valores a computar
- cada instância corresponde a uma ou mais soluções

Problemas computacionais

Um problema computacional é uma relação entre um conjunto de instâncias e um conjunto de soluções:

- uma instância é um conjunto de valores conhecidos
- uma solução é um conjunto de valores a computar
- cada instância corresponde a uma ou mais soluções

Problemas computacionais

Um problema computacional é uma relação entre um conjunto de instâncias e um conjunto de soluções:

- uma instância é um conjunto de valores conhecidos
- uma solução é um conjunto de valores a computar
- cada instância corresponde a uma ou mais soluções

Exemplo de problema: teste de primalidade

Problema: determinar se um dado número é primo

- instâncias: números inteiros
- soluções: sim ou não

Exemplo:

- Instância: 9411461
- Solução: sim

Exemplo:

- Instância: 8411461
- Solução: não

Exemplo de problema: teste de primalidade

Problema: determinar se um dado número é primo

- instâncias: números inteiros
- soluções: sim ou não

Exemplo:

- Instância: 9411461
- Solução: sim

Exemplo:

- Instância: 8411461
- Solução: não

Exemplo de problema: teste de primalidade

Problema: determinar se um dado número é primo

- instâncias: números inteiros
- soluções: sim ou não

Exemplo:

- Instância: 9411461
- Solução: sim

Exemplo:

- Instância: 8411461
- Solução: não

Exemplo de problema: teste de primalidade

Problema: determinar se um dado número é primo

- instâncias: números inteiros
- soluções: sim ou não

Exemplo:

- Instância: 9411461
- Solução: sim

Exemplo:

- Instância: 8411461
- Solução: não

Exemplo de problema: teste de primalidade

Problema: determinar se um dado número é primo

- instâncias: números inteiros
- soluções: sim ou não

Exemplo:

- Instância: 9411461
- Solução: sim

Exemplo:

- Instância: 8411461
- Solução: não

Exemplo de problema: teste de primalidade

Problema: determinar se um dado número é primo

- instâncias: números inteiros
- soluções: sim ou não

Exemplo:

- Instância: 9411461
- Solução: sim

Exemplo:

- Instância: 8411461
- Solução: não

Exemplo de problema: teste de primalidade

Problema: determinar se um dado número é primo

- instâncias: números inteiros
- soluções: sim ou não

Exemplo:

- Instância: 9411461
- Solução: sim

Exemplo:

- Instância: 8411461
- Solução: não

Exemplo de problema: teste de primalidade

Problema: determinar se um dado número é primo

- instâncias: números inteiros
- soluções: sim ou não

Exemplo:

- Instância: 9411461
- Solução: sim

Exemplo:

- Instância: 8411461
- Solução: não

Exemplo de problema: teste de primalidade

Problema: determinar se um dado número é primo

- instâncias: números inteiros
- soluções: sim ou não

Exemplo:

- Instância: 9411461
- Solução: sim

Exemplo:

- Instância: 8411461
- Solução: não

Exemplo de problema: ordenação

Problema: ordenar os elementos de um vetor

- instâncias: conjunto de vetores de inteiros
- soluções: conjunto de vetores de inteiros em ordem crescente

Exemplo:

- Instância:

1										n
33	55	33	44	33	22	11	99	22	55	77

- Solução:

1										n
11	22	22	33	33	33	44	55	55	77	99

Exemplo de problema: ordenação

Problema: ordenar os elementos de um vetor

- instâncias: conjunto de vetores de inteiros
- soluções: conjunto de vetores de inteiros em ordem crescente

Exemplo:

- Instância:

1										n
33	55	33	44	33	22	11	99	22	55	77

- Solução:

1										n
11	22	22	33	33	33	44	55	55	77	99

Exemplo de problema: ordenação

Problema: ordenar os elementos de um vetor

- instâncias: conjunto de vetores de inteiros
- soluções: conjunto de vetores de inteiros em ordem crescente

Exemplo:

- Instância:

1										n
33	55	33	44	33	22	11	99	22	55	77

- Solução:

1										n
11	22	22	33	33	33	44	55	55	77	99

Exemplo de problema: ordenação

Problema: ordenar os elementos de um vetor

- instâncias: conjunto de vetores de inteiros
- soluções: conjunto de vetores de inteiros em ordem crescente

Exemplo:

- Instância:

1										n
33	55	33	44	33	22	11	99	22	55	77

- Solução:

1										n
11	22	22	33	33	33	44	55	55	77	99

Exemplo de problema: ordenação

Problema: ordenar os elementos de um vetor

- instâncias: conjunto de vetores de inteiros
- soluções: conjunto de vetores de inteiros em ordem crescente

Exemplo:

- Instância:

1										n
33	55	33	44	33	22	11	99	22	55	77

- Solução:

1										n
11	22	22	33	33	33	44	55	55	77	99

Exemplo de problema: ordenação

Problema: ordenar os elementos de um vetor

- instâncias: conjunto de vetores de inteiros
- soluções: conjunto de vetores de inteiros em ordem crescente

Exemplo:

- Instância:

1										n
33	55	33	44	33	22	11	99	22	55	77

- Solução:

1										n
11	22	22	33	33	33	44	55	55	77	99

Algoritmos

Um algoritmo é uma sequência de instruções que

- recebe uma instância de um problema computacional
- devolve uma solução correspondente à instância recebida

Observações:

- a instância recebida é chamada de entrada
- a solução devolvida é chamada de saída
- toda instrução deve ser bem definida

Algoritmos

Um algoritmo é uma sequência de instruções que

- recebe uma instância de um problema computacional
- devolve uma solução correspondente à instância recebida

Observações:

- a instância recebida é chamada de entrada
- a solução devolvida é chamada de saída
- toda instrução deve ser bem definida

Algoritmos

Um algoritmo é uma sequência de instruções que

- recebe uma instância de um problema computacional
- devolve uma solução correspondente à instância recebida

Observações:

- a instância recebida é chamada de entrada
- a solução devolvida é chamada de saída
- toda instrução deve ser bem definida

Algoritmos

Um algoritmo é uma sequência de instruções que

- recebe uma instância de um problema computacional
- devolve uma solução correspondente à instância recebida

Observações:

- a instância recebida é chamada de entrada
- a solução devolvida é chamada de saída
- toda instrução deve ser bem definida

Algoritmos

Um algoritmo é uma sequência de instruções que

- recebe uma instância de um problema computacional
- devolve uma solução correspondente à instância recebida

Observações:

- a instância recebida é chamada de entrada
- a solução devolvida é chamada de saída
- toda instrução deve ser bem definida

Algoritmos

Um algoritmo é uma sequência de instruções que

- recebe uma instância de um problema computacional
- devolve uma solução correspondente à instância recebida

Observações:

- a instância recebida é chamada de entrada
- a solução devolvida é chamada de saída
- toda instrução deve ser bem definida

Algoritmos

Um algoritmo é uma sequência de instruções que

- recebe uma instância de um problema computacional
- devolve uma solução correspondente à instância recebida

Observações:

- a instância recebida é chamada de entrada
- a solução devolvida é chamada de saída
- toda instrução deve ser bem definida

Descrição de algoritmos

Podemos escrever um algoritmo de várias maneiras:

- em uma linguagem de programação, como C, Pascal, Java, Python...
- em português, ou outra língua natural
- em pseudocódigo, como no livro de CLRS

Usaremos apenas as duas últimas opções!

Descrição de algoritmos

Podemos escrever um algoritmo de várias maneiras:

- em uma linguagem de programação, como C, Pascal, Java, Python...
- em português, ou outra língua natural
- em pseudocódigo, como no livro de CLRS

Usaremos apenas as duas últimas opções!

Descrição de algoritmos

Podemos escrever um algoritmo de várias maneiras:

- em uma linguagem de programação, como C, Pascal, Java, Python...
- em português, ou outra língua natural
- em pseudocódigo, como no livro de CLRS

Usaremos apenas as duas últimas opções!

Descrição de algoritmos

Podemos escrever um algoritmo de várias maneiras:

- em uma linguagem de programação, como C, Pascal, Java, Python...
- em português, ou outra língua natural
- em pseudocódigo, como no livro de CLRS

Usaremos apenas as duas últimas opções!

Descrição de algoritmos

Podemos escrever um algoritmo de várias maneiras:

- em uma linguagem de programação, como C, Pascal, Java, Python...
- em português, ou outra língua natural
- em pseudocódigo, como no livro de CLRS

Usaremos apenas as duas últimas opções!

Exemplo de pseudocódigo

Um algoritmo para o problema da ordenação:

```
INSERTION-SORT(A, n)
1   para j ← 2 até n faça
2       chave ← A[j]
3       i ← j − 1
4       enquanto i ≥ 1 e A[i] > chave faça
5           A[i + 1] ← A[i]
6           i ← i − 1
7       A[i + 1] ← chave
```

Más práticas

- Não misture código com pseudocódigo:

```
for (i = 0; i < n; i++) ...
```

```
if (A[i] >= chave)
    break
```

- Não escreva frases confusas:

```
se A[i] > chave então
    troque as posições dos elementos
```

```
chave ← pega o próximo elemento
i ← procura posição da chave
```

- Evite algoritmos complicados ou com muitas variáveis:

```
se j = 1 então
    A[2] ← A[1]
    A[1] ← chave
senão
    enquanto i ≥ 1 e A[i] ≥ chave faça
        ...
```

Modelo computacional

Só podemos escrever instruções bem definidas:

- o resultado de cada instrução é inambíguo e depende somente do estado corrente da execução
- deve ser possível executar cada instrução usando o computador adotado

O conjunto de instruções permitidas é determinado pelo que chamamos de modelo computacional

Modelo computacional

Só podemos escrever instruções bem definidas:

- o resultado de cada instrução é inambíguo e depende somente do estado corrente da execução
- deve ser possível executar cada instrução usando o computador adotado

O conjunto de instruções permitidas é determinado pelo que chamamos de modelo computacional

Modelo computacional

Só podemos escrever instruções bem definidas:

- o resultado de cada instrução é inambíguo e depende somente do estado corrente da execução
- deve ser possível executar cada instrução usando o computador adotado

O conjunto de instruções permitidas é determinado pelo que chamamos de modelo computacional

Modelo computacional

Só podemos escrever instruções bem definidas:

- o resultado de cada instrução é inambíguo e depende somente do estado corrente da execução
- deve ser possível executar cada instrução usando o computador adotado

O conjunto de instruções permitidas é determinado pelo que chamamos de modelo computacional

Correção de algoritmos

Um algoritmo está correto se:

1. só utiliza instruções do modelo de computação adotado,
2. termina para toda instância do problema e
3. devolve uma solução que correspondente à instância recebida.

Ao escrever um algoritmo, sempre devemos

1. Testar o algoritmo
   - com uma ou mais instâncias de exemplo
   - executando ou simulando o algoritmo
2. Demonstrar que o algoritmo está correto
   - escrever uma prova formal genérica
   - vale para toda instância do problema

Correção de algoritmos

Um algoritmo está correto se:

1. só utiliza instruções do modelo de computação adotado,
2. termina para toda instância do problema e
3. devolve uma solução que correspondente à instância recebida.

Ao escrever um algoritmo, sempre devemos

1. Testar o algoritmo
   - com uma ou mais instâncias de exemplo
   - executando ou simulando o algoritmo
2. Demonstrar que o algoritmo está correto
   - escrever uma prova formal genérica
   - vale para toda instância do problema

Correção de algoritmos

Um algoritmo está correto se:

1. só utiliza instruções do modelo de computação adotado,
2. termina para toda instância do problema e
3. devolve uma solução que correspondente à instância recebida.

Ao escrever um algoritmo, sempre devemos

1. Testar o algoritmo
   - com uma ou mais instâncias de exemplo
   - executando ou simulando o algoritmo
2. Demonstrar que o algoritmo está correto
   - escrever uma prova formal genérica
   - vale para toda instância do problema

Correção de algoritmos

Um algoritmo está correto se:

1. só utiliza instruções do modelo de computação adotado,
2. termina para toda instância do problema e
3. devolve uma solução que correspondente à instância recebida.

Ao escrever um algoritmo, sempre devemos

1. Testar o algoritmo
   - com uma ou mais instâncias de exemplo
   - executando ou simulando o algoritmo
2. Demonstrar que o algoritmo está correto
   - escrever uma prova formal genérica
   - vale para toda instância do problema

Correção de algoritmos

Um algoritmo está correto se:

1. só utiliza instruções do modelo de computação adotado,
2. termina para toda instância do problema e
3. devolve uma solução que correspondente à instância recebida.

Ao escrever um algoritmo, sempre devemos

1. Testar o algoritmo
   - com uma ou mais instâncias de exemplo
   - executando ou simulando o algoritmo
2. Demonstrar que o algoritmo está correto
   - escrever uma prova formal genérica
   - vale para toda instância do problema

Correção de algoritmos

Um algoritmo está correto se:

1. só utiliza instruções do modelo de computação adotado,
2. termina para toda instância do problema e
3. devolve uma solução que correspondente à instância recebida.

Ao escrever um algoritmo, sempre devemos

1. Testar o algoritmo
   - com uma ou mais instâncias de exemplo
   - executando ou simulando o algoritmo
2. Demonstrar que o algoritmo está correto
   - escrever uma prova formal genérica
   - vale para toda instância do problema

Correção de algoritmos

Um algoritmo está correto se:

1. só utiliza instruções do modelo de computação adotado,
2. termina para toda instância do problema e
3. devolve uma solução que correspondente à instância recebida.

Ao escrever um algoritmo, sempre devemos

1. Testar o algoritmo
   - com uma ou mais instâncias de exemplo
   - executando ou simulando o algoritmo
2. Demonstrar que o algoritmo está correto
   - escrever uma prova formal genérica
   - vale para toda instância do problema

Correção de algoritmos

Um algoritmo está correto se:

1. só utiliza instruções do modelo de computação adotado,
2. termina para toda instância do problema e
3. devolve uma solução que correspondente à instância recebida.

Ao escrever um algoritmo, sempre devemos

1. Testar o algoritmo
   - com uma ou mais instâncias de exemplo
   - executando ou simulando o algoritmo
2. Demonstrar que o algoritmo está correto
   - escrever uma prova formal genérica
   - vale para toda instância do problema

Correção de algoritmos

Um algoritmo está correto se:

1. só utiliza instruções do modelo de computação adotado,
2. termina para toda instância do problema e
3. devolve uma solução que correspondente à instância recebida.

Ao escrever um algoritmo, sempre devemos

1. Testar o algoritmo
   - com uma ou mais instâncias de exemplo
   - executando ou simulando o algoritmo
2. Demonstrar que o algoritmo está correto
   - escrever uma prova formal genérica
   - vale para toda instância do problema

Correção de algoritmos

Um algoritmo está correto se:

1. só utiliza instruções do modelo de computação adotado,
2. termina para toda instância do problema e
3. devolve uma solução que correspondente à instância recebida.

Ao escrever um algoritmo, sempre devemos

1. Testar o algoritmo
   - com uma ou mais instâncias de exemplo
   - executando ou simulando o algoritmo
2. Demonstrar que o algoritmo está correto
   - escrever uma prova formal genérica
   - vale para toda instância do problema

Correção de algoritmos

Um algoritmo está correto se:

1. só utiliza instruções do modelo de computação adotado,
2. termina para toda instância do problema e
3. devolve uma solução que correspondente à instância recebida.

Ao escrever um algoritmo, sempre devemos

1. Testar o algoritmo
   - com uma ou mais instâncias de exemplo
   - executando ou simulando o algoritmo
2. Demonstrar que o algoritmo está correto
   - escrever uma prova formal genérica
   - vale para toda instância do problema

Complexidade de algoritmos

- Nem sempre é viável executar um algoritmo
  - pode levar anos ou séculos para terminar
  - pode precisar de mais memória do que há disponível
- Queremos
  1. projetar algoritmos eficientes
  2. comparar algoritmos diferentes
- Para isso, precisamos medir a complexidade do algoritmo
  - independentemente de quem programou,
  - da linguagem em foi escrito
  - e da máquina a ser usada

Complexidade de algoritmos

- Nem sempre é viável executar um algoritmo
  - pode levar anos ou séculos para terminar
  - pode precisar de mais memória do que há disponível
- Queremos
  1. projetar algoritmos eficientes
  2. comparar algoritmos diferentes
- Para isso, precisamos medir a complexidade do algoritmo
  - independentemente de quem programou,
  - da linguagem em foi escrito
  - e da máquina a ser usada

Complexidade de algoritmos

- Nem sempre é viável executar um algoritmo
  - pode levar anos ou séculos para terminar
  - pode precisar de mais memória do que há disponível

- Queremos
  1. projetar algoritmos eficientes
  2. comparar algoritmos diferentes

- Para isso, precisamos medir a complexidade do algoritmo
  - independentemente de quem programou,
  - da linguagem em foi escrito
  - e da máquina a ser usada

Complexidade de algoritmos

- Nem sempre é viável executar um algoritmo
  - pode levar anos ou séculos para terminar
  - pode precisar de mais memória do que há disponível

- Queremos
  1. projetar algoritmos eficientes
  2. comparar algoritmos diferentes

- Para isso, precisamos medir a complexidade do algoritmo
  - independentemente de quem programou,
  - da linguagem em foi escrito
  - e da máquina a ser usada

Complexidade de algoritmos

- Nem sempre é viável executar um algoritmo
  - pode levar anos ou séculos para terminar
  - pode precisar de mais memória do que há disponível

- Queremos
  1. projetar algoritmos eficientes
  2. comparar algoritmos diferentes

- Para isso, precisamos medir a complexidade do algoritmo
  - independentemente de quem programou,
  - da linguagem em foi escrito
  - e da máquina a ser usada

Complexidade de algoritmos

- Nem sempre é viável executar um algoritmo
  - pode levar anos ou séculos para terminar
  - pode precisar de mais memória do que há disponível

- Queremos
  1. projetar algoritmos eficientes
  2. comparar algoritmos diferentes

- Para isso, precisamos medir a complexidade do algoritmo
  - independentemente de quem programou,
  - da linguagem em foi escrito
  - e da máquina a ser usada

Complexidade de algoritmos

- Nem sempre é viável executar um algoritmo
  - pode levar anos ou séculos para terminar
  - pode precisar de mais memória do que há disponível

- Queremos
  1. projetar algoritmos eficientes
  2. comparar algoritmos diferentes

- Para isso, precisamos medir a complexidade do algoritmo
  - independentemente de quem programou,
  - da linguagem em foi escrito
  - e da máquina a ser usada

Complexidade de algoritmos

- Nem sempre é viável executar um algoritmo
  - pode levar anos ou séculos para terminar
  - pode precisar de mais memória do que há disponível

- Queremos
  1. projetar algoritmos eficientes
  2. comparar algoritmos diferentes

- Para isso, precisamos medir a complexidade do algoritmo
  - independentemente de quem programou,
  - da linguagem em foi escrito
  - e da máquina a ser usada

Complexidade de algoritmos

- Nem sempre é viável executar um algoritmo
  - pode levar anos ou séculos para terminar
  - pode precisar de mais memória do que há disponível

- Queremos
  1. projetar algoritmos eficientes
  2. comparar algoritmos diferentes

- Para isso, precisamos medir a complexidade do algoritmo
  - independentemente de quem programou,
  - da linguagem em foi escrito
  - e da máquina a ser usada

Complexidade de algoritmos

- Nem sempre é viável executar um algoritmo
  - pode levar anos ou séculos para terminar
  - pode precisar de mais memória do que há disponível

- Queremos
  1. projetar algoritmos eficientes
  2. comparar algoritmos diferentes

- Para isso, precisamos medir a complexidade do algoritmo
  - independentemente de quem programou,
  - da linguagem em foi escrito
  - e da máquina a ser usada

Modelo computacional e complexidade

Vamos estimar o tempo de execução de um algoritmo

- contamos o número de instruções elementares executadas
- supomos que cada instrução elementar consome um tempo contante

A análise de complexidade depende sempre do modelo computacional adotado!

- queremos um modelo computacional realista
- o conjunto de instruções elementares deve ser compatível com os computadores modernos
- mas deve ser suficientemente genérico para as diferentes arquiteturas

Modelo computacional e complexidade

Vamos estimar o tempo de execução de um algoritmo

- contamos o número de instruções elementares executadas
- supomos que cada instrução elementar consome um tempo contante

A análise de complexidade depende sempre do modelo computacional adotado!

- queremos um modelo computacional realista
- o conjunto de instruções elementares deve ser compatível com os computadores modernos
- mas deve ser suficientemente genérico para as diferentes arquiteturas

Modelo computacional e complexidade

Vamos estimar o tempo de execução de um algoritmo

- contamos o número de instruções elementares executadas
- supomos que cada instrução elementar consome um tempo contante

A análise de complexidade depende sempre do modelo computacional adotado!

- queremos um modelo computacional realista
- o conjunto de instruções elementares deve ser compatível com os computadores modernos
- mas deve ser suficientemente genérico para as diferentes arquiteturas

Modelo computacional e complexidade

Vamos estimar o tempo de execução de um algoritmo

- contamos o número de instruções elementares executadas
- supomos que cada instrução elementar consome um tempo contante

A análise de complexidade depende sempre do modelo computacional adotado!

- queremos um modelo computacional realista
- o conjunto de instruções elementares deve ser compatível com os computadores modernos
- mas deve ser suficientemente genérico para as diferentes arquiteturas

Modelo computacional e complexidade

Vamos estimar o tempo de execução de um algoritmo

- contamos o número de instruções elementares executadas
- supomos que cada instrução elementar consome um tempo contante

A análise de complexidade depende sempre do modelo computacional adotado!

- queremos um modelo computacional realista
- o conjunto de instruções elementares deve ser compatível com os computadores modernos
- mas deve ser suficientemente genérico para as diferentes arquiteturas

Modelo computacional e complexidade

Vamos estimar o tempo de execução de um algoritmo

- contamos o número de instruções elementares executadas
- supomos que cada instrução elementar consome um tempo contante

A análise de complexidade depende sempre do modelo computacional adotado!

- queremos um modelo computacional realista
- o conjunto de instruções elementares deve ser compatível com os computadores modernos
- mas deve ser suficientemente genérico para as diferentes arquiteturas

Modelo computacional e complexidade

Vamos estimar o tempo de execução de um algoritmo

- contamos o número de instruções elementares executadas
- supomos que cada instrução elementar consome um tempo contante

A análise de complexidade depende sempre do modelo computacional adotado!

- queremos um modelo computacional realista
- o conjunto de instruções elementares deve ser compatível com os computadores modernos
- mas deve ser suficientemente genérico para as diferentes arquiteturas

Máquinas RAM

Usaremos o Modelo Abstrato RAM (Random Access Machine)

- simula máquinas convencionais
- possui um único processador sequencial
- tipos básicos são números inteiros e pontos flutuantes
- cada *palavra de memória* tem tamanho limitado, i.e., valores não podem ser arbitrários

Instruções elementares

- operações aritméticas como soma, subtração, produto...
- acesso direto às posições da memória
- comandos de fluxo de controle (`se`, `enquanto`...)

Operações como exponenciação não são elementares.

Máquinas RAM

Usaremos o Modelo Abstrato RAM (Random Access Machine)

- simula máquinas convencionais
- possui um único processador sequencial
- tipos básicos são números inteiros e pontos flutuantes
- cada *palavra de memória* tem tamanho limitado, i.e., valores não podem ser arbitrários

Instruções elementares

- operações aritméticas como soma, subtração, produto...
- acesso direto às posições da memória
- comandos de fluxo de controle (`se`, `enquanto`...)

Operações como exponenciação não são elementares.

Máquinas RAM

Usaremos o Modelo Abstrato RAM (Random Access Machine)

- simula máquinas convencionais
- possui um único processador sequencial
- tipos básicos são números inteiros e pontos flutuantes
- cada *palavra de memória* tem tamanho limitado, i.e., valores não podem ser arbitrários

Instruções elementares

- operações aritméticas como soma, subtração, produto...
- acesso direto às posições da memória
- comandos de fluxo de controle (`se`, `enquanto`...)

Operações como exponenciação não são elementares.

Máquinas RAM

Usaremos o Modelo Abstrato RAM (Random Access Machine)

- simula máquinas convencionais
- possui um único processador sequencial
- tipos básicos são números inteiros e pontos flutuantes
- cada *palavra de memória* tem tamanho limitado, i.e., valores não podem ser arbitrários

Instruções elementares

- operações aritméticas como soma, subtração, produto...
- acesso direto às posições da memória
- comandos de fluxo de controle (`se`, `enquanto`...)

Operações como exponenciação não são elementares.

Máquinas RAM

Usaremos o Modelo Abstrato RAM (Random Access Machine)

- simula máquinas convencionais
- possui um único processador sequencial
- tipos básicos são números inteiros e pontos flutuantes
- cada *palavra de memória* tem tamanho limitado, i.e., valores não podem ser arbitrários

Instruções elementares

- operações aritméticas como soma, subtração, produto...
- acesso direto às posições da memória
- comandos de fluxo de controle (`se`, `enquanto`...)

Operações como exponenciação não são elementares.

Máquinas RAM

Usaremos o Modelo Abstrato RAM (Random Access Machine)

- simula máquinas convencionais
- possui um único processador sequencial
- tipos básicos são números inteiros e pontos flutuantes
- cada *palavra de memória* tem tamanho limitado, i.e., valores não podem ser arbitrários

Instruções elementares

- operações aritméticas como soma, subtração, produto...
- acesso direto às posições da memória
- comandos de fluxo de controle (`se`, `enquanto`...)

Operações como exponenciação não são elementares.

Máquinas RAM

Usaremos o Modelo Abstrato RAM (Random Access Machine)

- simula máquinas convencionais
- possui um único processador sequencial
- tipos básicos são números inteiros e pontos flutuantes
- cada *palavra de memória* tem tamanho limitado, i.e., valores não podem ser arbitrários

Instruções elementares

- operações aritméticas como soma, subtração, produto...
- acesso direto às posições da memória
- comandos de fluxo de controle (`se`, `enquanto`...)

Operações como exponenciação não são elementares.

Máquinas RAM

Usaremos o Modelo Abstrato RAM (Random Access Machine)

- simula máquinas convencionais
- possui um único processador sequencial
- tipos básicos são números inteiros e pontos flutuantes
- cada *palavra de memória* tem tamanho limitado, i.e., valores não podem ser arbitrários

Instruções elementares

- operações aritméticas como soma, subtração, produto...
- acesso direto às posições da memória
- comandos de fluxo de controle (`se`, `enquanto`...)

Operações como exponenciação não são elementares.

Máquinas RAM

Usaremos o Modelo Abstrato RAM (Random Access Machine)

- simula máquinas convencionais
- possui um único processador sequencial
- tipos básicos são números inteiros e pontos flutuantes
- cada *palavra de memória* tem tamanho limitado, i.e., valores não podem ser arbitrários

Instruções elementares

- operações aritméticas como soma, subtração, produto...
- acesso direto às posições da memória
- comandos de fluxo de controle (**se**, **enquanto**...)

Operações como exponenciação não são elementares.

Máquinas RAM

Usaremos o Modelo Abstrato RAM (Random Access Machine)

- simula máquinas convencionais
- possui um único processador sequencial
- tipos básicos são números inteiros e pontos flutuantes
- cada *palavra de memória* tem tamanho limitado, i.e., valores não podem ser arbitrários

Instruções elementares

- operações aritméticas como soma, subtração, produto...
- acesso direto às posições da memória
- comandos de fluxo de controle (**se**, **enquanto**...)

Operações como exponenciação não são elementares.

Tamanho da entrada

Um parâmetro importante é o tamanho da entrada:

- normalmente proporcional ao número de bits da entrada
- também usamos o número de elementos do vetor

Problema: Primalidade

- Entrada: inteiro n
- Tamanho: $\lceil \log_2 n \rceil$ bits

Problema: Ordenação

- Entrada: vetor $A[1 \ldots n]$
- Tamanho: $n\lceil \log_2 M \rceil$ bits, onde M é o máximo em $A[1 \ldots n]$

Tamanho da entrada

Um parâmetro importante é o tamanho da entrada:

- normalmente proporcional ao número de bits da entrada
- também usamos o número de elementos do vetor

Problema: Primalidade

- Entrada: inteiro n
- Tamanho: $\lceil \log_2 n \rceil$ bits

Problema: Ordenação

- Entrada: vetor $A[1 \ldots n]$
- Tamanho: $n\lceil \log_2 M \rceil$ bits, onde M é o máximo em $A[1 \ldots n]$

Tamanho da entrada

Um parâmetro importante é o tamanho da entrada:

- normalmente proporcional ao número de bits da entrada
- também usamos o número de elementos do vetor

Problema: Primalidade

- Entrada: inteiro n
- Tamanho: $\lceil \log_2 n \rceil$ bits

Problema: Ordenação

- Entrada: vetor $A[1 \ldots n]$
- Tamanho: $n\lceil \log_2 M \rceil$ bits, onde M é o máximo em $A[1 \ldots n]$

Tamanho da entrada

Um parâmetro importante é o tamanho da entrada:

- normalmente proporcional ao número de bits da entrada
- também usamos o número de elementos do vetor

Problema: Primalidade

- Entrada: inteiro n
- Tamanho: $\lceil \log_2 n \rceil$ bits

Problema: Ordenação

- Entrada: vetor $A[1 \ldots n]$
- Tamanho: $n\lceil \log_2 M \rceil$ bits, onde M é o máximo em $A[1 \ldots n]$

Tamanho da entrada

Um parâmetro importante é o tamanho da entrada:

- normalmente proporcional ao número de bits da entrada
- também usamos o número de elementos do vetor

Problema: Primalidade

- Entrada: inteiro n
- Tamanho: $\lceil \log_2 n \rceil$ bits

Problema: Ordenação

- Entrada: vetor $A[1 \ldots n]$
- Tamanho: $n\lceil \log_2 M \rceil$ bits, onde M é o máximo em $A[1 \ldots n]$

Tamanho da entrada

Um parâmetro importante é o tamanho da entrada:

- normalmente proporcional ao número de bits da entrada
- também usamos o número de elementos do vetor

Problema: Primalidade

- Entrada: inteiro n
- Tamanho: $\lceil \log_2 n \rceil$ bits

Problema: Ordenação

- Entrada: vetor $A[1 \ldots n]$
- Tamanho: $n\lceil \log_2 M \rceil$ bits, onde M é o máximo em $A[1 \ldots n]$

Tamanho da entrada

Um parâmetro importante é o tamanho da entrada:

- normalmente proporcional ao número de bits da entrada
- também usamos o número de elementos do vetor

Problema: Primalidade

- Entrada: inteiro n
- Tamanho: $\lceil \log_2 n \rceil$ bits

Problema: Ordenação

- Entrada: vetor $A[1 \ldots n]$
- Tamanho: $n\lceil \log_2 M \rceil$ bits, onde M é o máximo em $A[1 \ldots n]$

Tamanho da entrada

Um parâmetro importante é o tamanho da entrada:

- normalmente proporcional ao número de bits da entrada
- também usamos o número de elementos do vetor

Problema: Primalidade

- Entrada: inteiro n
- Tamanho: $\lceil \log_2 n \rceil$ bits

Problema: Ordenação

- Entrada: vetor $A[1 \ldots n]$
- Tamanho: $n\lceil \log_2 M \rceil$ bits, onde M é o máximo em $A[1 \ldots n]$

Tamanho da entrada

Um parâmetro importante é o tamanho da entrada:

- normalmente proporcional ao número de bits da entrada
- também usamos o número de elementos do vetor

Problema: Primalidade

- Entrada: inteiro n
- Tamanho: $\lceil \log_2 n \rceil$ bits

Problema: Ordenação

- Entrada: vetor $A[1 \ldots n]$
- Tamanho: $n\lceil \log_2 M \rceil$ bits, onde M é o máximo em $A[1 \ldots n]$

Análise assintótica e de pior caso

Consideramos apenas instâncias grandes

- o número de instruções normalmente cresce com o tamanho da entrada n
- instâncias com tamanho limitado por constante gastam tempo constante

Fazemos apenas análise de pior caso

- restringimos a entradas com um dado tamanho n
- consideramos apenas uma instância para a qual o algoritmo executa o maior número de instruções

Denotamos por $T(n)$ o número de instruções executadas no pior caso para entradas de tamanho n

Análise assintótica e de pior caso

Consideramos apenas instâncias grandes

- o número de instruções normalmente cresce com o tamanho da entrada n
- instâncias com tamanho limitado por constante gastam tempo constante

Fazemos apenas análise de pior caso

- restringimos a entradas com um dado tamanho n
- consideramos apenas uma instância para a qual o algoritmo executa o maior número de instruções

Denotamos por $T(n)$ o número de instruções executadas no pior caso para entradas de tamanho n

Análise assintótica e de pior caso

Consideramos apenas instâncias grandes

- o número de instruções normalmente cresce com o tamanho da entrada n
- instâncias com tamanho limitado por constante gastam tempo constante

Fazemos apenas análise de pior caso

- restringimos a entradas com um dado tamanho n
- consideramos apenas uma instância para a qual o algoritmo executa o maior número de instruções

Denotamos por $T(n)$ o número de instruções executadas no pior caso para entradas de tamanho n

Análise assintótica e de pior caso

Consideramos apenas instâncias grandes

- o número de instruções normalmente cresce com o tamanho da entrada n
- instâncias com tamanho limitado por constante gastam tempo constante

Fazemos apenas análise de pior caso

- restringimos a entradas com um dado tamanho n
- consideramos apenas uma instância para a qual o algoritmo executa o maior número de instruções

Denotamos por $T(n)$ o número de instruções executadas no pior caso para entradas de tamanho n

Análise assintótica e de pior caso

Consideramos apenas instâncias grandes

- o número de instruções normalmente cresce com o tamanho da entrada n
- instâncias com tamanho limitado por constante gastam tempo constante

Fazemos apenas análise de pior caso

- restringimos a entradas com um dado tamanho n
- consideramos apenas uma instância para a qual o algoritmo executa o maior número de instruções

Denotamos por $T(n)$ o número de instruções executadas no pior caso para entradas de tamanho n

Análise assintótica e de pior caso

Consideramos apenas instâncias grandes

- o número de instruções normalmente cresce com o tamanho da entrada n
- instâncias com tamanho limitado por constante gastam tempo constante

Fazemos apenas análise de pior caso

- restringimos a entradas com um dado tamanho n
- consideramos apenas uma instância para a qual o algoritmo executa o maior número de instruções

Denotamos por $T(n)$ o número de instruções executadas no pior caso para entradas de tamanho n

Análise assintótica e de pior caso

Consideramos apenas instâncias grandes

- o número de instruções normalmente cresce com o tamanho da entrada n
- instâncias com tamanho limitado por constante gastam tempo constante

Fazemos apenas análise de pior caso

- restringimos a entradas com um dado tamanho n
- consideramos apenas uma instância para a qual o algoritmo executa o maior número de instruções

Denotamos por $T(n)$ o número de instruções executadas no pior caso para entradas de tamanho n

Características e limitações

Esse tipo de análise de complexidade:

- normalmente estima bem tempo de execução real
- permite comparar diversos algoritmos para um problema
- continua relevante mesmo com evoluções tecnológicas

Limitações:

- é uma análise pessimista do tempo de execução
- em certas aplicações, certas instâncias ocorrem mais frequentemente que um pior caso
- não fornece informação sobre tempo de execução médio

Características e limitações

Esse tipo de análise de complexidade:

- normalmente estima bem tempo de execução real
- permite comparar diversos algoritmos para um problema
- continua relevante mesmo com evoluções tecnológicas

Limitações:

- é uma análise pessimista do tempo de execução
- em certas aplicações, certas instâncias ocorrem mais frequentemente que um pior caso
- não fornece informação sobre tempo de execução médio

Características e limitações

Esse tipo de análise de complexidade:

- normalmente estima bem tempo de execução real
- permite comparar diversos algoritmos para um problema
- continua relevante mesmo com evoluções tecnológicas

Limitações:

- é uma análise pessimista do tempo de execução
- em certas aplicações, certas instâncias ocorrem mais frequentemente que um pior caso
- não fornece informação sobre tempo de execução médio

Características e limitações

Esse tipo de análise de complexidade:

- normalmente estima bem tempo de execução real
- permite comparar diversos algoritmos para um problema
- continua relevante mesmo com evoluções tecnológicas

Limitações:

- é uma análise pessimista do tempo de execução
- em certas aplicações, certas instâncias ocorrem mais frequentemente que um pior caso
- não fornece informação sobre tempo de execução médio

Características e limitações

Esse tipo de análise de complexidade:

- normalmente estima bem tempo de execução real
- permite comparar diversos algoritmos para um problema
- continua relevante mesmo com evoluções tecnológicas

Limitações:

- é uma análise pessimista do tempo de execução
- em certas aplicações, certas instâncias ocorrem mais frequentemente que um pior caso
- não fornece informação sobre tempo de execução médio

Características e limitações

Esse tipo de análise de complexidade:

- normalmente estima bem tempo de execução real
- permite comparar diversos algoritmos para um problema
- continua relevante mesmo com evoluções tecnológicas

Limitações:

- é uma análise pessimista do tempo de execução
- em certas aplicações, certas instâncias ocorrem mais frequentemente que um pior caso
- não fornece informação sobre tempo de execução médio

Características e limitações

Esse tipo de análise de complexidade:

- normalmente estima bem tempo de execução real
- permite comparar diversos algoritmos para um problema
- continua relevante mesmo com evoluções tecnológicas

Limitações:

- é uma análise pessimista do tempo de execução
- em certas aplicações, certas instâncias ocorrem mais frequentemente que um pior caso
- não fornece informação sobre tempo de execução médio

Características e limitações

Esse tipo de análise de complexidade:

- normalmente estima bem tempo de execução real
- permite comparar diversos algoritmos para um problema
- continua relevante mesmo com evoluções tecnológicas

Limitações:

- é uma análise pessimista do tempo de execução
- em certas aplicações, certas instâncias ocorrem mais frequentemente que um pior caso
- não fornece informação sobre tempo de execução médio

Começando a trabalhar

Ordenação

Problema: ordenar os elementos de um vetor

- Entrada: vetor $A[1 \ldots n]$
- Saída: rearranjo de $A[1 \ldots n]$ em ordem crescente

Vamos começar revendo a ordenação por inserção.

Ordenação

Problema: ordenar os elementos de um vetor

- Entrada: vetor $A[1 \ldots n]$
- Saída: rearranjo de $A[1 \ldots n]$ em ordem crescente

Vamos começar revendo a ordenação por inserção.

Ordenação

Problema: ordenar os elementos de um vetor

- Entrada: vetor $A[1 \ldots n]$
- Saída: rearranjo de $A[1 \ldots n]$ em ordem crescente

Vamos começar revendo a ordenação por inserção.

Ordenação

Problema: ordenar os elementos de um vetor

- Entrada: vetor $A[1 \ldots n]$
- Saída: rearranjo de $A[1 \ldots n]$ em ordem crescente

Vamos começar revendo a ordenação por inserção.

Inserção em um vetor ordenado

Ideia do algoritmo

- suponha que o subvetor $A[1 \ldots j-1]$ já está ordenado.
- vamos inserir o $A[j]$ para que $A[1 \ldots j]$ fique ordenado
- o valor do elemento a ser inserido é chamado de *chave*

Antes de inserir:

1						j				n
20	25	35	40	44	55	38	99	10	65	50

Após inserir:

1						j				n
20	25	35	38	40	44	55	99	10	65	50

Inserção em um vetor ordenado

Ideia do algoritmo

- suponha que o subvetor $A[1 \ldots j-1]$ já está ordenado.
- vamos inserir o $A[j]$ para que $A[1 \ldots j]$ fique ordenado
- o valor do elemento a ser inserido é chamado de *chave*

Antes de inserir:

1						j				n
20	25	35	40	44	55	38	99	10	65	50

Após inserir:

1						j				n
20	25	35	38	40	44	55	99	10	65	50

Inserção em um vetor ordenado

Ideia do algoritmo

- suponha que o subvetor $A[1 \ldots j-1]$ já está ordenado.
- vamos inserir o $A[j]$ para que $A[1 \ldots j]$ fique ordenado
- o valor do elemento a ser inserido é chamado de *chave*

Antes de inserir:

1						j				n
20	25	35	40	44	55	38	99	10	65	50

Após inserir:

1						j				n
20	25	35	38	40	44	55	99	10	65	50

Inserção em um vetor ordenado

Ideia do algoritmo

- suponha que o subvetor $A[1 \ldots j-1]$ já está ordenado.
- vamos inserir o $A[j]$ para que $A[1 \ldots j]$ fique ordenado
- o valor do elemento a ser inserido é chamado de *chave*

Antes de inserir:

1						j				n
20	25	35	40	44	55	38	99	10	65	50

Após inserir:

1						j				n
20	25	35	38	40	44	55	99	10	65	50

Inserção em um vetor ordenado

Ideia do algoritmo

- suponha que o subvetor $A[1 \ldots j-1]$ já está ordenado.
- vamos inserir o $A[j]$ para que $A[1 \ldots j]$ fique ordenado
- o valor do elemento a ser inserido é chamado de *chave*

Antes de inserir:

1						j				n
20	25	35	40	44	55	38	99	10	65	50

Após inserir:

1						j				n
20	25	35	38	40	44	55	99	10	65	50

Inserção em um vetor ordenado

Ideia do algoritmo

- suponha que o subvetor $A[1 \ldots j-1]$ já está ordenado.
- vamos inserir o $A[j]$ para que $A[1 \ldots j]$ fique ordenado
- o valor do elemento a ser inserido é chamado de *chave*

Antes de inserir:

1						j				n
20	25	35	40	44	55	38	99	10	65	50

Após inserir:

1						j				n
20	25	35	38	40	44	55	99	10	65	50

Inserindo uma chave

$chave = 38$

1					i	j				n
20	25	35	40	44	55	38	99	10	65	50

Inserindo uma chave

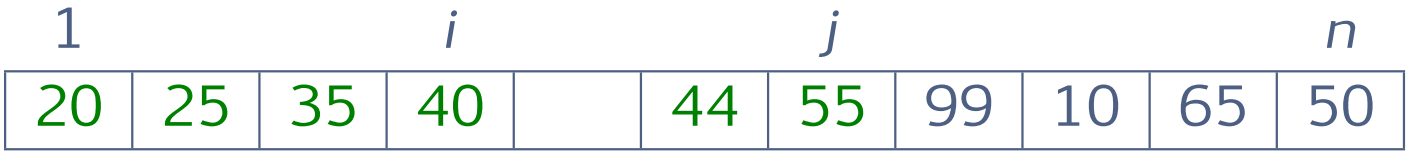
$chave = 38$

1					i	j				n
20	25	35	40	44	55	38	99	10	65	50

1				i		j				n
20	25	35	40	44		55	99	10	65	50

Inserindo uma chave

$chave = 38$



Inserindo uma chave

$chave = 38$

1					i	j				n
20	25	35	40	44	55	38	99	10	65	50

1				i		j				n
20	25	35	40	44		55	99	10	65	50

1			i			j				n
20	25	35	40		44	55	99	10	65	50

1		i				j				n
20	25	35		40	44	55	99	10	65	50

Inserindo uma chave

$chave = 38$

1					i	j				n
20	25	35	40	44	55	38	99	10	65	50

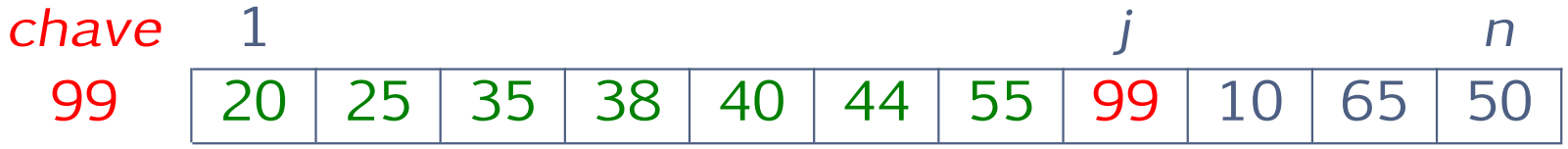
1				i		j				n
20	25	35	40	44		55	99	10	65	50

1			i			j				n
20	25	35	40		44	55	99	10	65	50

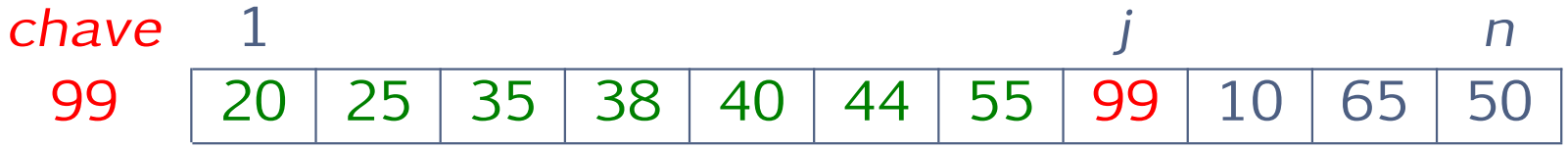
1		i				j				n
20	25	35		40	44	55	99	10	65	50

1		i				j				n
20	25	35	38	40	44	55	99	10	65	50

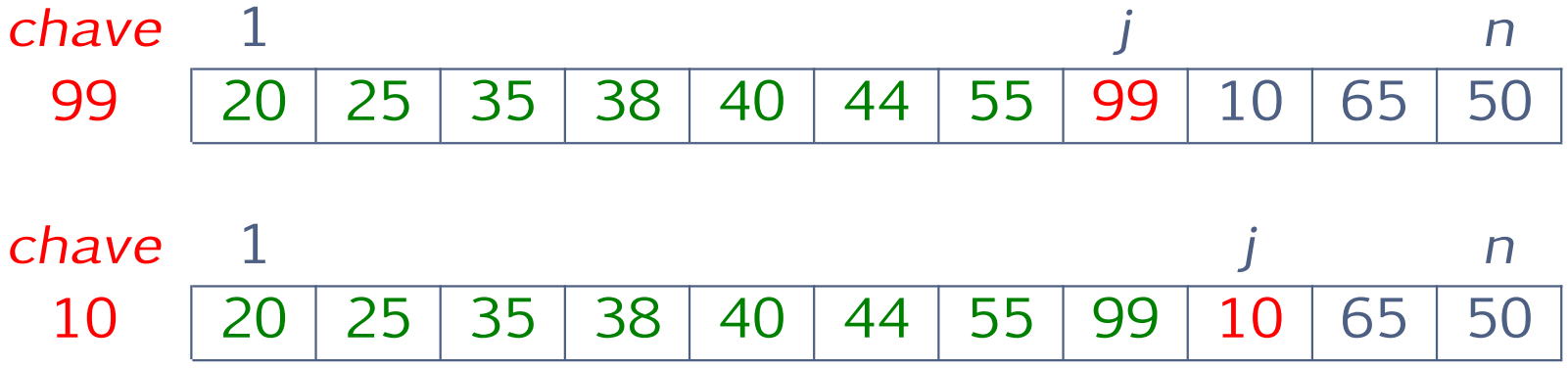
Ordenando por inserção



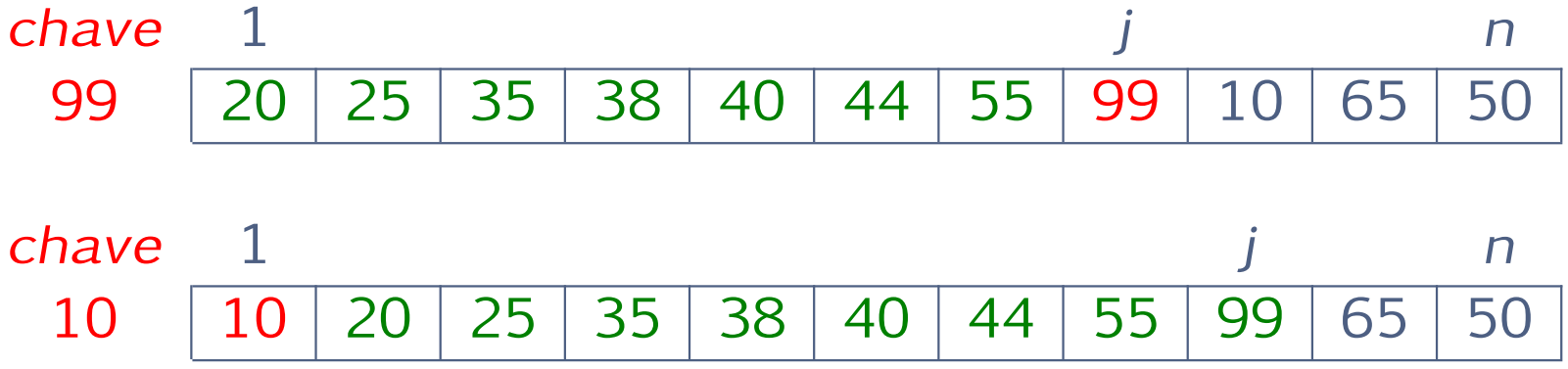
Ordenando por inserção



Ordenando por inserção



Ordenando por inserção



Ordenando por inserção

chave	1							*j*			*n*
99	20	25	35	38	40	44	55	99	10	65	50

chave	1								*j*		*n*
10	10	20	25	35	38	40	44	55	99	65	50

chave	1									*j*	*n*
65	10	20	25	35	38	40	44	55	99	65	50

Ordenando por inserção

	1							j		n
chave 99	20	25	35	38	40	44	55	99	10	65	50

	1								j	n
chave 10	10	20	25	35	38	40	44	55	99	65	50

	1									j	n
chave 65	10	20	25	35	38	40	44	55	65	99	50

Ordenando por inserção

chave	1							*j*			*n*
99	20	25	35	38	40	44	55	99	10	65	50

chave	1								*j*		*n*
10	10	20	25	35	38	40	44	55	99	65	50

chave	1									*j*	*n*
65	10	20	25	35	38	40	44	55	65	99	50

chave	1										*j*
50	10	20	25	35	38	40	44	55	65	99	50

Ordenando por inserção

chave	1							*j*			*n*
99	20	25	35	38	40	44	55	99	10	65	50

chave	1								*j*		*n*
10	10	20	25	35	38	40	44	55	99	65	50

chave	1									*j*	*n*
65	10	20	25	35	38	40	44	55	65	99	50

chave	1										*j*
50	10	20	25	35	38	40	44	50	55	65	99

Pseudocódigo de Insertion-Sort

```
INSERTION-SORT(A, n)
1   para j ← 2 até n faça
2       chave ← A[j]
3       i ← j − 1
4       enquanto i ≥ 1 e A[i] > chave faça
5           A[i + 1] ← A[i]
6           i ← i − 1
7       A[i + 1] ← chave
```

Análise do algoritmo

O que é importante analisar?

- Correção
  - já testamos o algoritmo com um exemplo
  - por enquanto, suponha que o algoritmo está correto
  - vamos mostrar que ele está correto depois
- Complexidade de tempo
  - considere vetores com n elementos
  - quantas instruções são executas?

Análise do algoritmo

O que é importante analisar?

- Correção
  - já testamos o algoritmo com um exemplo
  - por enquanto, suponha que o algoritmo está correto
  - vamos mostrar que ele está correto depois

- Complexidade de tempo
  - considere vetores com n elementos
  - quantas instruções são executas?

Análise do algoritmo

O que é importante analisar?

- Correção
  - já testamos o algoritmo com um exemplo
  - por enquanto, suponha que o algoritmo está correto
  - vamos mostrar que ele está correto depois

- Complexidade de tempo
  - considere vetores com n elementos
  - quantas instruções são executas?

Análise do algoritmo

O que é importante analisar?

- Correção
  - já testamos o algoritmo com um exemplo
  - por enquanto, suponha que o algoritmo está correto
  - vamos mostrar que ele está correto depois

- Complexidade de tempo
  - considere vetores com n elementos
  - quantas instruções são executas?

Análise do algoritmo

O que é importante analisar?

- Correção
  - já testamos o algoritmo com um exemplo
  - por enquanto, suponha que o algoritmo está correto
  - vamos mostrar que ele está correto depois

- Complexidade de tempo
  - considere vetores com n elementos
  - quantas instruções são executas?

Análise do algoritmo

O que é importante analisar?

- Correção
  - já testamos o algoritmo com um exemplo
  - por enquanto, suponha que o algoritmo está correto
  - vamos mostrar que ele está correto depois

- Complexidade de tempo
  - considere vetores com n elementos
  - quantas instruções são executas?

Análise do algoritmo

O que é importante analisar?

- Correção
  - já testamos o algoritmo com um exemplo
  - por enquanto, suponha que o algoritmo está correto
  - vamos mostrar que ele está correto depois

- Complexidade de tempo
  - considere vetores com n elementos
  - quantas instruções são executas?

Análise do algoritmo

O que é importante analisar?

- Correção
  - já testamos o algoritmo com um exemplo
  - por enquanto, suponha que o algoritmo está correto
  - vamos mostrar que ele está correto depois

- Complexidade de tempo
  - considere vetores com n elementos
  - quantas instruções são executas?

Contando o número de instruções

INSERTION-SORT(A, n)	Custo	Qnts vezes?
1 **para** $j \leftarrow 2$ **até** n **faça**	?	?
2 $\quad chave \leftarrow A[j]$	?	?
3 $\quad i \leftarrow j - 1$	?	?
4 $\quad$ **enquanto** $i \geq 1$ **e** $A[i] > chave$ **faça**	?	?
5 $\quad\quad A[i+1] \leftarrow A[i]$	?	?
6 $\quad\quad i \leftarrow i - 1$	?	?
7 $\quad A[i+1] \leftarrow chave$	?	?

- a linha k executa um número constante de instruções c_k
- cada linha executa uma ou mais vezes
- quantas vezes a linha 4 executa depende da entrada
  - seja t_j quantas vezes enquanto executa para um certo j

Contando o número de instruções

INSERTION-SORT(A, n)	Custo	Qnts vezes?
1 **para** $j \leftarrow 2$ **até** n **faça**	c_1	?
2 $\quad$ $chave \leftarrow A[j]$	c_2	?
3 $\quad$ $i \leftarrow j - 1$	c_3	?
4 $\quad$ **enquanto** $i \geq 1$ **e** $A[i] > chave$ **faça**	c_4	?
5 $\quad\quad$ $A[i+1] \leftarrow A[i]$	c_5	?
6 $\quad\quad$ $i \leftarrow i - 1$	c_6	?
7 $\quad$ $A[i+1] \leftarrow chave$	c_7	?

- a linha k executa um número constante de instruções c_k
- cada linha executa uma ou mais vezes
- quantas vezes a linha 4 executa depende da entrada
  - seja t_j quantas vezes enquanto executa para um certo j

Contando o número de instruções

INSERTION-SORT(A, n)	Custo	Qnts vezes?
1 **para** $j \leftarrow 2$ **até** n **faça**	c_1	?
2 $\quad$ $chave \leftarrow A[j]$	c_2	?
3 $\quad$ $i \leftarrow j - 1$	c_3	?
4 $\quad$ **enquanto** $i \geq 1$ **e** $A[i] > chave$ **faça**	c_4	?
5 $\quad\quad$ $A[i+1] \leftarrow A[i]$	c_5	?
6 $\quad\quad$ $i \leftarrow i - 1$	c_6	?
7 $\quad$ $A[i+1] \leftarrow chave$	c_7	?

- a linha k executa um número constante de instruções c_k
- cada linha executa uma ou mais vezes
- quantas vezes a linha 4 executa depende da entrada
  - seja t_j quantas vezes enquanto executa para um certo j

Contando o número de instruções

INSERTION-SORT(A, n)	Custo	Qnts vezes?
1 **para** $j \leftarrow 2$ **até** n **faça**	c_1	n
2 $\quad chave \leftarrow A[j]$	c_2	?
3 $\quad i \leftarrow j-1$	c_3	?
4 $\quad$ **enquanto** $i \geq 1$ **e** $A[i] > chave$ **faça**	c_4	?
5 $\quad\quad A[i+1] \leftarrow A[i]$	c_5	?
6 $\quad\quad i \leftarrow i-1$	c_6	?
7 $\quad A[i+1] \leftarrow chave$	c_7	?

- a linha k executa um número constante de instruções c_k
- cada linha executa uma ou mais vezes
- quantas vezes a linha 4 executa depende da entrada
  - seja t_j quantas vezes enquanto executa para um certo j

Contando o número de instruções

INSERTION-SORT(A, n)	Custo	Qnts vezes?
1 **para** $j \leftarrow 2$ **até** n **faça**	c_1	n
2 $\quad chave \leftarrow A[j]$	c_2	$n-1$
3 $\quad i \leftarrow j-1$	c_3	?
4 $\quad$ **enquanto** $i \geq 1$ **e** $A[i] > chave$ **faça**	c_4	?
5 $\quad\quad A[i+1] \leftarrow A[i]$	c_5	?
6 $\quad\quad i \leftarrow i-1$	c_6	?
7 $\quad A[i+1] \leftarrow chave$	c_7	?

- a linha k executa um número constante de instruções c_k
- cada linha executa uma ou mais vezes
- quantas vezes a linha 4 executa depende da entrada
  - seja t_j quantas vezes enquanto executa para um certo j

Contando o número de instruções

INSERTION-SORT(A, n)	Custo	Qnts vezes?
1 **para** $j \leftarrow 2$ **até** n **faça**	c_1	n
2 $\quad chave \leftarrow A[j]$	c_2	$n-1$
3 $\quad i \leftarrow j-1$	c_3	$n-1$
4 $\quad$ **enquanto** $i \geq 1$ **e** $A[i] > chave$ **faça**	c_4	?
5 $\quad\quad A[i+1] \leftarrow A[i]$	c_5	?
6 $\quad\quad i \leftarrow i-1$	c_6	?
7 $\quad A[i+1] \leftarrow chave$	c_7	?

- a linha k executa um número constante de instruções c_k
- cada linha executa uma ou mais vezes
- quantas vezes a linha 4 executa depende da entrada
  - seja t_j quantas vezes enquanto executa para um certo j

Contando o número de instruções

INSERTION-SORT(A, n)	Custo	Qnts vezes?
1 **para** $j \leftarrow 2$ **até** n **faça**	c_1	n
2 $\quad chave \leftarrow A[j]$	c_2	$n-1$
3 $\quad i \leftarrow j-1$	c_3	$n-1$
4 $\quad$ **enquanto** $i \geq 1$ **e** $A[i] > chave$ **faça**	c_4	?
5 $\quad\quad A[i+1] \leftarrow A[i]$	c_5	?
6 $\quad\quad i \leftarrow i-1$	c_6	?
7 $\quad A[i+1] \leftarrow chave$	c_7	?

- a linha k executa um número constante de instruções c_k
- cada linha executa uma ou mais vezes
- quantas vezes a linha 4 executa depende da entrada
  - seja t_j quantas vezes **enquanto** executa para um certo j

Contando o número de instruções

INSERTION-SORT(A, n)	Custo	Qnts vezes?
1 **para** $j \leftarrow 2$ **até** n **faça**	c_1	n
2 $\quad chave \leftarrow A[j]$	c_2	$n-1$
3 $\quad i \leftarrow j-1$	c_3	$n-1$
4 $\quad$ **enquanto** $i \geq 1$ **e** $A[i] > chave$ **faça**	c_4	$\sum_{j=2}^{n} t_j$
5 $\quad\quad A[i+1] \leftarrow A[i]$	c_5	?
6 $\quad\quad i \leftarrow i-1$	c_6	?
7 $\quad A[i+1] \leftarrow chave$	c_7	?

- a linha k executa um número constante de instruções c_k
- cada linha executa uma ou mais vezes
- quantas vezes a linha 4 executa depende da entrada
  - seja t_j quantas vezes **enquanto** executa para um certo j

Contando o número de instruções

INSERTION-SORT(A, n)	Custo	Qnts vezes?
1 **para** $j \leftarrow 2$ **até** n **faça**	c_1	n
2 $\quad$ *chave* $\leftarrow A[j]$	c_2	$n-1$
3 $\quad$ $i \leftarrow j-1$	c_3	$n-1$
4 $\quad$ **enquanto** $i \geq 1$ **e** $A[i] >$ *chave* **faça**	c_4	$\sum_{j=2}^{n} t_j$
5 $\quad\quad$ $A[i+1] \leftarrow A[i]$	c_5	$\sum_{j=2}^{n}(t_j-1)$
6 $\quad\quad$ $i \leftarrow i-1$	c_6	?
7 $\quad$ $A[i+1] \leftarrow$ *chave*	c_7	?

- a linha k executa um número constante de instruções c_k
- cada linha executa uma ou mais vezes
- quantas vezes a linha 4 executa depende da entrada
  - seja t_j quantas vezes **enquanto** executa para um certo j

Contando o número de instruções

INSERTION-SORT(A, n)	Custo	Qnts vezes?
1 **para** $j \leftarrow 2$ **até** n **faça**	c_1	n
2 $\quad$ *chave* $\leftarrow A[j]$	c_2	$n-1$
3 $\quad$ $i \leftarrow j-1$	c_3	$n-1$
4 $\quad$ **enquanto** $i \geq 1$ **e** $A[i] >$ *chave* **faça**	c_4	$\sum_{j=2}^{n} t_j$
5 $\quad\quad$ $A[i+1] \leftarrow A[i]$	c_5	$\sum_{j=2}^{n}(t_j-1)$
6 $\quad\quad$ $i \leftarrow i-1$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7 $\quad$ $A[i+1] \leftarrow$ *chave*	c_7	?

- a linha k executa um número constante de instruções c_k
- cada linha executa uma ou mais vezes
- quantas vezes a linha 4 executa depende da entrada
  - seja t_j quantas vezes **enquanto** executa para um certo j

Contando o número de instruções

INSERTION-SORT(A, n)	Custo	Qnts vezes?
1 **para** $j \leftarrow 2$ **até** n **faça**	c_1	n
2 $\quad chave \leftarrow A[j]$	c_2	$n-1$
3 $\quad i \leftarrow j-1$	c_3	$n-1$
4 $\quad$ **enquanto** $i \geq 1$ **e** $A[i] > chave$ **faça**	c_4	$\sum_{j=2}^{n} t_j$
5 $\quad\quad A[i+1] \leftarrow A[i]$	c_5	$\sum_{j=2}^{n}(t_j-1)$
6 $\quad\quad i \leftarrow i-1$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7 $\quad A[i+1] \leftarrow chave$	c_7	$n-1$

- a linha k executa um número constante de instruções c_k
- cada linha executa uma ou mais vezes
- quantas vezes a linha 4 executa depende da entrada
  - seja t_j quantas vezes **enquanto** executa para um certo j

Tempo de execução total

- Considere uma instância de tamanho n
- Seja $T(n)$ o número de instruções executadas para ela
- Basta somar para todas as linhas

$$T(n) = c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot \sum_{j=2}^{n} t_j + c_5 \cdot \sum_{j=2}^{n} (t_j - 1) + c_6 \cdot \sum_{j=2}^{n} (t_j - 1) + c_7 \cdot (n-1)$$

Observações:

- entradas do mesmo tamanho têm tempos diferentes
- vamos considerar diferentes instâncias
  - melhor caso: quando $T(n)$ é o menor possível
  - pior caso: quando $T(n)$ é o maior possível

Tempo de execução total

- Considere uma instância de tamanho n
- Seja $T(n)$ o número de instruções executadas para ela
- Basta somar para todas as linhas

$$T(n) = c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot \sum_{j=2}^{n} t_j + c_5 \cdot \sum_{j=2}^{n} (t_j - 1) + c_6 \cdot \sum_{j=2}^{n} (t_j - 1) + c_7 \cdot (n-1)$$

Observações:

- entradas do mesmo tamanho têm tempos diferentes
- vamos considerar diferentes instâncias
  - melhor caso: quando $T(n)$ é o menor possível
  - pior caso: quando $T(n)$ é o maior possível

Tempo de execução total

- Considere uma instância de tamanho n
- Seja $T(n)$ o número de instruções executadas para ela
- Basta somar para todas as linhas

$$T(n) = c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot \sum_{j=2}^{n} t_j + c_5 \cdot \sum_{j=2}^{n} (t_j - 1) + c_6 \cdot \sum_{j=2}^{n} (t_j - 1) + c_7 \cdot (n-1)$$

Observações:

- entradas do mesmo tamanho têm tempos diferentes
- vamos considerar diferentes instâncias
  - melhor caso: quando $T(n)$ é o menor possível
  - pior caso: quando $T(n)$ é o maior possível

Tempo de execução total

- Considere uma instância de tamanho n
- Seja $T(n)$ o número de instruções executadas para ela
- Basta somar para todas as linhas

$$T(n) = c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot \sum_{j=2}^{n} t_j + c_5 \cdot \sum_{j=2}^{n} (t_j - 1) + c_6 \cdot \sum_{j=2}^{n} (t_j - 1) + c_7 \cdot (n-1)$$

Observações:

- entradas do mesmo tamanho têm tempos diferentes
- vamos considerar diferentes instâncias
  - melhor caso: quando $T(n)$ é o menor possível
  - pior caso: quando $T(n)$ é o maior possível

Tempo de execução total

- Considere uma instância de tamanho n
- Seja $T(n)$ o número de instruções executadas para ela
- Basta somar para todas as linhas

$$T(n) = c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot \sum_{j=2}^{n} t_j + c_5 \cdot \sum_{j=2}^{n} (t_j - 1) + c_6 \cdot \sum_{j=2}^{n} (t_j - 1) + c_7 \cdot (n-1)$$

Observações:

- entradas do mesmo tamanho têm tempos diferentes
- vamos considerar diferentes instâncias
  - melhor caso: quando $T(n)$ é o menor possível
  - pior caso: quando $T(n)$ é o maior possível

Tempo de execução total

- Considere uma instância de tamanho n
- Seja $T(n)$ o número de instruções executadas para ela
- Basta somar para todas as linhas

$$T(n) = c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot \sum_{j=2}^{n} t_j + c_5 \cdot \sum_{j=2}^{n} (t_j - 1) + c_6 \cdot \sum_{j=2}^{n} (t_j - 1) + c_7 \cdot (n-1)$$

Observações:

- entradas do mesmo tamanho têm tempos diferentes
- vamos considerar diferentes instâncias
  - melhor caso: quando $T(n)$ é o menor possível
  - pior caso: quando $T(n)$ é o maior possível

Tempo de execução total

- Considere uma instância de tamanho n
- Seja $T(n)$ o número de instruções executadas para ela
- Basta somar para todas as linhas

$$T(n) = c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot \sum_{j=2}^{n} t_j + c_5 \cdot \sum_{j=2}^{n} (t_j - 1) + c_6 \cdot \sum_{j=2}^{n} (t_j - 1) + c_7 \cdot (n-1)$$

Observações:

- entradas do mesmo tamanho têm tempos diferentes
- vamos considerar diferentes instâncias
  - melhor caso: quando $T(n)$ é o menor possível
  - pior caso: quando $T(n)$ é o maior possível

Tempo de execução total

- Considere uma instância de tamanho n
- Seja $T(n)$ o número de instruções executadas para ela
- Basta somar para todas as linhas

$$T(n) = c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot \sum_{j=2}^{n} t_j + c_5 \cdot \sum_{j=2}^{n} (t_j - 1) + c_6 \cdot \sum_{j=2}^{n} (t_j - 1) + c_7 \cdot (n-1)$$

Observações:

- entradas do mesmo tamanho têm tempos diferentes
- vamos considerar diferentes instâncias
  - melhor caso: quando $T(n)$ é o menor possível
  - pior caso: quando $T(n)$ é o maior possível

Melhor caso

Um melhor caso ocorre quando $t_j = 1$ para cada j

- basta que a condição do **enquanto** sempre falhe
- ocorre se a entrada A já vem ordenada

Nesse caso:

$$\begin{aligned} T(n) &= c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot (n-1) + c_7 \cdot (n-1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_7) \cdot n - (c_2 + c_3 + c_4 + c_7) \\ &= a \cdot n + b \end{aligned}$$

- os valores de a e b são constantes
- o tempo de execução no melhor caso é linear em n

Melhor caso

Um melhor caso ocorre quando $t_j = 1$ para cada j

- basta que a condição do **enquanto** sempre falhe
- ocorre se a entrada A já vem ordenada

Nesse caso:

$$\begin{aligned} T(n) &= c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot (n-1) + c_7 \cdot (n-1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_7) \cdot n - (c_2 + c_3 + c_4 + c_7) \\ &= a \cdot n + b \end{aligned}$$

- os valores de a e b são constantes
- o tempo de execução no melhor caso é linear em n

Melhor caso

Um melhor caso ocorre quando $t_j = 1$ para cada j

- basta que a condição do **enquanto** sempre falhe
- ocorre se a entrada A já vem ordenada

Nesse caso:

$$\begin{aligned} T(n) &= c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot (n-1) + c_7 \cdot (n-1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_7) \cdot n - (c_2 + c_3 + c_4 + c_7) \\ &= a \cdot n + b \end{aligned}$$

- os valores de a e b são constantes
- o tempo de execução no melhor caso é linear em n

Melhor caso

Um melhor caso ocorre quando $t_j = 1$ para cada j

- basta que a condição do **enquanto** sempre falhe
- ocorre se a entrada A já vem ordenada

Nesse caso:

$$\begin{aligned} T(n) &= c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot (n-1) + c_7 \cdot (n-1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_7) \cdot n - (c_2 + c_3 + c_4 + c_7) \\ &= a \cdot n + b \end{aligned}$$

- os valores de a e b são constantes
- o tempo de execução no melhor caso é linear em n

Melhor caso

Um melhor caso ocorre quando $t_j = 1$ para cada j

- basta que a condição do **enquanto** sempre falhe
- ocorre se a entrada A já vem ordenada

Nesse caso:

$$\begin{aligned} T(n) &= c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot (n-1) + c_7 \cdot (n-1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_7) \cdot n - (c_2 + c_3 + c_4 + c_7) \\ &= a \cdot n + b \end{aligned}$$

- os valores de a e b são constantes
- o tempo de execução no melhor caso é linear em n

Melhor caso

Um melhor caso ocorre quando $t_j = 1$ para cada j

- basta que a condição do **enquanto** sempre falhe
- ocorre se a entrada A já vem ordenada

Nesse caso:

$$\begin{aligned} T(n) &= c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot (n-1) + c_7 \cdot (n-1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_7) \cdot n - (c_2 + c_3 + c_4 + c_7) \\ &= a \cdot n + b \end{aligned}$$

- os valores de a e b são constantes
- o tempo de execução no melhor caso é linear em n

Melhor caso

Um melhor caso ocorre quando $t_j = 1$ para cada j

- basta que a condição do **enquanto** sempre falhe
- ocorre se a entrada A já vem ordenada

Nesse caso:

$$\begin{aligned} T(n) &= c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot (n-1) + c_7 \cdot (n-1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_7) \cdot n - (c_2 + c_3 + c_4 + c_7) \\ &= a \cdot n + b \end{aligned}$$

- os valores de a e b são constantes
- o tempo de execução no melhor caso é linear em n

Pior caso

Um pior caso ocorre quando t_j é máximo para cada j

- basta que a condição do **enquanto** só falha quando $i = 0$
- nessa situação, teremos $t_j = j$
- ocorre se a entrada A vem ordenada decrescentemente

Relembre que

- $\sum_{j=2}^{n} j = n(n+1)/2 - 1$ e $\sum_{j=2}^{n} (j-1) = n(n-1)/2$

Pior caso

Um pior caso ocorre quando t_j é máximo para cada j

- basta que a condição do **enquanto** só falha quando $i = 0$
- nessa situação, teremos $t_j = j$
- ocorre se a entrada A vem ordenada decrescentemente

Relembre que

- $\sum_{j=2}^{n} j = n(n+1)/2 - 1$ e $\sum_{j=2}^{n} (j-1) = n(n-1)/2$

Pior caso

Um pior caso ocorre quando t_j é máximo para cada j

- basta que a condição do **enquanto** só falha quando $i = 0$
- nessa situação, teremos $t_j = j$
- ocorre se a entrada A vem ordenada decrescentemente

Relembre que

- $\sum_{j=2}^{n} j = n(n+1)/2 - 1$ e $\sum_{j=2}^{n} (j-1) = n(n-1)/2$

Pior caso

Um pior caso ocorre quando t_j é máximo para cada j

- basta que a condição do **enquanto** só falha quando $i = 0$
- nessa situação, teremos $t_j = j$
- ocorre se a entrada A vem ordenada decrescentemente

Relembre que

- $\sum_{j=2}^{n} j = n(n+1)/2 - 1$ e $\sum_{j=2}^{n} (j-1) = n(n-1)/2$

Pior caso

Um pior caso ocorre quando t_j é máximo para cada j

- basta que a condição do **enquanto** só falha quando $i = 0$
- nessa situação, teremos $t_j = j$
- ocorre se a entrada A vem ordenada decrescentemente

Relembre que

- $\sum_{j=2}^{n} j = n(n+1)/2 - 1$ e $\sum_{j=2}^{n} (j-1) = n(n-1)/2$

Pior caso

Um pior caso ocorre quando t_j é máximo para cada j

- basta que a condição do **enquanto** só falha quando $i = 0$
- nessa situação, teremos $t_j = j$
- ocorre se a entrada A vem ordenada decrescentemente

Relembre que

- $\sum_{j=2}^{n} j = n(n+1)/2 - 1$ e $\sum_{j=2}^{n} (j-1) = n(n-1)/2$

Pior caso (cont)

Substituindo, temos

$$\begin{aligned} T(n) &= c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot (n(n+1)/2 - 1) + \\ &\quad c_5 \cdot n(n-1)/2 + c_6 \cdot n(n-1)/2 + c_7 \cdot (n-1) \\ &= (c_4/2 + c_5/2 + c_6/2) \cdot n^2 + \\ &\quad (c_1 + c_2 + c_3 + c_4/2 - c_5/2 - c_6/2 + c_7) \cdot n - \\ &\quad (c_2 + c_3 + c_4 + c_7) \\ &= a \cdot n^2 + b \cdot n + c \end{aligned}$$

- os valores de a, b, c são constantes
- o tempo de execução no pior caso é quadrático em n

Complexidade assintótica

Estamos interessados principalmente

- na análise de pior caso
- no tempo de execução para instâncias grandes

Comportamento assintótico

- no pior caso temos $T(n) = an^2 + bn + c$
  - o termo dominante é o que contém n^2
  - o tempo de execução é uma função quadrática
  - as constantes a, b, c só dependem da implementação
- não nos preocupamos com os valores de a, b, c

Por que isso é razoável?

Complexidade assintótica

Estamos interessados principalmente

- na análise de pior caso
- no tempo de execução para instâncias grandes

Comportamento assintótico

- no pior caso temos $T(n) = an^2 + bn + c$
  - o termo dominante é o que contém n^2
  - o tempo de execução é uma função quadrática
  - as constantes a, b, c só dependem da implementação
- não nos preocupamos com os valores de a, b, c

Por que isso é razoável?

Complexidade assintótica

Estamos interessados principalmente

- na análise de pior caso
- no tempo de execução para instâncias grandes

Comportamento assintótico

- no pior caso temos $T(n) = an^2 + bn + c$
  - o termo dominante é o que contém n^2
  - o tempo de execução é uma função quadrática
  - as constantes a, b, c só dependem da implementação
- não nos preocupamos com os valores de a, b, c

Por que isso é razoável?

Complexidade assintótica

Estamos interessados principalmente

- na análise de pior caso
- no tempo de execução para instâncias grandes

Comportamento assintótico

- no pior caso temos $T(n) = an^2 + bn + c$
  - o termo dominante é o que contém n^2
  - o tempo de execução é uma função quadrática
  - as constantes a, b, c só dependem da implementação
- não nos preocupamos com os valores de a, b, c

Por que isso é razoável?

Complexidade assintótica

Estamos interessados principalmente

- na análise de pior caso
- no tempo de execução para instâncias grandes

Comportamento assintótico

- no pior caso temos $T(n) = an^2 + bn + c$
  - o termo dominante é o que contém n^2
  - o tempo de execução é uma função quadrática
  - as constantes a, b, c só dependem da implementação
- não nos preocupamos com os valores de a, b, c

Por que isso é razoável?

Complexidade assintótica

Estamos interessados principalmente

- na análise de pior caso
- no tempo de execução para instâncias grandes

Comportamento assintótico

- no pior caso temos $T(n) = an^2 + bn + c$
  - o termo dominante é o que contém n^2
  - o tempo de execução é uma função quadrática
  - as constantes a, b, c só dependem da implementação
- não nos preocupamos com os valores de a, b, c

Por que isso é razoável?

Complexidade assintótica

Estamos interessados principalmente

- na análise de pior caso
- no tempo de execução para instâncias grandes

Comportamento assintótico

- no pior caso temos $T(n) = an^2 + bn + c$
  - o termo dominante é o que contém n^2
  - o tempo de execução é uma função quadrática
  - as constantes a, b, c só dependem da implementação
- não nos preocupamos com os valores de a, b, c

Por que isso é razoável?

Complexidade assintótica

Estamos interessados principalmente

- na análise de pior caso
- no tempo de execução para instâncias grandes

Comportamento assintótico

- no pior caso temos $T(n) = an^2 + bn + c$
  - o termo dominante é o que contém n^2
  - o tempo de execução é uma função quadrática
  - as constantes a, b, c só dependem da implementação
- não nos preocupamos com os valores de a, b, c

Por que isso é razoável?

Complexidade assintótica

Estamos interessados principalmente

- na análise de pior caso
- no tempo de execução para instâncias grandes

Comportamento assintótico

- no pior caso temos $T(n) = an^2 + bn + c$
  - o termo dominante é o que contém n^2
  - o tempo de execução é uma função quadrática
  - as constantes a, b, c só dependem da implementação
- não nos preocupamos com os valores de a, b, c

Por que isso é razoável?

Complexidade assintótica

Estamos interessados principalmente

- na análise de pior caso
- no tempo de execução para instâncias grandes

Comportamento assintótico

- no pior caso temos $T(n) = an^2 + bn + c$
  - o termo dominante é o que contém n^2
  - o tempo de execução é uma função quadrática
  - as constantes a, b, c só dependem da implementação
- não nos preocupamos com os valores de a, b, c

Por que isso é razoável?

Um exemplo de função quadrática

Considere a função $3n^2 + 10n + 50$

n	$3n^2 + 10n + 50$	$3n^2$
64	12978	12288
128	50482	49152
512	791602	786432
1024	3156018	3145728
2048	12603442	12582912
4096	50372658	50331648
8192	201408562	201326592
16384	805470258	805306368
32768	3221553202	3221225472

- quando n é grande, o termo $3n^2$ é uma boa estimativa
- podemos nos concentrar nos termos dominantes

Um exemplo de função quadrática

Considere a função $3n^2 + 10n + 50$

n	$3n^2 + 10n + 50$	$3n^2$
64	12978	12288
128	50482	49152
512	791602	786432
1024	3156018	3145728
2048	12603442	12582912
4096	50372658	50331648
8192	201408562	201326592
16384	805470258	805306368
32768	3221553202	3221225472

- quando n é grande, o termo $3n^2$ é uma boa estimativa
- podemos nos concentrar nos termos dominantes

Um exemplo de função quadrática

Considere a função $3n^2 + 10n + 50$

n	$3n^2 + 10n + 50$	$3n^2$
64	12978	12288
128	50482	49152
512	791602	786432
1024	3156018	3145728
2048	12603442	12582912
4096	50372658	50331648
8192	201408562	201326592
16384	805470258	805306368
32768	3221553202	3221225472

- quando n é grande, o termo $3n^2$ é uma boa estimativa
- podemos nos concentrar nos termos dominantes

Um exemplo de função quadrática

Considere a função $3n^2 + 10n + 50$

n	$3n^2 + 10n + 50$	$3n^2$
64	12978	12288
128	50482	49152
512	791602	786432
1024	3156018	3145728
2048	12603442	12582912
4096	50372658	50331648
8192	201408562	201326592
16384	805470258	805306368
32768	3221553202	3221225472

- quando n é grande, o termo $3n^2$ é uma boa estimativa
- podemos nos concentrar nos termos dominantes

Notação assintótica

Como simplificar o tempo de pior caso de INSERTION-SORT?

- ao invés de escrever $T(n) = an^2 + bn + c$,
- escrevemos somente $T(n) = \Theta(n^2)$

Essa notação significa que, para n suficientemente grande,

1. $T(n)$ é limitada superiormente por $c \cdot n^2$, para algum $c > 0$
2. $T(n)$ é limitada inferiormente por $d \cdot n^2$, para algum $d > 0$

Vamos formalizar essa notação assintótica mais adiante!

Notação assintótica

Como simplificar o tempo de pior caso de INSERTION-SORT?

- ao invés de escrever $T(n) = an^2 + bn + c$,
- escrevemos somente $T(n) = \Theta(n^2)$

Essa notação significa que, para n suficientemente grande,

1. $T(n)$ é limitada superiormente por $c \cdot n^2$, para algum $c > 0$
2. $T(n)$ é limitada inferiormente por $d \cdot n^2$, para algum $d > 0$

Vamos formalizar essa notação assintótica mais adiante!

Notação assintótica

Como simplificar o tempo de pior caso de INSERTION-SORT?

- ao invés de escrever $T(n) = an^2 + bn + c$,
- escrevemos somente $T(n) = \Theta(n^2)$

Essa notação significa que, para n suficientemente grande,

1. $T(n)$ é limitada superiormente por $c \cdot n^2$, para algum $c > 0$
2. $T(n)$ é limitada inferiormente por $d \cdot n^2$, para algum $d > 0$

Vamos formalizar essa notação assintótica mais adiante!

Notação assintótica

Como simplificar o tempo de pior caso de INSERTION-SORT?

- ao invés de escrever $T(n) = an^2 + bn + c$,
- escrevemos somente $T(n) = \Theta(n^2)$

Essa notação significa que, para n suficientemente grande,

1. $T(n)$ é limitada superiormente por $c \cdot n^2$, para algum $c > 0$
2. $T(n)$ é limitada inferiormente por $d \cdot n^2$, para algum $d > 0$

Vamos formalizar essa notação assintótica mais adiante!

Notação assintótica

Como simplificar o tempo de pior caso de INSERTION-SORT?

- ao invés de escrever $T(n) = an^2 + bn + c$,
- escrevemos somente $T(n) = \Theta(n^2)$

Essa notação significa que, para n suficientemente grande,

1. $T(n)$ é limitada superiormente por $c \cdot n^2$, para algum $c > 0$
2. $T(n)$ é limitada inferiormente por $d \cdot n^2$, para algum $d > 0$

Vamos formalizar essa notação assintótica mais adiante!

Notação assintótica

Como simplificar o tempo de pior caso de INSERTION-SORT?

- ao invés de escrever $T(n) = an^2 + bn + c$,
- escrevemos somente $T(n) = \Theta(n^2)$

Essa notação significa que, para n suficientemente grande,

1. $T(n)$ é limitada superiormente por $c \cdot n^2$, para algum $c > 0$
2. $T(n)$ é limitada inferiormente por $d \cdot n^2$, para algum $d > 0$

Vamos formalizar essa notação assintótica mais adiante!

Notação assintótica

Como simplificar o tempo de pior caso de INSERTION-SORT?

- ao invés de escrever $T(n) = an^2 + bn + c$,
- escrevemos somente $T(n) = \Theta(n^2)$

Essa notação significa que, para n suficientemente grande,

1. $T(n)$ é limitada superiormente por $c \cdot n^2$, para algum $c > 0$
2. $T(n)$ é limitada inferiormente por $d \cdot n^2$, para algum $d > 0$

Vamos formalizar essa notação assintótica mais adiante!

Projetando algoritmos

Projetando algoritmos

- Divisão e conquista

Entendendo e melhorando

Até agora:

- ordenamos incrementalmente com o Insertion-Sort
- vimos que sua complexidade de pior caso é $\Theta(n^2)$

Vamos estudar uma maneira alternativa de ordenar números

- vamos utilizar uma técnica recursiva chamada de divisão e conquista
- muitas vezes, obtemos algoritmos mais rápidos do que os incrementais

Entendendo e melhorando

Até agora:

- ordenamos incrementalmente com o Insertion-Sort
- vimos que sua complexidade de pior caso é $\Theta(n^2)$

Vamos estudar uma maneira alternativa de ordenar números

- vamos utilizar uma técnica recursiva chamada de divisão e conquista
- muitas vezes, obtemos algoritmos mais rápidos do que os incrementais

Entendendo e melhorando

Até agora:

- ordenamos incrementalmente com o Insertion-Sort
- vimos que sua complexidade de pior caso é $\Theta(n^2)$

Vamos estudar uma maneira alternativa de ordenar números

- vamos utilizar uma técnica recursiva chamada de divisão e conquista
- muitas vezes, obtemos algoritmos mais rápidos do que os incrementais

Entendendo e melhorando

Até agora:

- ordenamos incrementalmente com o Insertion-Sort
- vimos que sua complexidade de pior caso é $\Theta(n^2)$

Vamos estudar uma maneira alternativa de ordenar números

- vamos utilizar uma técnica recursiva chamada de divisão e conquista
- muitas vezes, obtemos algoritmos mais rápidos do que os incrementais

Entendendo e melhorando

Até agora:

- ordenamos incrementalmente com o INSERTION-SORT
- vimos que sua complexidade de pior caso é $\Theta(n^2)$

Vamos estudar uma maneira alternativa de ordenar números

- vamos utilizar uma técnica recursiva chamada de divisão e conquista
- muitas vezes, obtemos algoritmos mais rápidos do que os incrementais

Entendendo e melhorando

Até agora:

- ordenamos incrementalmente com o Insertion-Sort
- vimos que sua complexidade de pior caso é $\Theta(n^2)$

Vamos estudar uma maneira alternativa de ordenar números

- vamos utilizar uma técnica recursiva chamada de divisão e conquista
- muitas vezes, obtemos algoritmos mais rápidos do que os incrementais

Algoritmos recursivos

> *"To understand recursion, we must first understand recursion."*
> (autor desconhecido)

- Um algoritmo recursivo resolve um problema
  - diretamente, se a instância for pequena
  - executando a si mesmo, se a instância não for pequena
- A chamada recursiva deve receber uma instância menor

Algoritmos recursivos

"To understand recursion, we must first understand recursion."
(autor desconhecido)

- Um algoritmo recursivo resolve um problema
  - diretamente, se a instância for pequena
  - executando a si mesmo, se a instância não for pequena
- A chamada recursiva deve receber uma instância menor

Algoritmos recursivos

"To understand recursion, we must first understand recursion."
(autor desconhecido)

- Um algoritmo recursivo resolve um problema
  - diretamente, se a instância for pequena
  - executando a si mesmo, se a instância não for pequena
- A chamada recursiva deve receber uma instância menor

Algoritmos recursivos

"To understand recursion, we must first understand recursion."
(autor desconhecido)

- Um algoritmo recursivo resolve um problema
  - diretamente, se a instância for pequena
  - executando a si mesmo, se a instância não for pequena
- A chamada recursiva deve receber uma instância menor

Algoritmos recursivos

"To understand recursion, we must first understand recursion."
(autor desconhecido)

- Um algoritmo recursivo resolve um problema
  - diretamente, se a instância for pequena
  - executando a si mesmo, se a instância não for pequena
- A chamada recursiva deve receber uma instância menor

Divisão e conquista

Um algoritmo de divisão e conquista tem três etapas:

1. Divisão: dividir o problema em subproblemas semelhantes, mas com instâncias menores
2. Conquista: cada subproblema é resolvido recursivamente, ou diretamente se os subproblemas forem pequenos
3. Combinação: as soluções dos subproblemas são combinadas para obter uma solução da instância original

Divisão e conquista

Um algoritmo de divisão e conquista tem três etapas:

1. Divisão: dividir o problema em subproblemas semelhantes, mas com instâncias menores
2. Conquista: cada subproblema é resolvido recursivamente, ou diretamente se os subproblemas forem pequenos
3. Combinação: as soluções dos subproblemas são combinadas para obter uma solução da instância original

Divisão e conquista

Um algoritmo de divisão e conquista tem três etapas:

1. Divisão: dividir o problema em subproblemas semelhantes, mas com instâncias menores
2. Conquista: cada subproblema é resolvido recursivamente, ou diretamente se os subproblemas forem pequenos
3. Combinação: as soluções dos subproblemas são combinadas para obter uma solução da instância original

Divisão e conquista

Um algoritmo de divisão e conquista tem três etapas:

1. Divisão: dividir o problema em subproblemas semelhantes, mas com instâncias menores
2. Conquista: cada subproblema é resolvido recursivamente, ou diretamente se os subproblemas forem pequenos
3. Combinação: as soluções dos subproblemas são combinadas para obter uma solução da instância original

Exemplo: ordenando usando divisão e conquista

MERGE-SORT é um exemplo clássico de divisão e conquista.

Ideia:

1. Divisão: divida um vetor de tamanho n em dois subvetores de tamanhos $\lfloor n/2 \rfloor$ e $\lceil n/2 \rceil$
2. Conquista: ordene os dois subvetores recursivamente
3. Combinação: intercale os dois subvetores obtendo um vetor ordenado

Exemplo: ordenando usando divisão e conquista

MERGE-SORT é um exemplo clássico de divisão e conquista.

Ideia:

1. Divisão: divida um vetor de tamanho n em dois subvetores de tamanhos $\lfloor n/2 \rfloor$ e $\lceil n/2 \rceil$
2. Conquista: ordene os dois subvetores recursivamente
3. Combinação: intercale os dois subvetores obtendo um vetor ordenado

Exemplo: ordenando usando divisão e conquista

MERGE-SORT é um exemplo clássico de divisão e conquista.

Ideia:

1. Divisão: divida um vetor de tamanho n em dois subvetores de tamanhos $\lfloor n/2 \rfloor$ e $\lceil n/2 \rceil$
2. Conquista: ordene os dois subvetores recursivamente
3. Combinação: intercale os dois subvetores obtendo um vetor ordenado

Exemplo: ordenando usando divisão e conquista

MERGE-SORT é um exemplo clássico de divisão e conquista.

Ideia:

1. Divisão: divida um vetor de tamanho n em dois subvetores de tamanhos $\lfloor n/2 \rfloor$ e $\lceil n/2 \rceil$
2. Conquista: ordene os dois subvetores recursivamente
3. Combinação: intercale os dois subvetores obtendo um vetor ordenado

Exemplo: ordenando usando divisão e conquista

MERGE-SORT é um exemplo clássico de divisão e conquista.

Ideia:

1. Divisão: divida um vetor de tamanho n em dois subvetores de tamanhos $\lfloor n/2 \rfloor$ e $\lceil n/2 \rceil$
2. Conquista: ordene os dois subvetores recursivamente
3. Combinação: intercale os dois subvetores obtendo um vetor ordenado

Mergesort

O vetor de entrada é representado como $A[p \ldots r]$, com $p \leq r$.

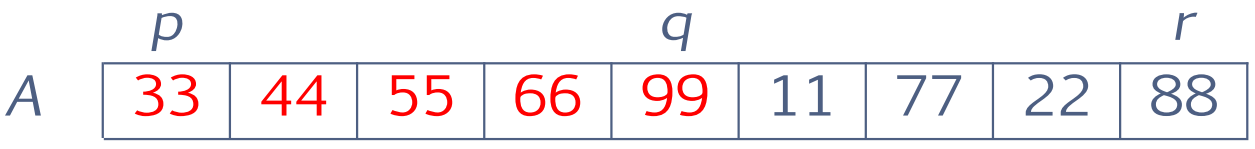
```
MERGE-SORT(A, p, r)
1   se p < r
2       então q ← ⌊(p + r)/2⌋
3             MERGE-SORT(A, p, q)
4             MERGE-SORT(A, q + 1, r)
5             INTERCALA(A, p, q, r)
```

	p				q				r
A	66	33	55	44	99	11	77	22	88

Mergesort

O vetor de entrada é representado como $A[p \ldots r]$, com $p \leq r$.

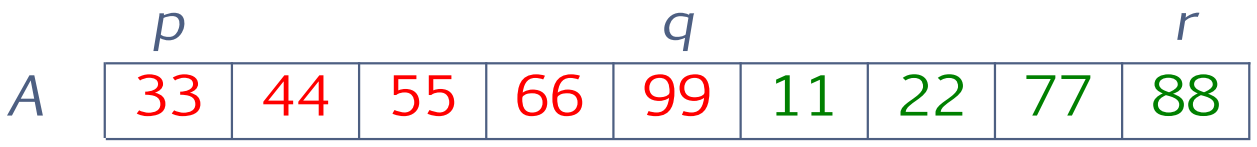
```
MERGE-SORT(A, p, r)
1   se p < r
2      então q ← ⌊(p + r)/2⌋
3            MERGE-SORT(A, p, q)
4            MERGE-SORT(A, q + 1, r)
5            INTERCALA(A, p, q, r)
```



Mergesort

O vetor de entrada é representado como $A[p \ldots r]$, com $p \leq r$.

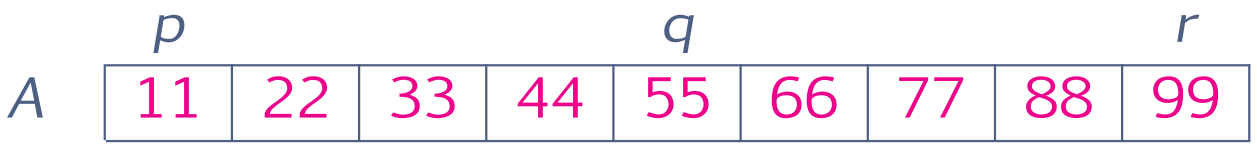
```
MERGE-SORT(A, p, r)
1   se p < r
2       então q ← ⌊(p + r)/2⌋
3             MERGE-SORT(A, p, q)
4             MERGE-SORT(A, q + 1, r)
5             INTERCALA(A, p, q, r)
```



Mergesort

O vetor de entrada é representado como $A[p \ldots r]$, com $p \leq r$.

```
MERGE-SORT(A, p, r)
1   se p < r
2       então q ← ⌊(p + r)/2⌋
3             MERGE-SORT(A, p, q)
4             MERGE-SORT(A, q + 1, r)
5             INTERCALA(A, p, q, r)
```



Combinando soluções dos subproblemas

Problema: Intercalar dois subvetores

Entrada: vetor $A[p \ldots r]$ tal que

1. subvetor $A[p \ldots q]$ está ordenado
2. subvetor $A[q+1 \ldots r]$ está ordenado

Saída: rearranjo $A[p \ldots r]$ ordenado

Entrada:

	p				q				r
A	22	33	55	77	99	11	44	66	88

Saída:

	p				q				r
A	11	22	33	44	55	66	77	88	99

Combinando soluções dos subproblemas

Problema: Intercalar dois subvetores

Entrada: vetor $A[p \ldots r]$ tal que

1. subvetor $A[p \ldots q]$ está ordenado
2. subvetor $A[q+1 \ldots r]$ está ordenado

Saída: rearranjo $A[p \ldots r]$ ordenado

Entrada:

	p				q				r
A	22	33	55	77	99	11	44	66	88

Saída:

	p				q				r
A	11	22	33	44	55	66	77	88	99

Combinando soluções dos subproblemas

Problema: Intercalar dois subvetores

Entrada: vetor $A[p \ldots r]$ tal que

1. subvetor $A[p \ldots q]$ está ordenado
2. subvetor $A[q+1 \ldots r]$ está ordenado

Saída: rearranjo $A[p \ldots r]$ ordenado

Entrada:

	p				q				r
A	22	33	55	77	99	11	44	66	88

Saída:

	p				q				r
A	11	22	33	44	55	66	77	88	99

Combinando soluções dos subproblemas

Problema: Intercalar dois subvetores

Entrada: vetor $A[p \ldots r]$ tal que

1. subvetor $A[p \ldots q]$ está ordenado
2. subvetor $A[q+1 \ldots r]$ está ordenado

Saída: rearranjo $A[p \ldots r]$ ordenado

Entrada:

	p				q				r
A	22	33	55	77	99	11	44	66	88

Saída:

	p				q				r
A	11	22	33	44	55	66	77	88	99

Combinando soluções dos subproblemas

Problema: Intercalar dois subvetores

Entrada: vetor $A[p \ldots r]$ tal que

1. subvetor $A[p \ldots q]$ está ordenado
2. subvetor $A[q+1 \ldots r]$ está ordenado

Saída: rearranjo $A[p \ldots r]$ ordenado

Entrada:

	p				q				r
A	22	33	55	77	99	11	44	66	88

Saída:

	p				q				r
A	11	22	33	44	55	66	77	88	99

Combinando soluções dos subproblemas

Problema: Intercalar dois subvetores

Entrada: vetor $A[p \ldots r]$ tal que

1. subvetor $A[p \ldots q]$ está ordenado
2. subvetor $A[q+1 \ldots r]$ está ordenado

Saída: rearranjo $A[p \ldots r]$ ordenado

Entrada:

	p				q				r
A	22	33	55	77	99	11	44	66	88

Saída:

	p				q				r
A	11	22	33	44	55	66	77	88	99

Combinando soluções dos subproblemas

Problema: Intercalar dois subvetores

Entrada: vetor $A[p \ldots r]$ tal que

1. subvetor $A[p \ldots q]$ está ordenado
2. subvetor $A[q+1 \ldots r]$ está ordenado

Saída: rearranjo $A[p \ldots r]$ ordenado

Entrada:

	p				q				r
A	22	33	55	77	99	11	44	66	88

Saída:

	p				q				r
A	11	22	33	44	55	66	77	88	99

Combinando soluções dos subproblemas

Problema: Intercalar dois subvetores

Entrada: vetor $A[p \ldots r]$ tal que

1. subvetor $A[p \ldots q]$ está ordenado
2. subvetor $A[q+1 \ldots r]$ está ordenado

Saída: rearranjo $A[p \ldots r]$ ordenado

Entrada:

	p				q				r
A	22	33	55	77	99	11	44	66	88

Saída:

	p				q				r
A	11	22	33	44	55	66	77	88	99

Combinando soluções dos subproblemas

Problema: Intercalar dois subvetores

Entrada: vetor $A[p \ldots r]$ tal que

1. subvetor $A[p \ldots q]$ está ordenado
2. subvetor $A[q+1 \ldots r]$ está ordenado

Saída: rearranjo $A[p \ldots r]$ ordenado

Entrada:

	p				q				r
A	22	33	55	77	99	11	44	66	88

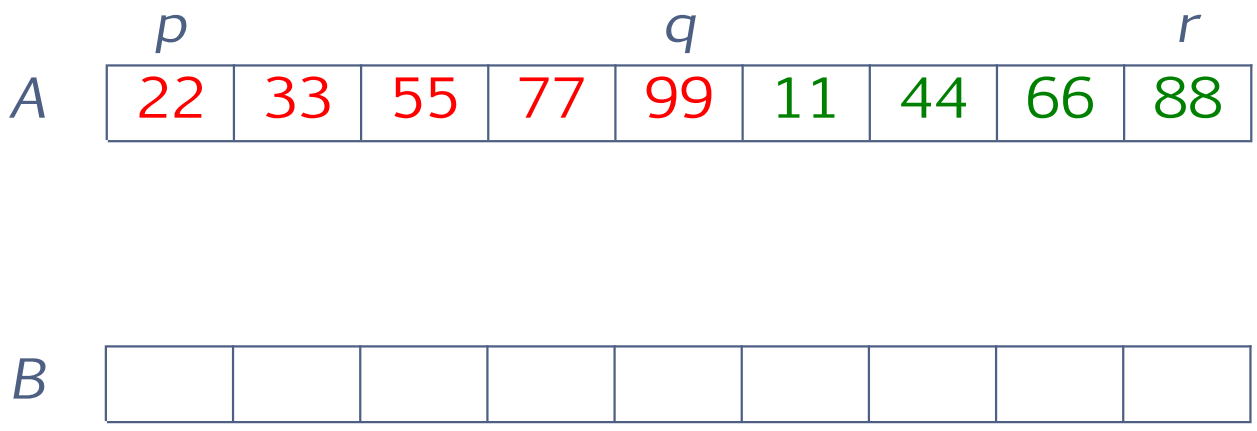
Saída:

	p				q				r
A	11	22	33	44	55	66	77	88	99

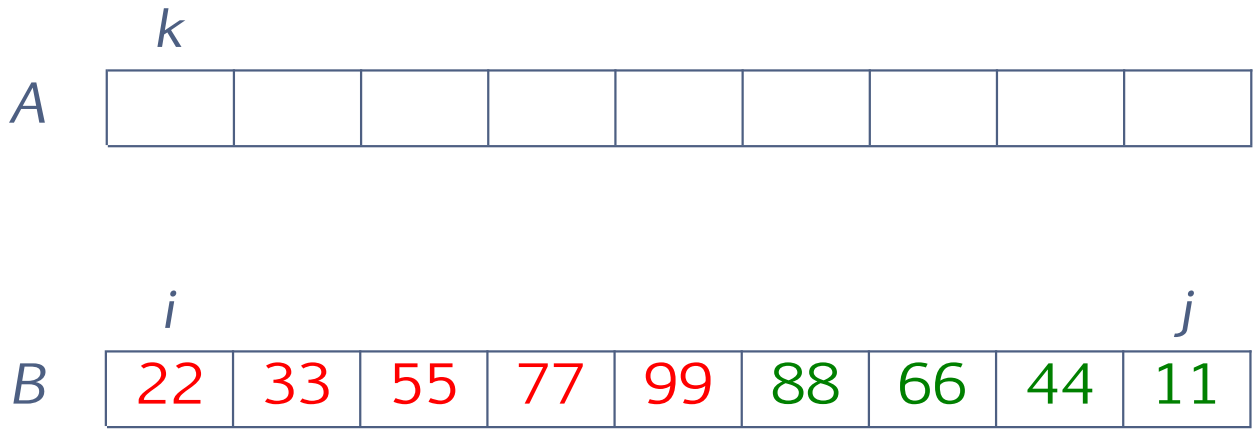
Pseudocódigo de INTERCALA

```
INTERCALA(A, p, q, r)
 1  para i ← p até q faça
 2      B[i] ← A[i]
 3  para j ← q + 1 até r faça
 4      B[r + q + 1 − j] ← A[j]
 5  i ← p
 6  j ← r
 7  para k ← p até r faça
 8      se B[i] ≤ B[j]
 9          então A[k] ← B[i]
10                i ← i + 1
11          senão A[k] ← B[j]
12                j ← j − 1
```

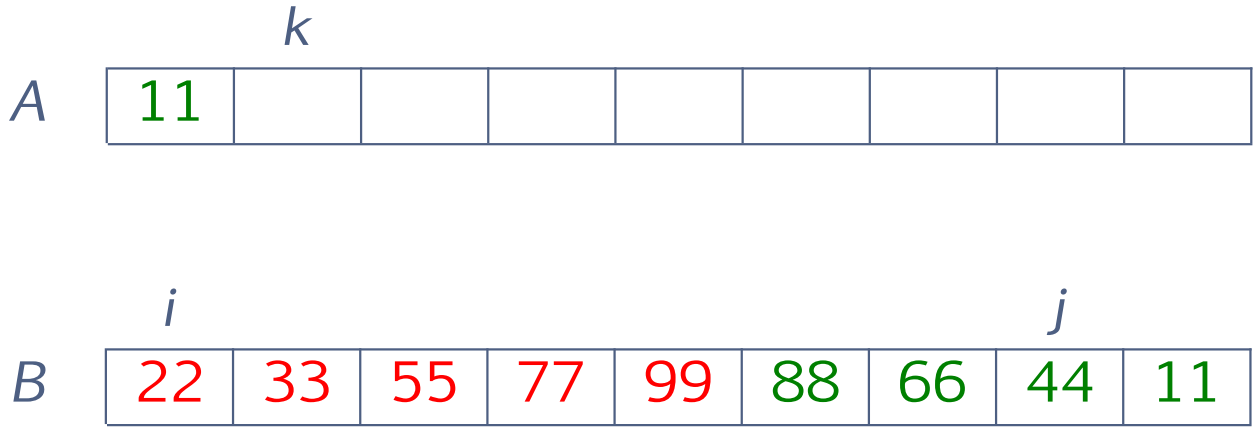
Intercalando



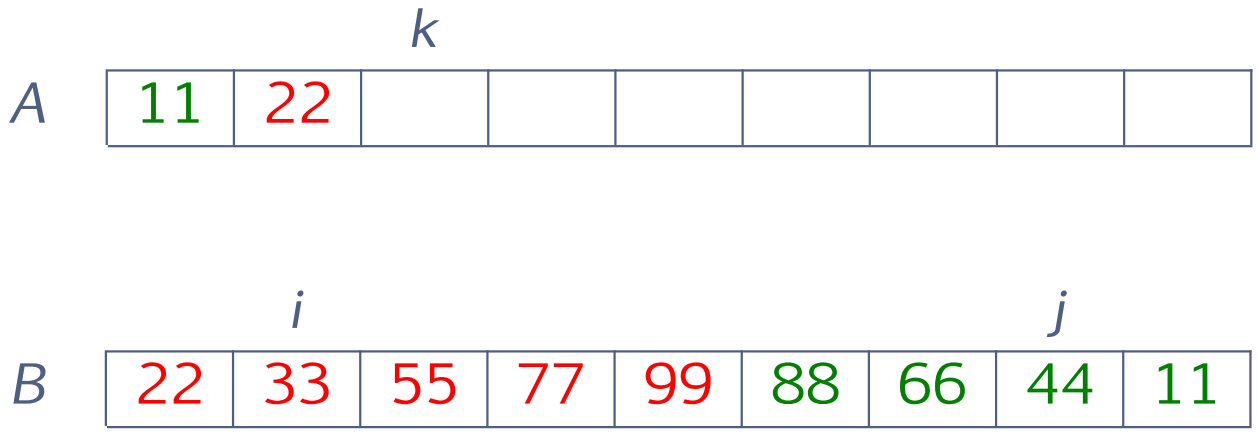
Intercalando



Intercalando

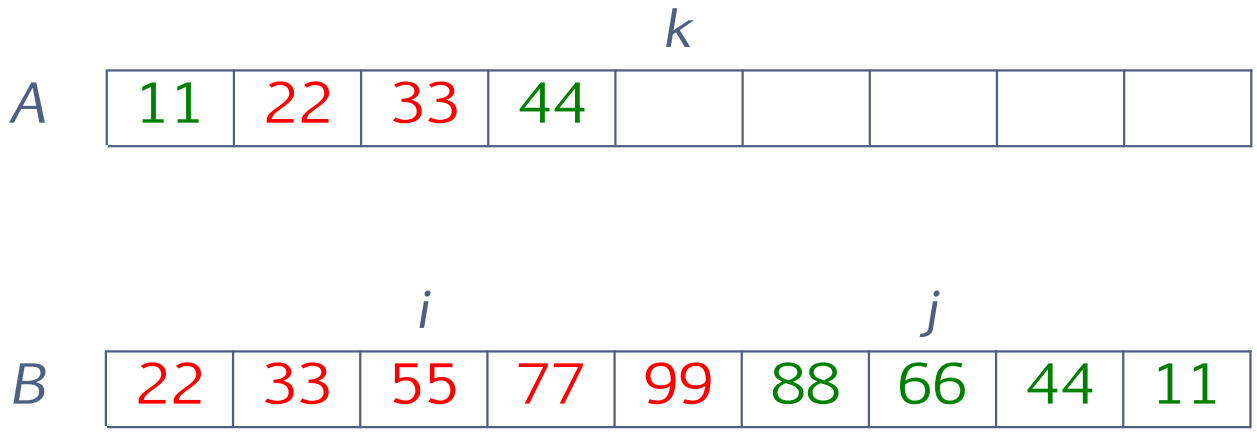


Intercalando



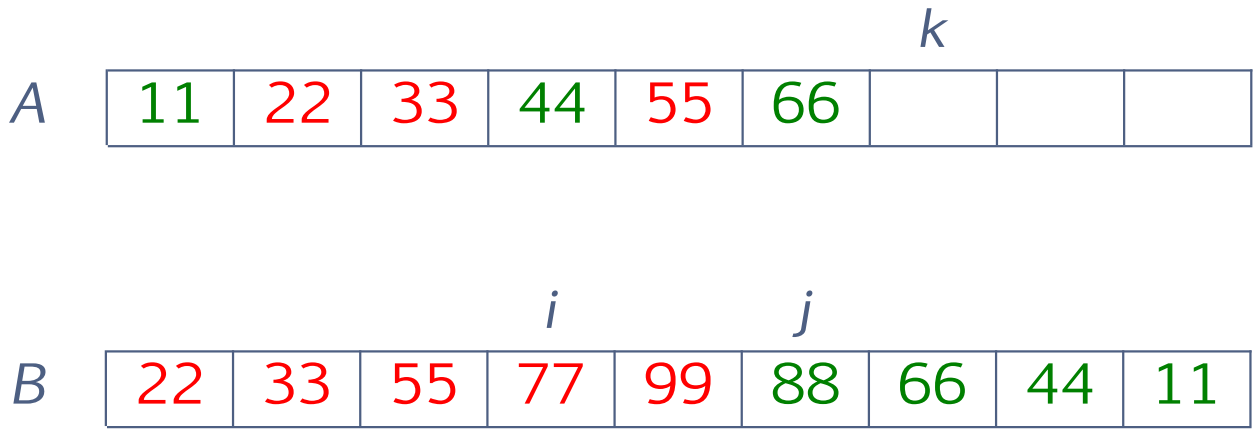
Intercalando

Intercalando

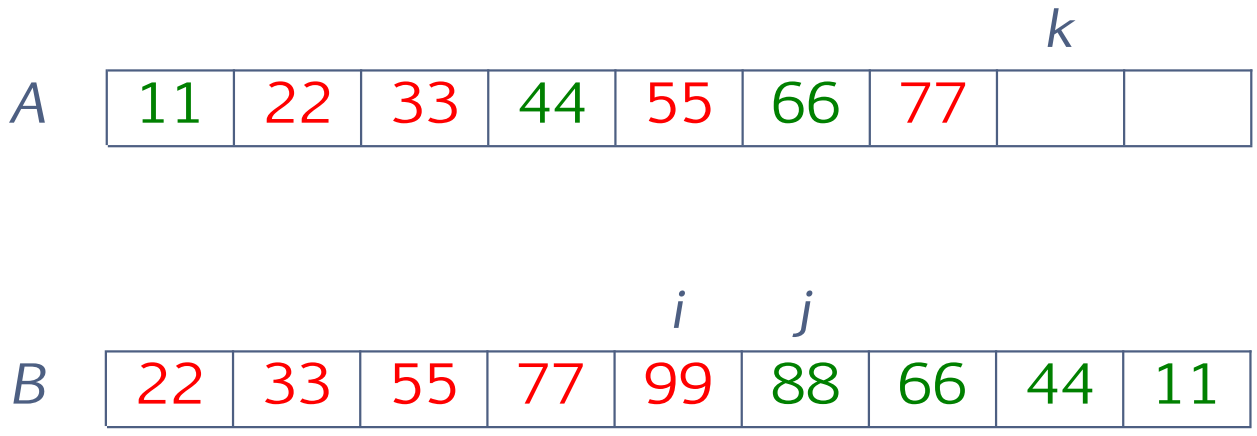


Intercalando

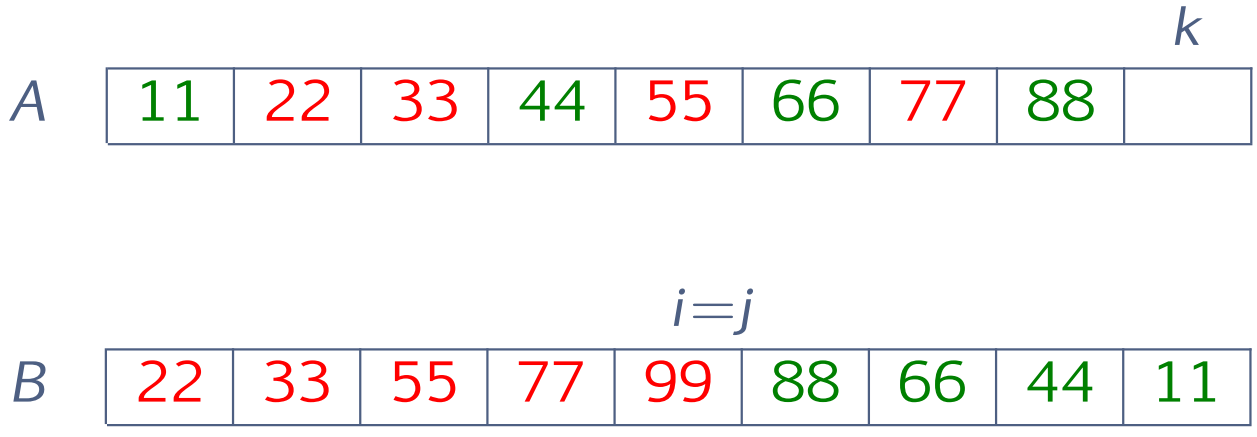
Intercalando



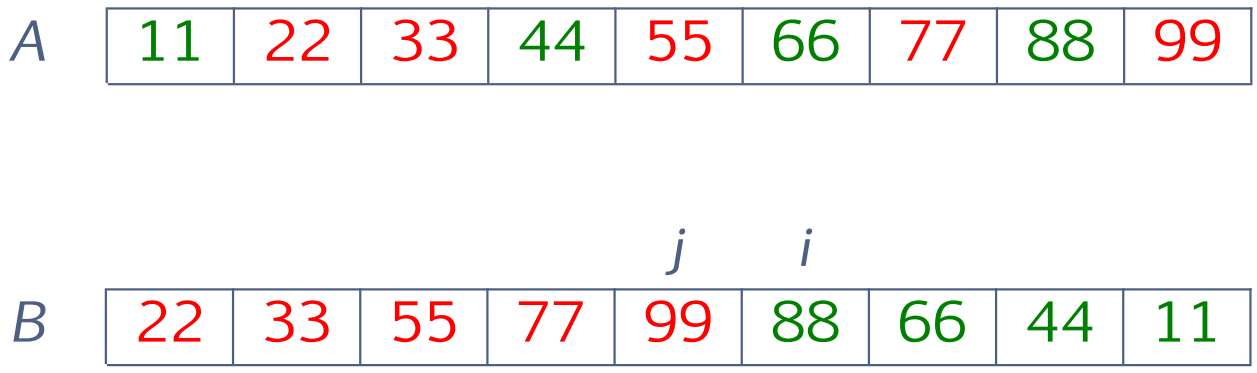
Intercalando



Intercalando



Intercalando



Complexidade de Intercala

Entrada:

	p				q				r
A	22	33	55	77	99	11	44	66	88

Saída:

	p				q				r
A	11	22	33	44	55	66	77	88	99

Tamanho da entrada: $n = r - p + 1$

Consumo de tempo: $\Theta(n)$

Ordenando por intercalação

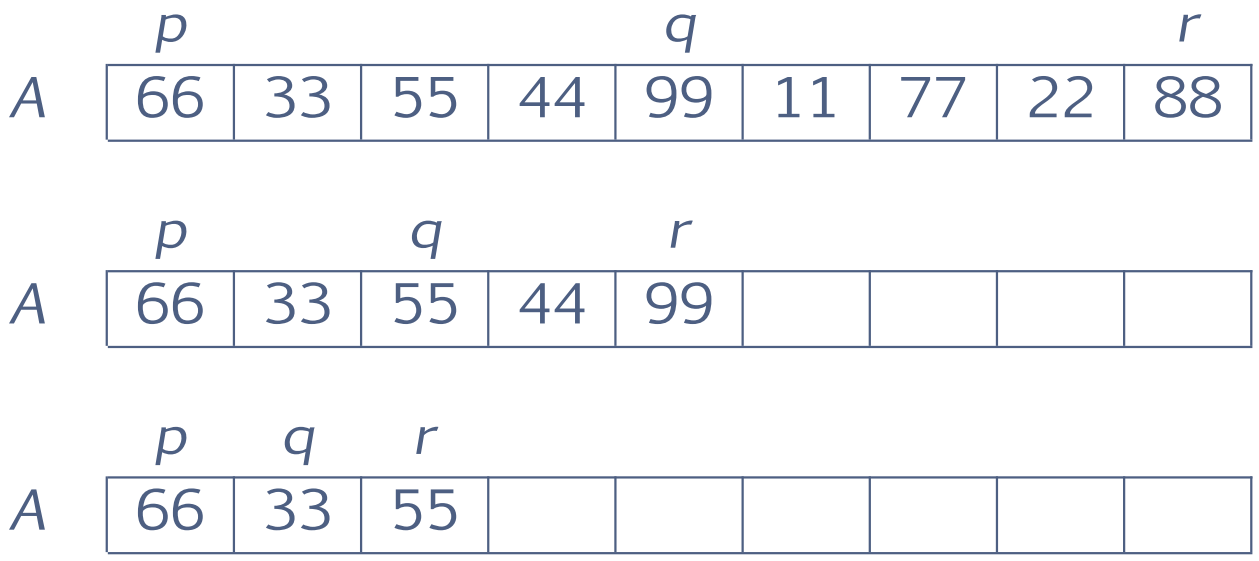
	p				q				r
A	66	33	55	44	99	11	77	22	88

Ordenando por intercalação

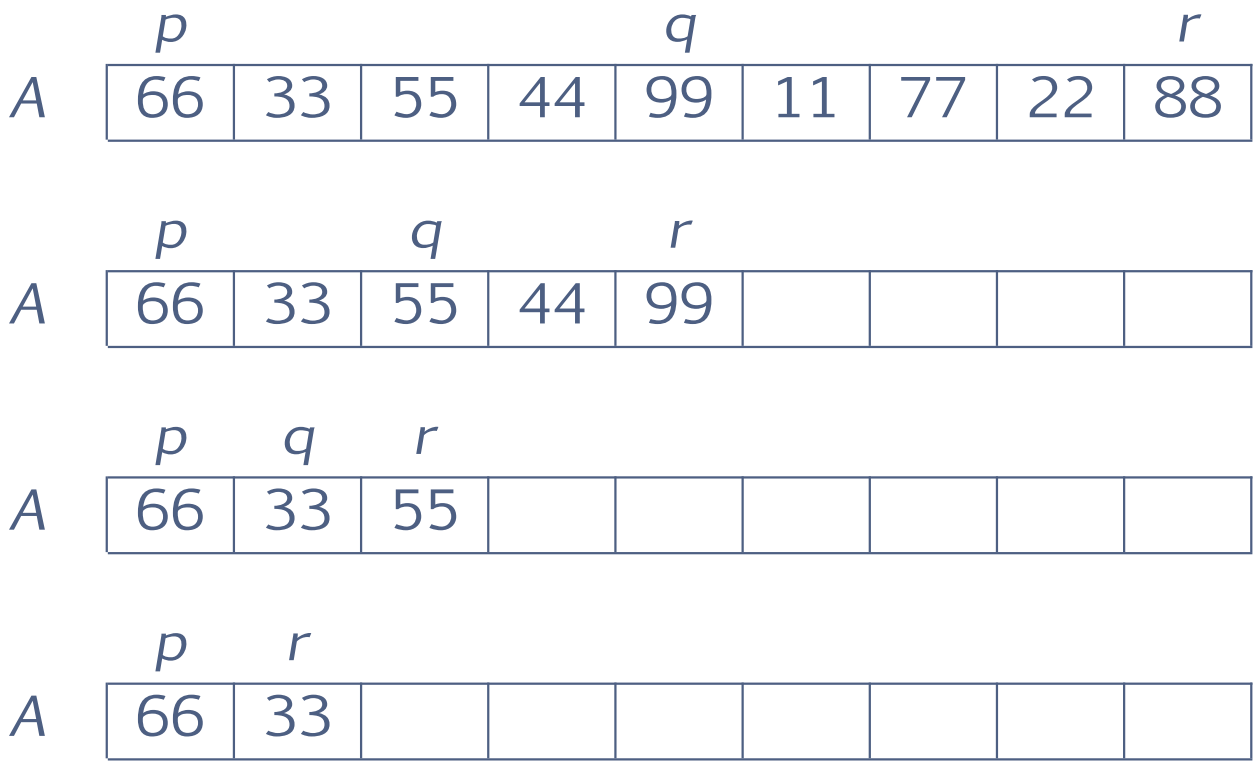
	p				q				r
A	66	33	55	44	99	11	77	22	88

	p		q		r				
A	66	33	55	44	99				

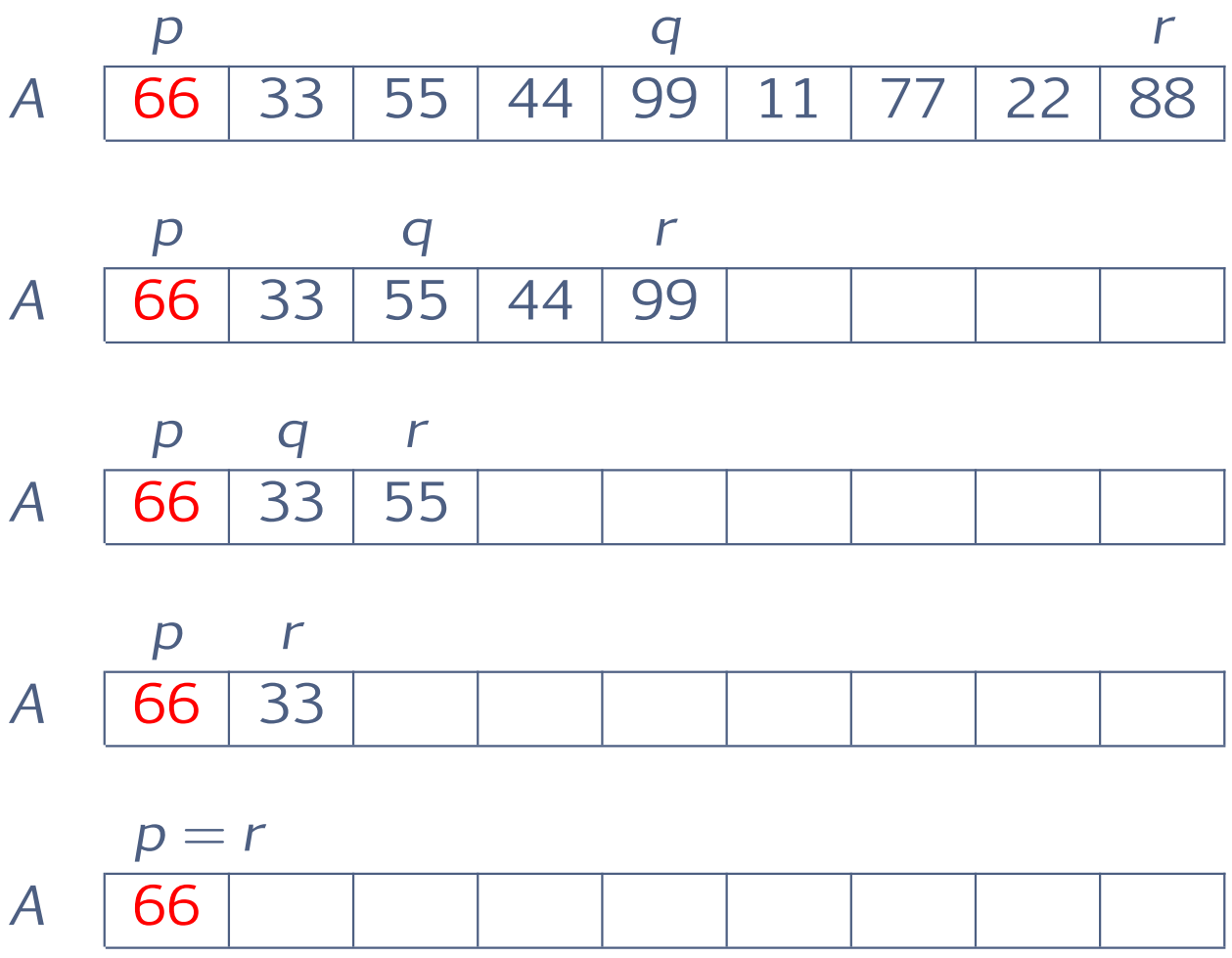
Ordenando por intercalação



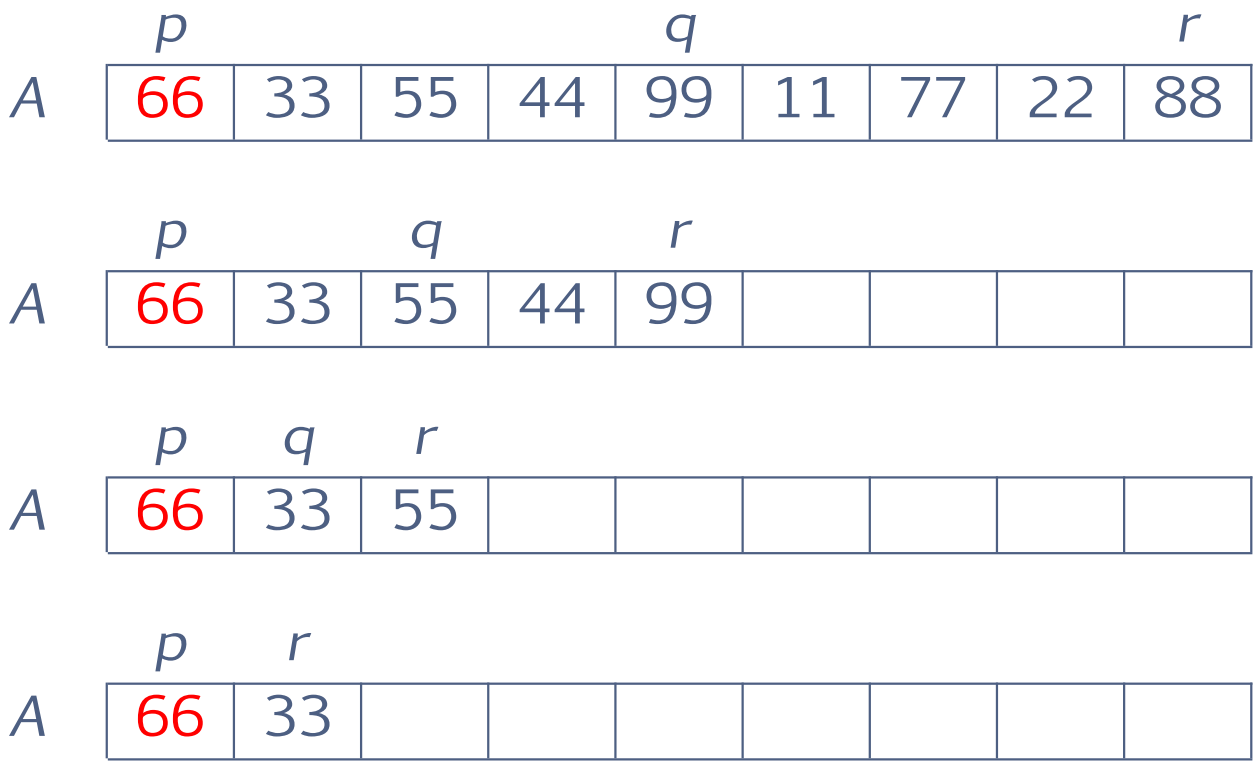
Ordenando por intercalação



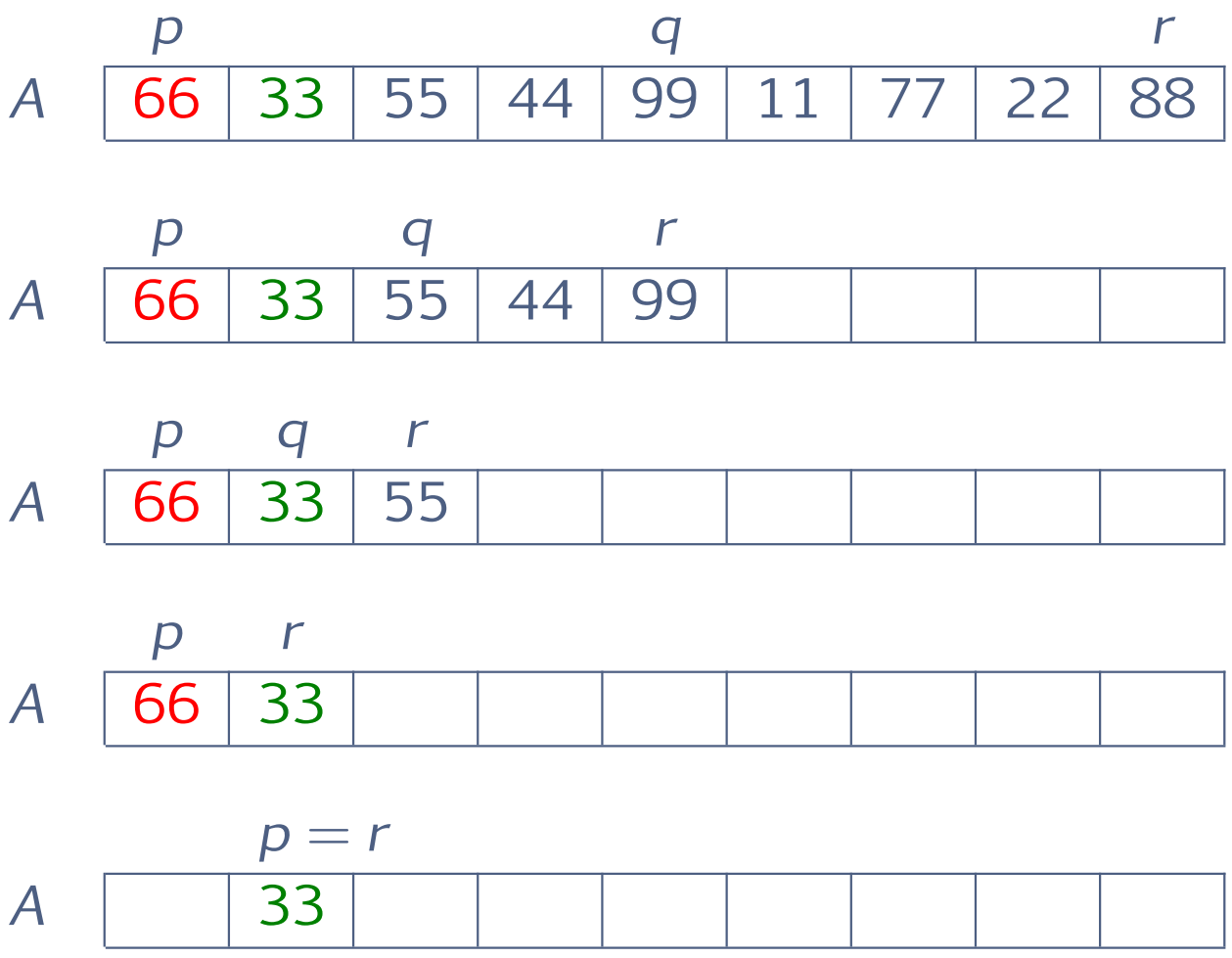
Ordenando por intercalação



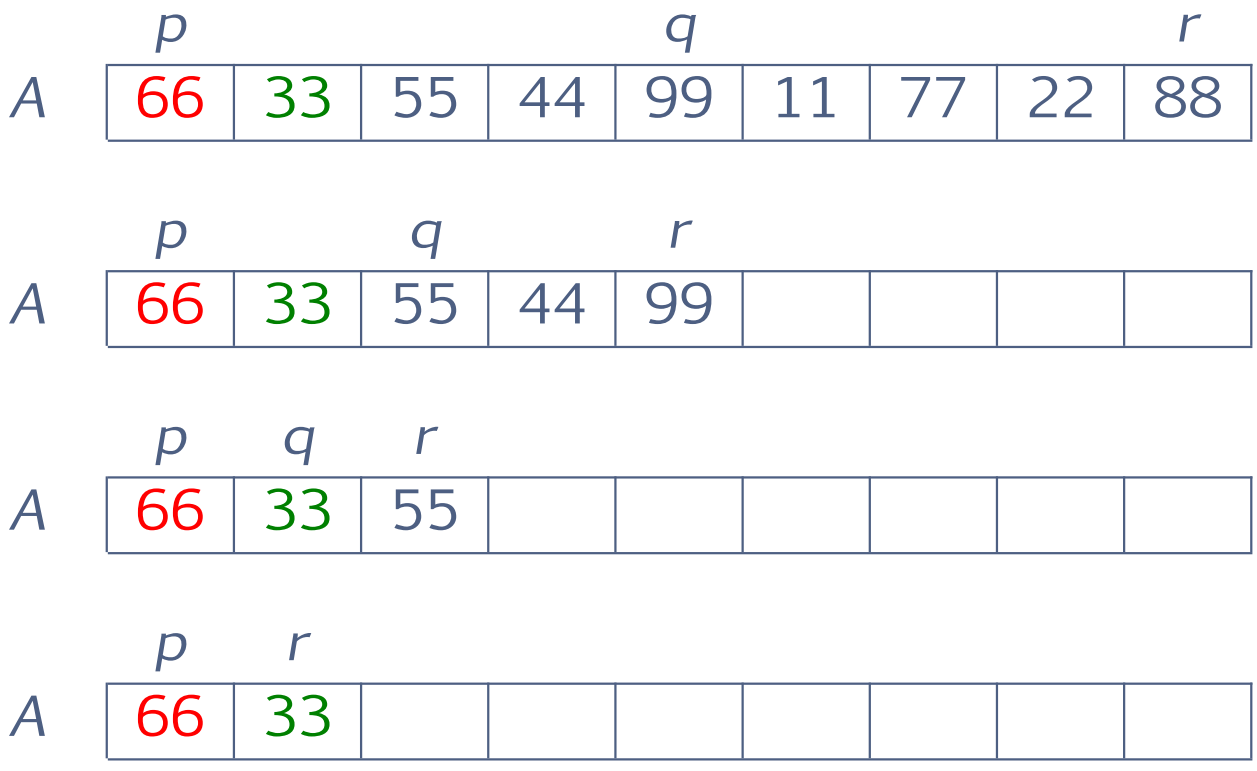
Ordenando por intercalação



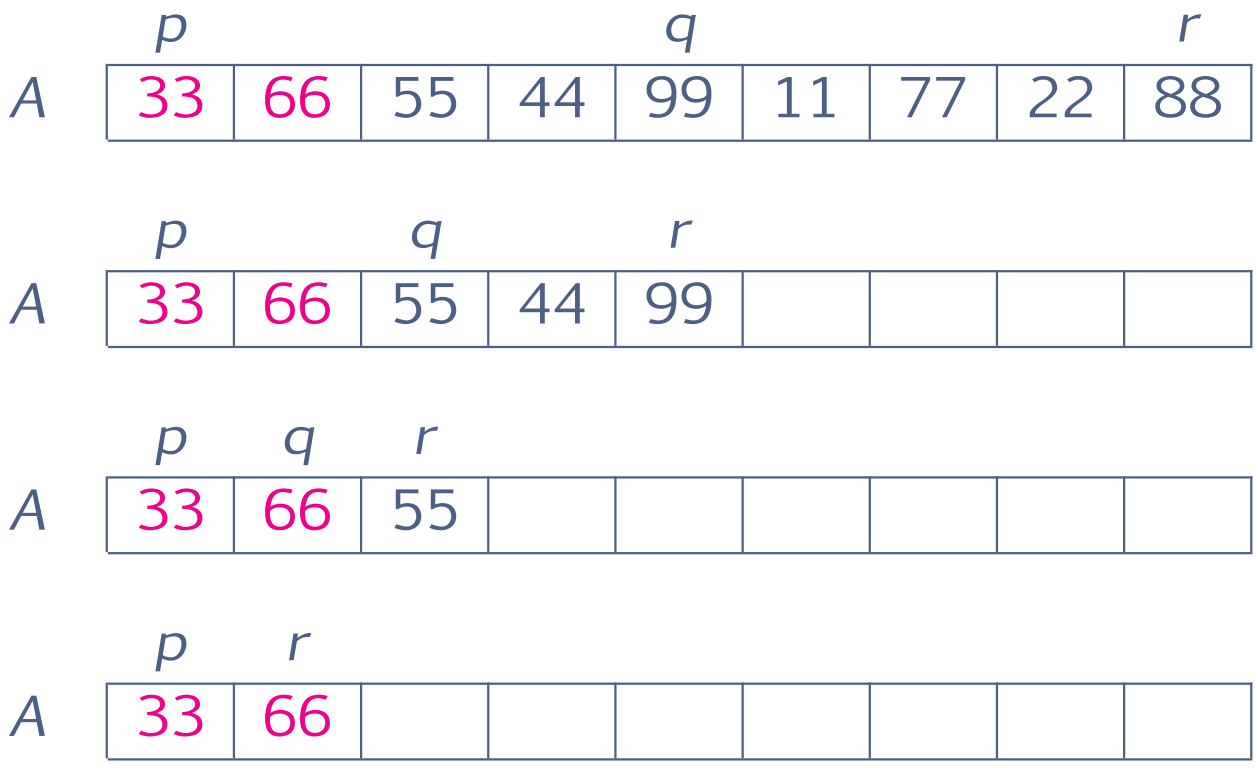
Ordenando por intercalação



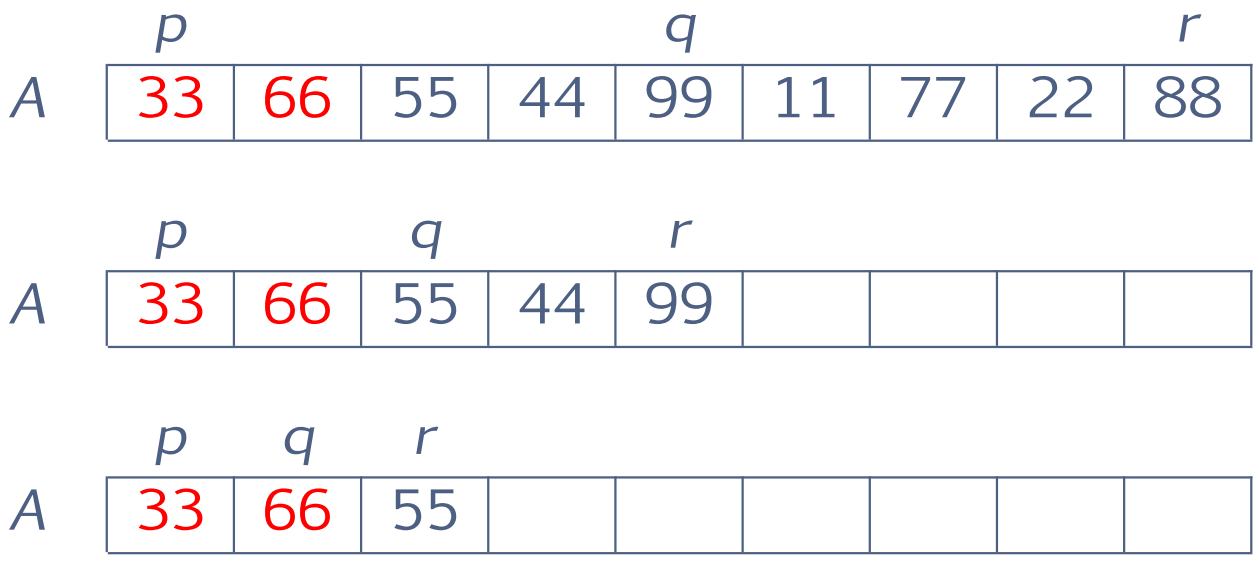
Ordenando por intercalação



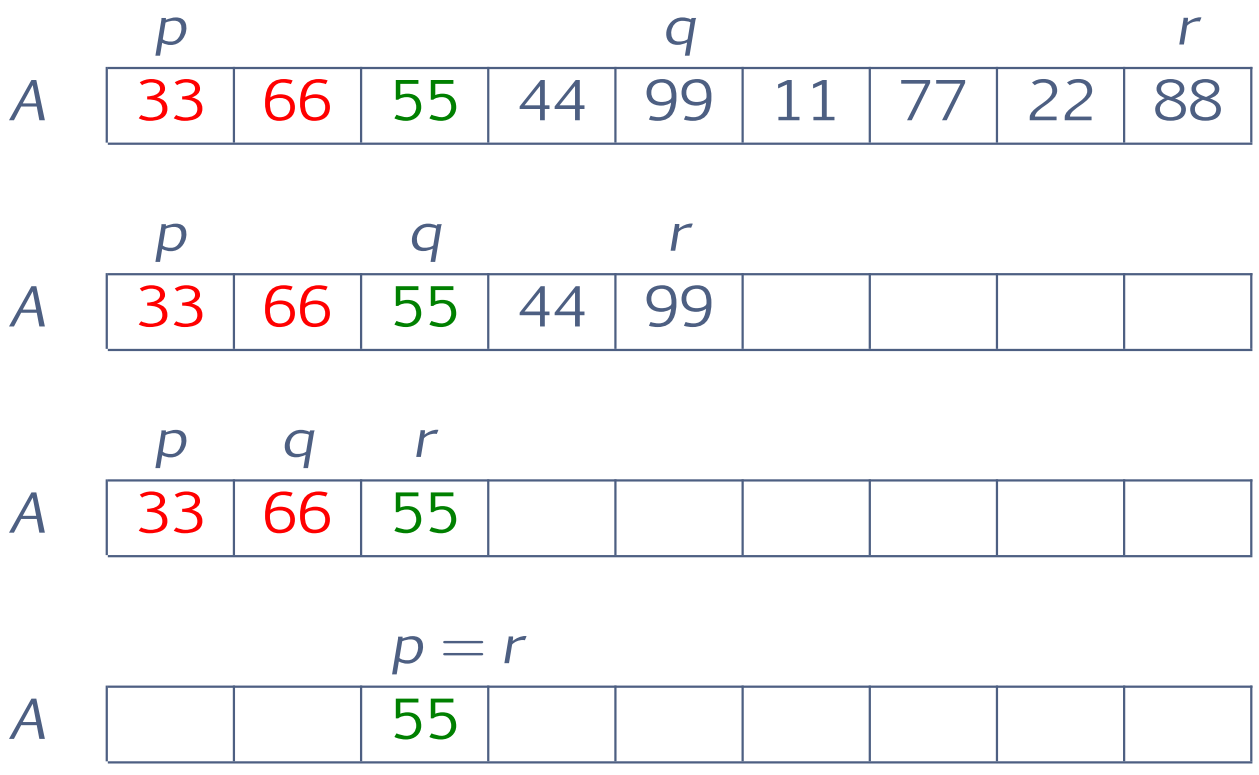
Ordenando por intercalação



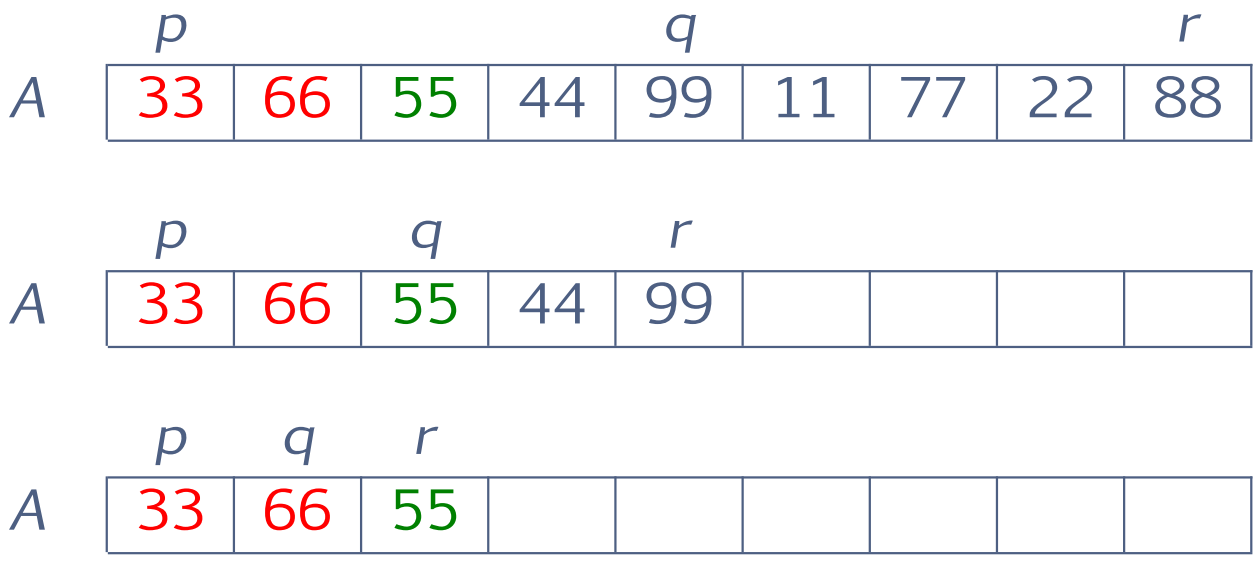
Ordenando por intercalação



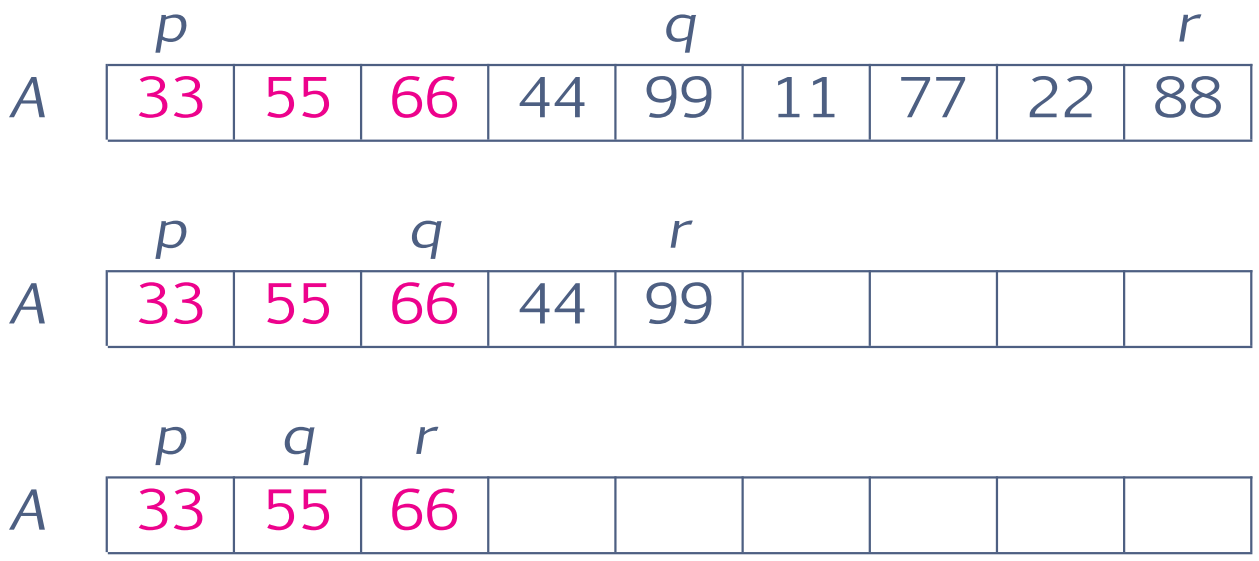
Ordenando por intercalação



Ordenando por intercalação

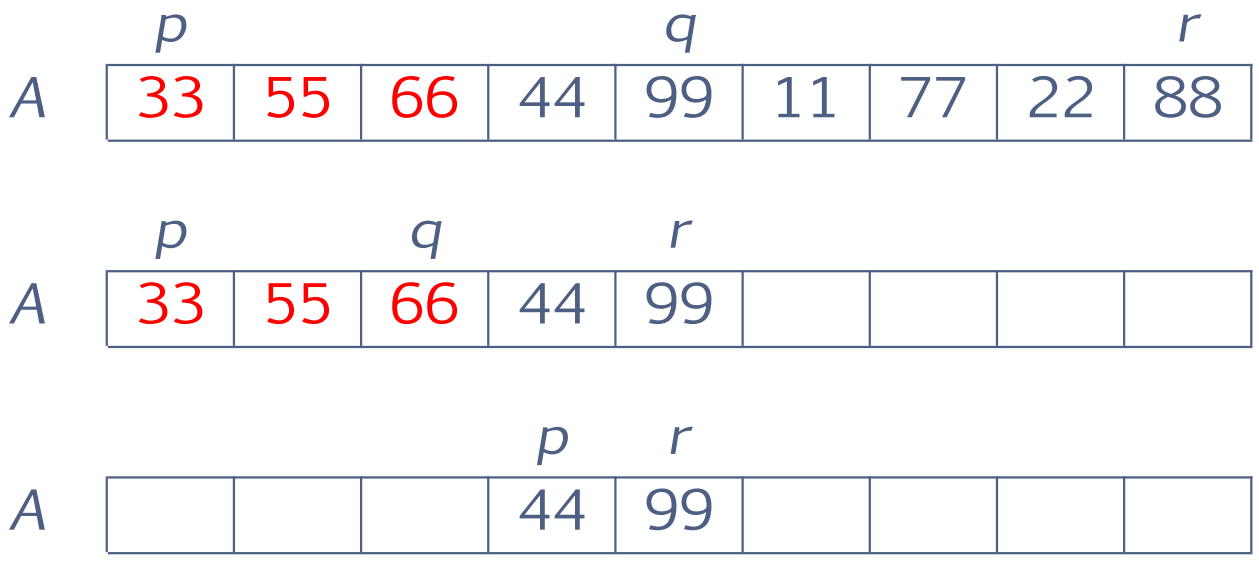


Ordenando por intercalação

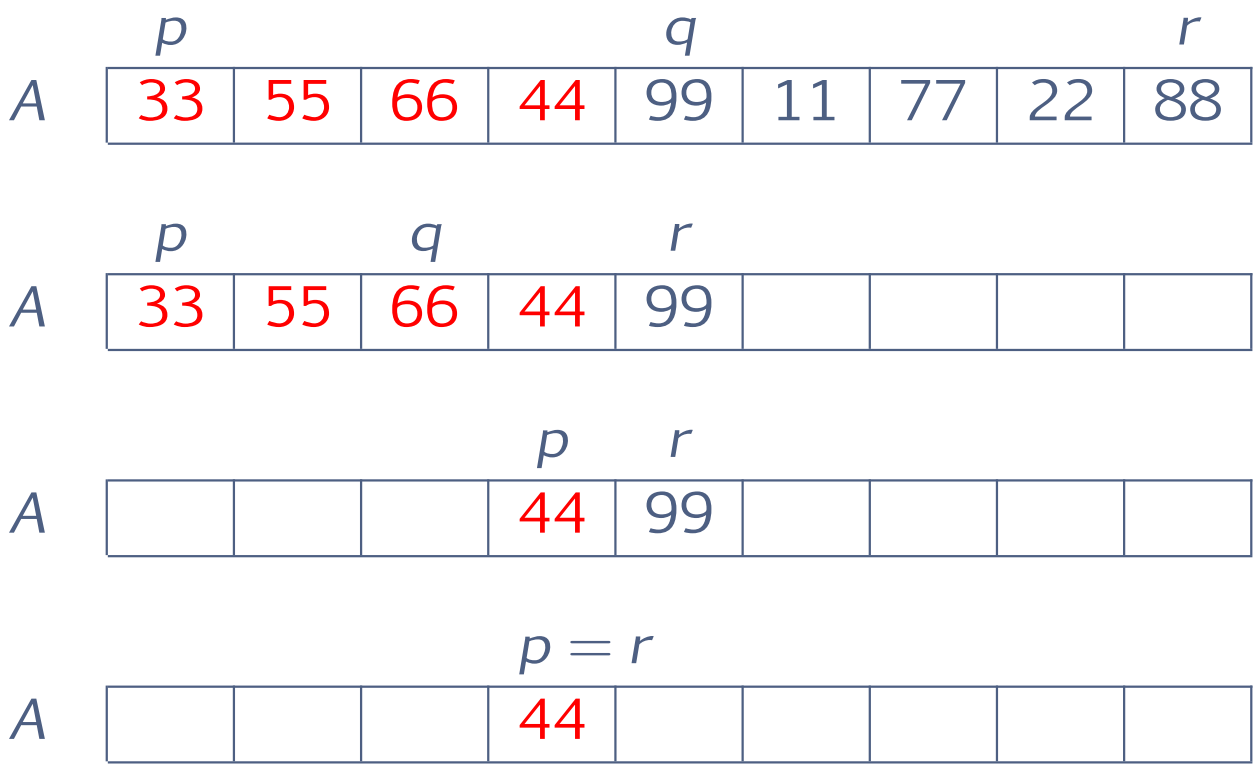


Ordenando por intercalação

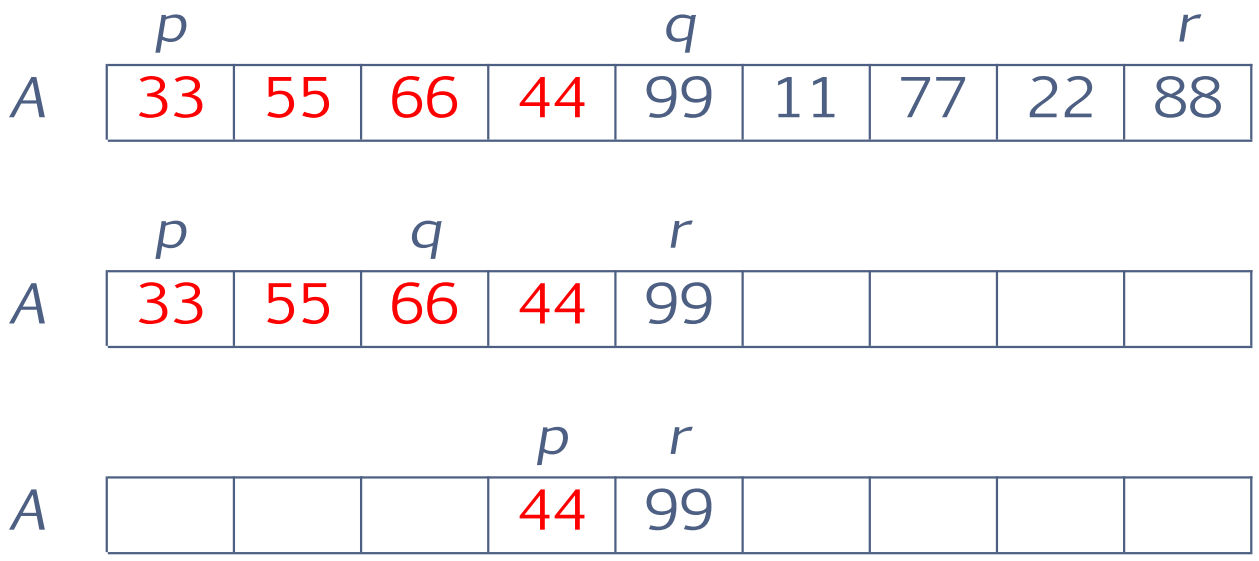
Ordenando por intercalação



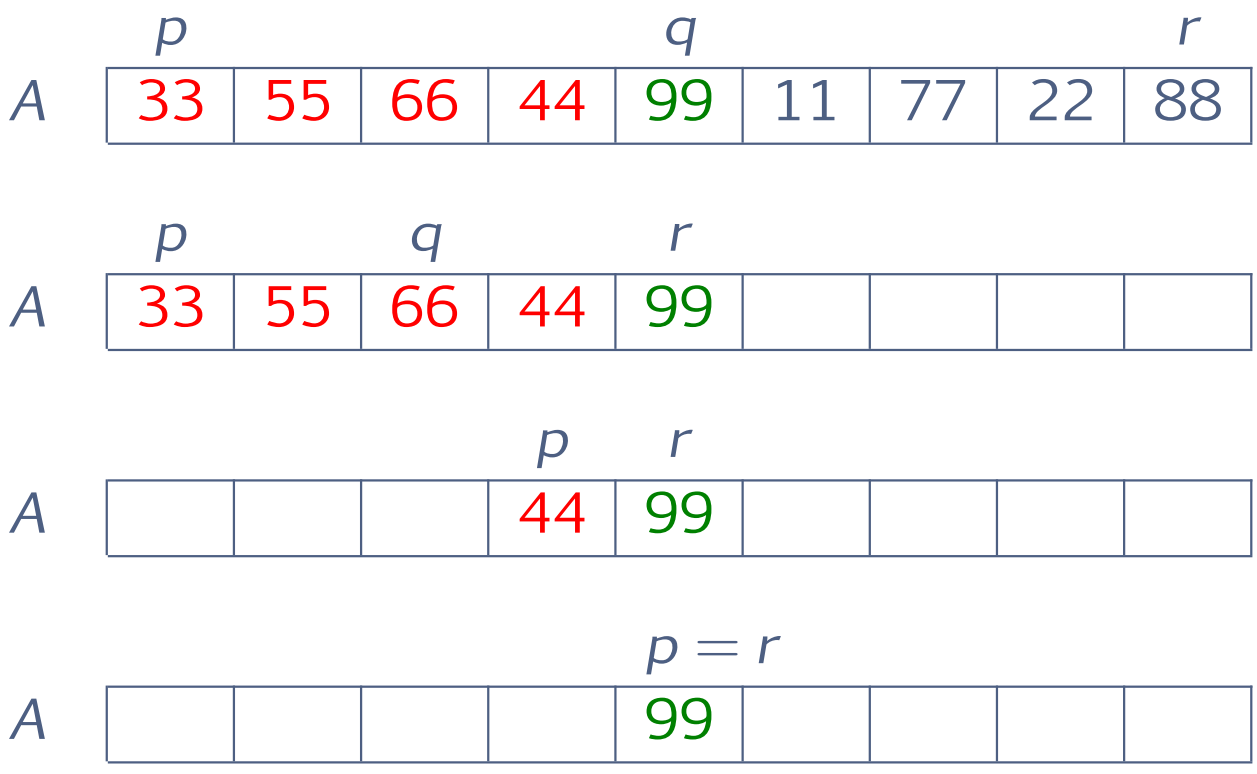
Ordenando por intercalação



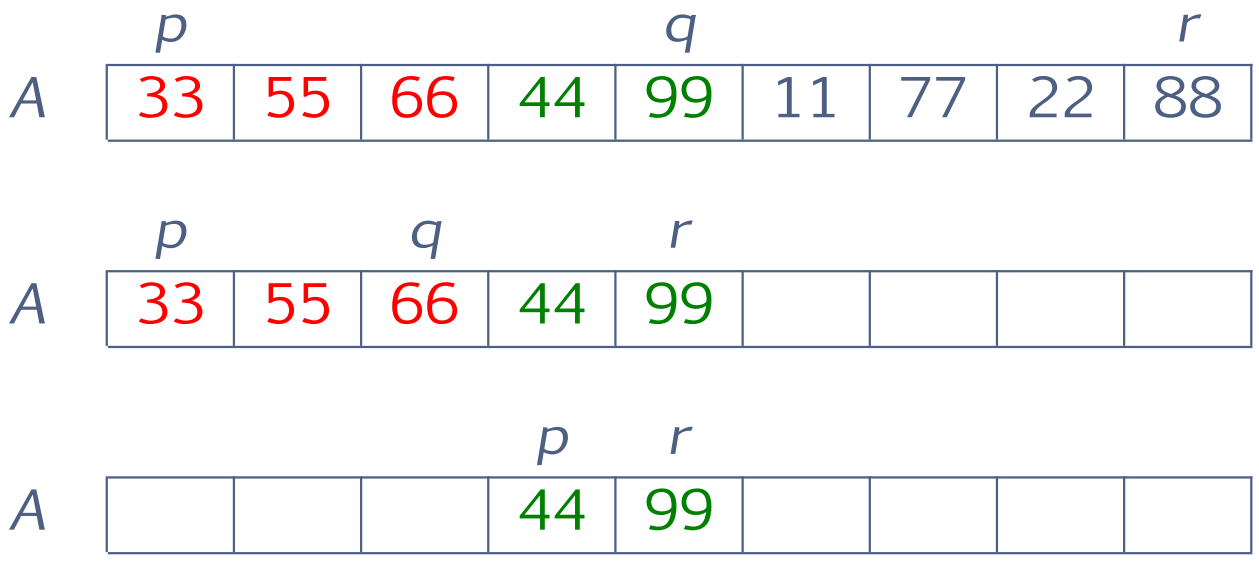
Ordenando por intercalação



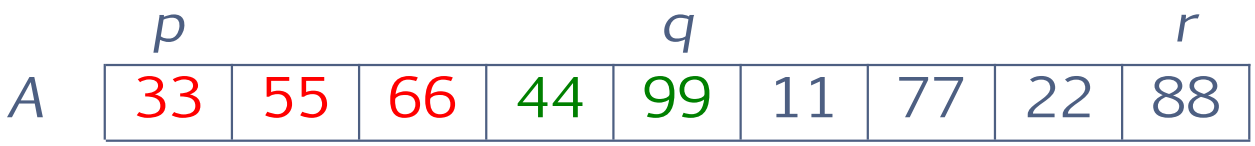
Ordenando por intercalação



Ordenando por intercalação



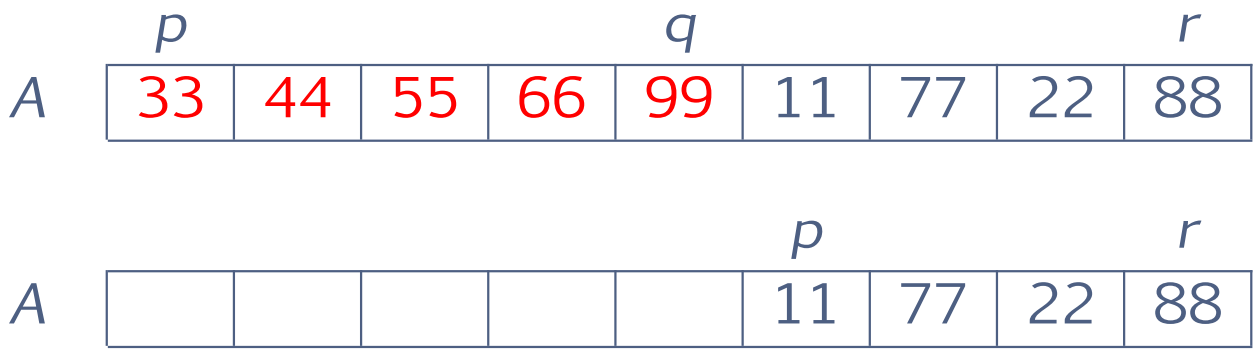
Ordenando por intercalação



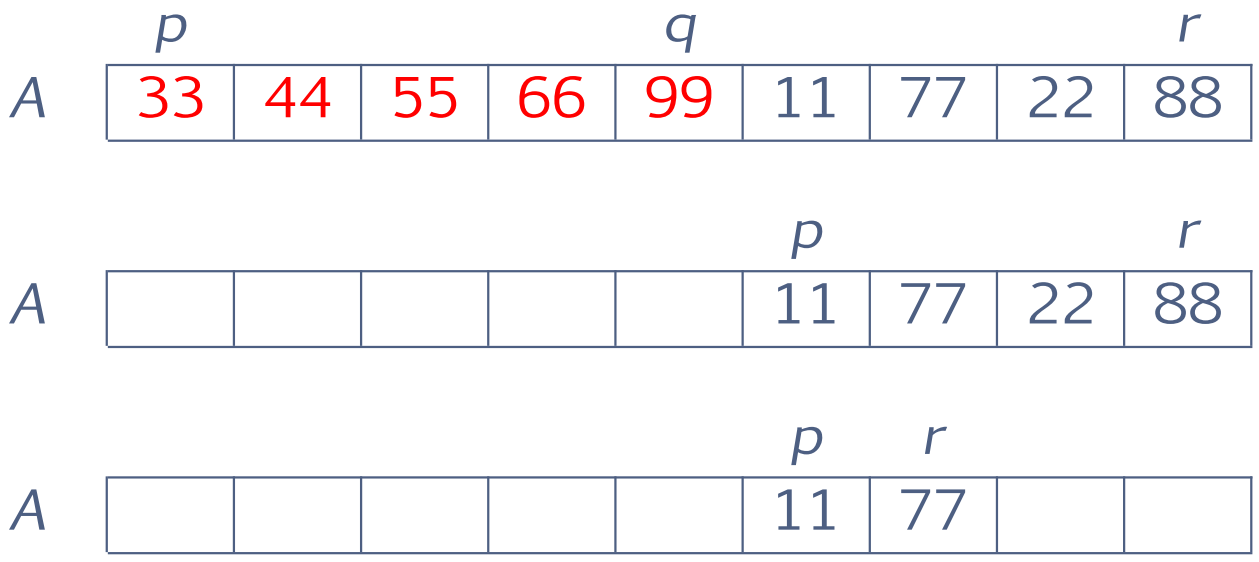
Ordenando por intercalação

Ordenando por intercalação

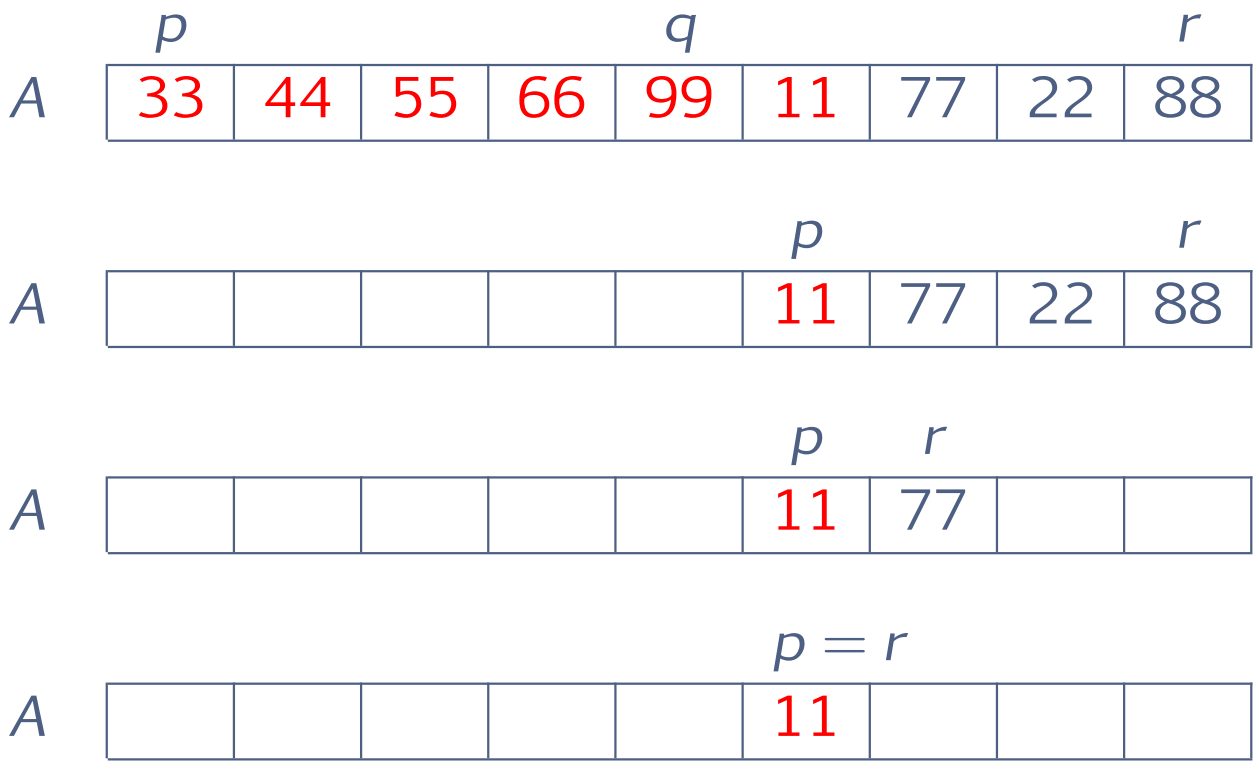
Ordenando por intercalação



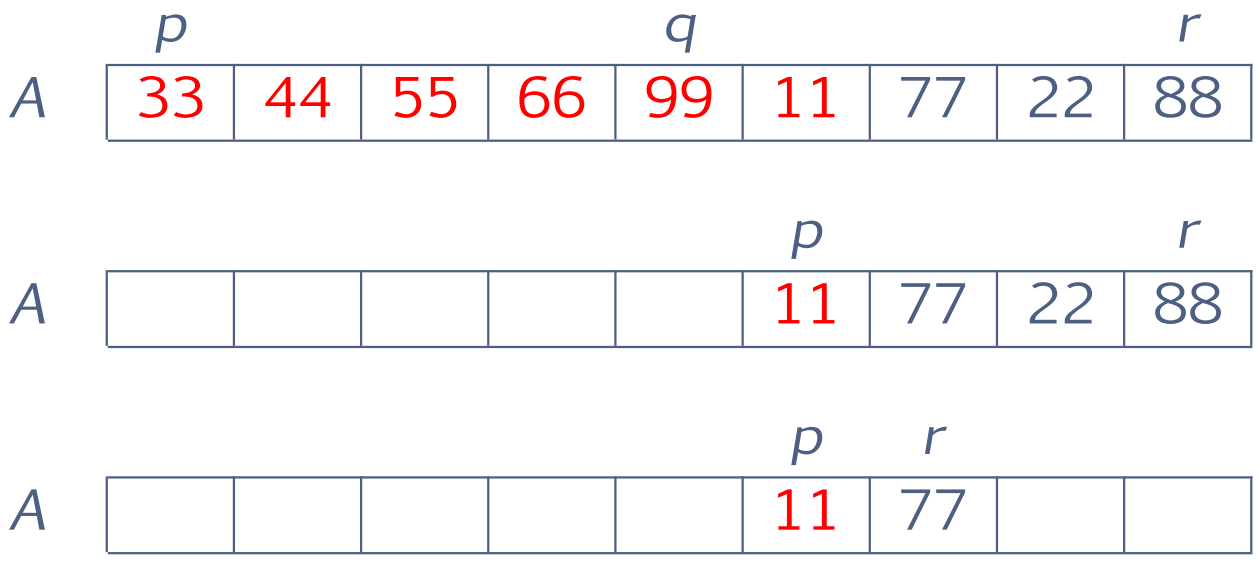
Ordenando por intercalação



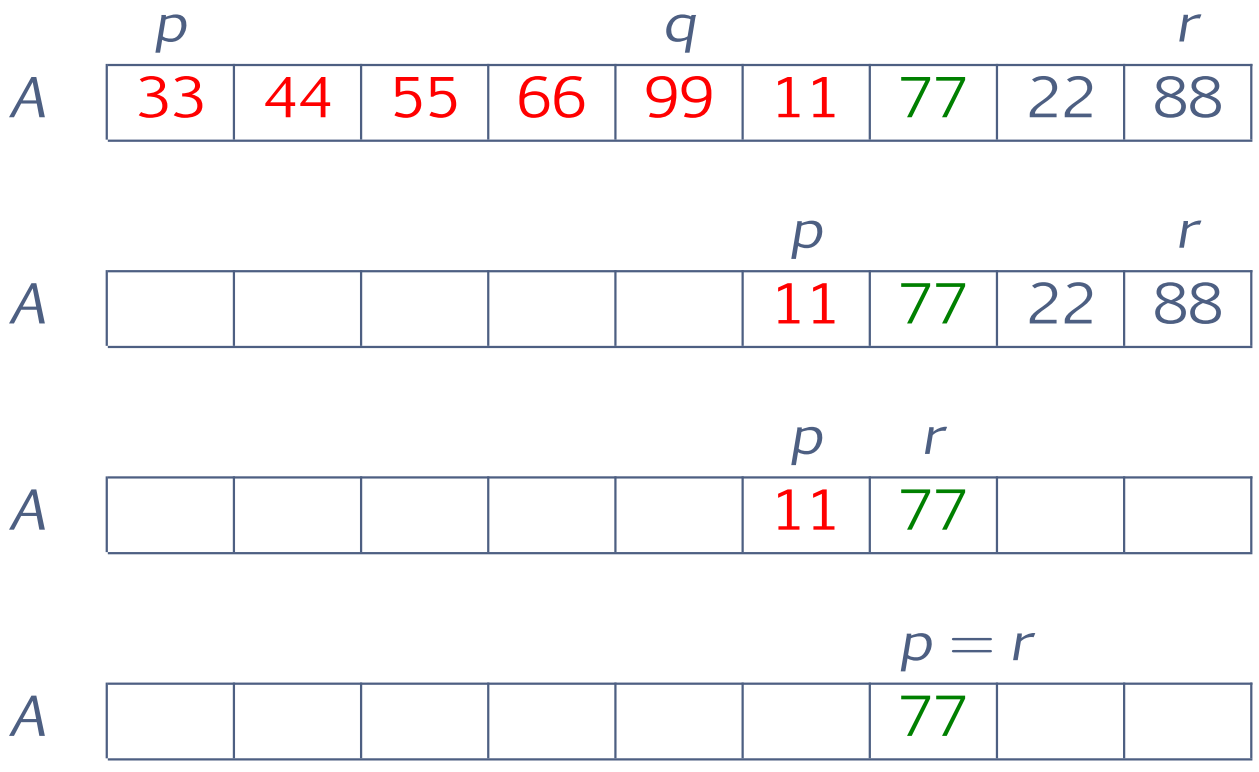
Ordenando por intercalação



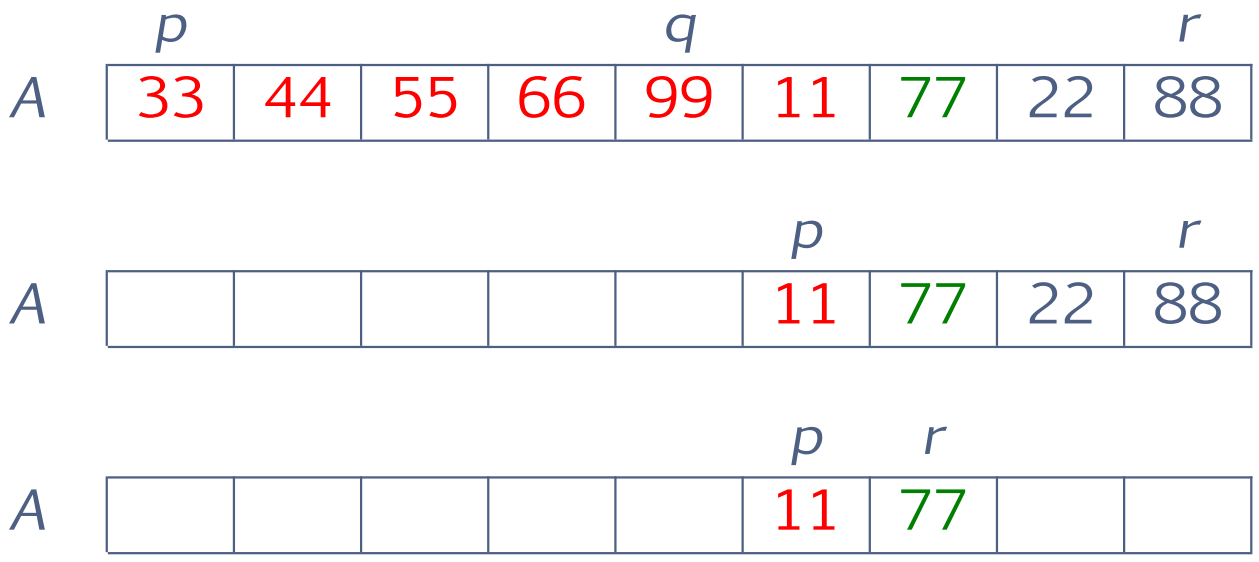
Ordenando por intercalação



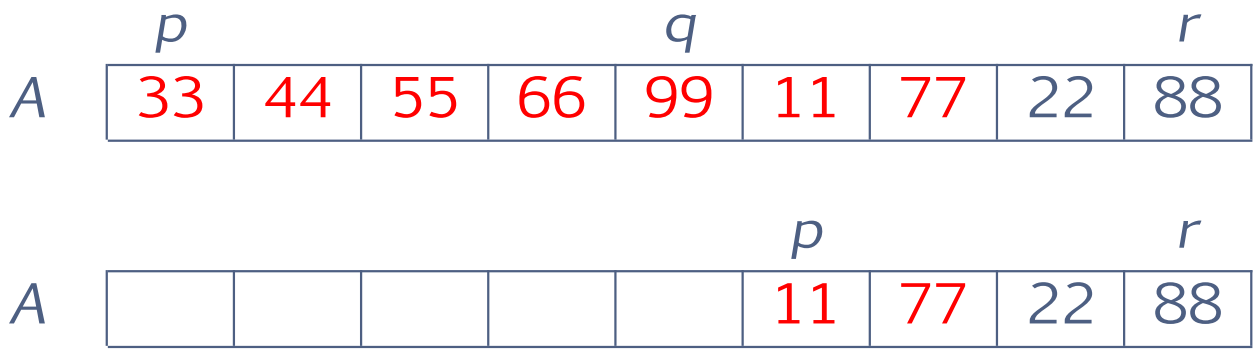
Ordenando por intercalação



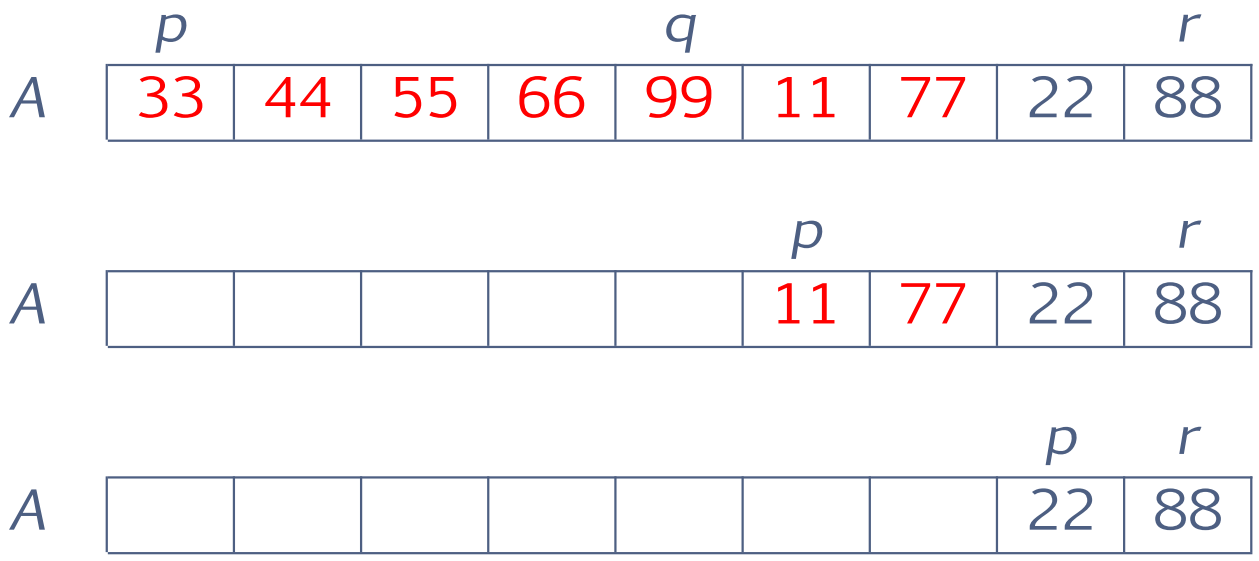
Ordenando por intercalação



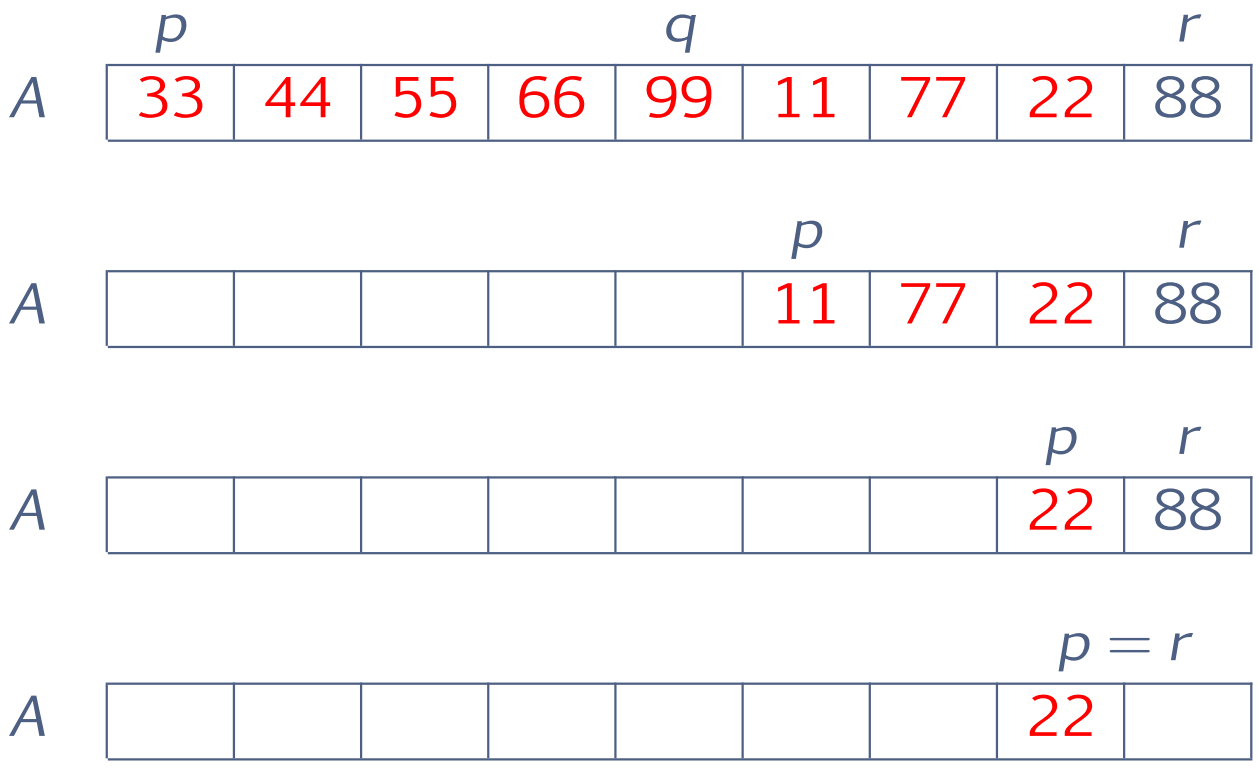
Ordenando por intercalação



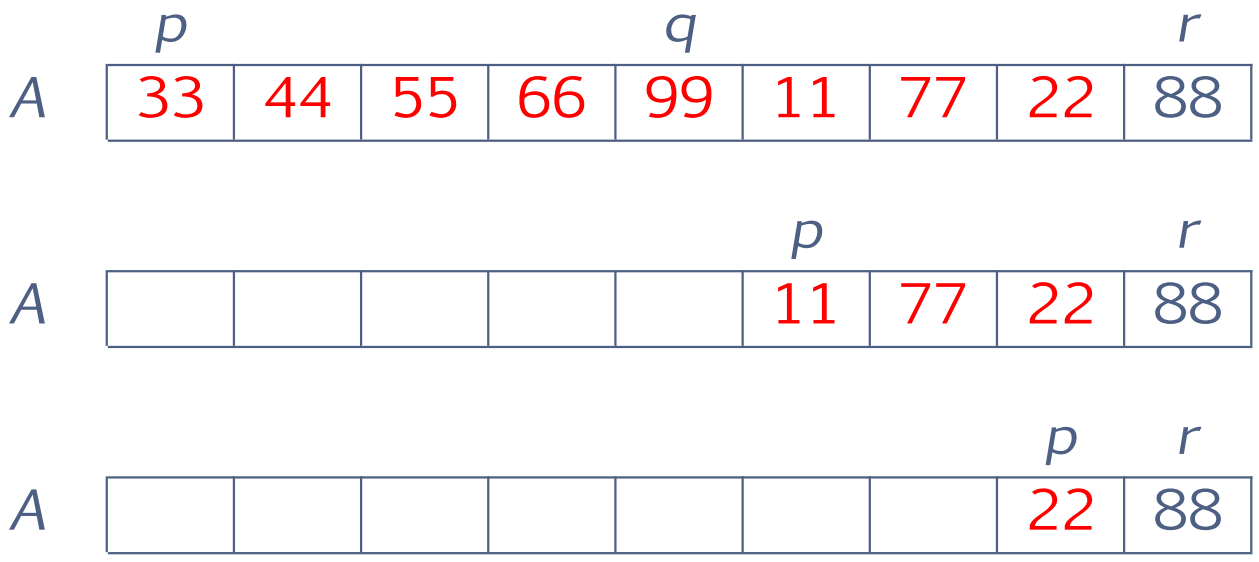
Ordenando por intercalação



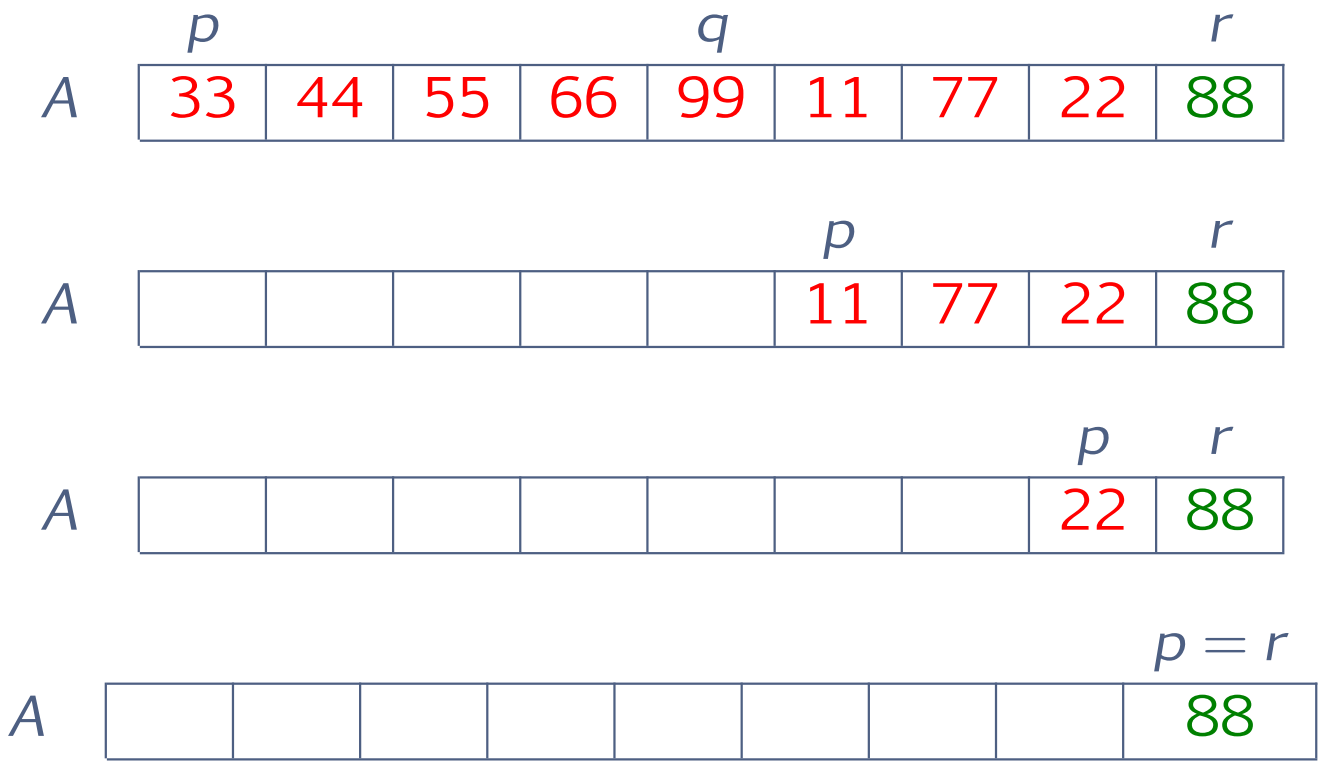
Ordenando por intercalação



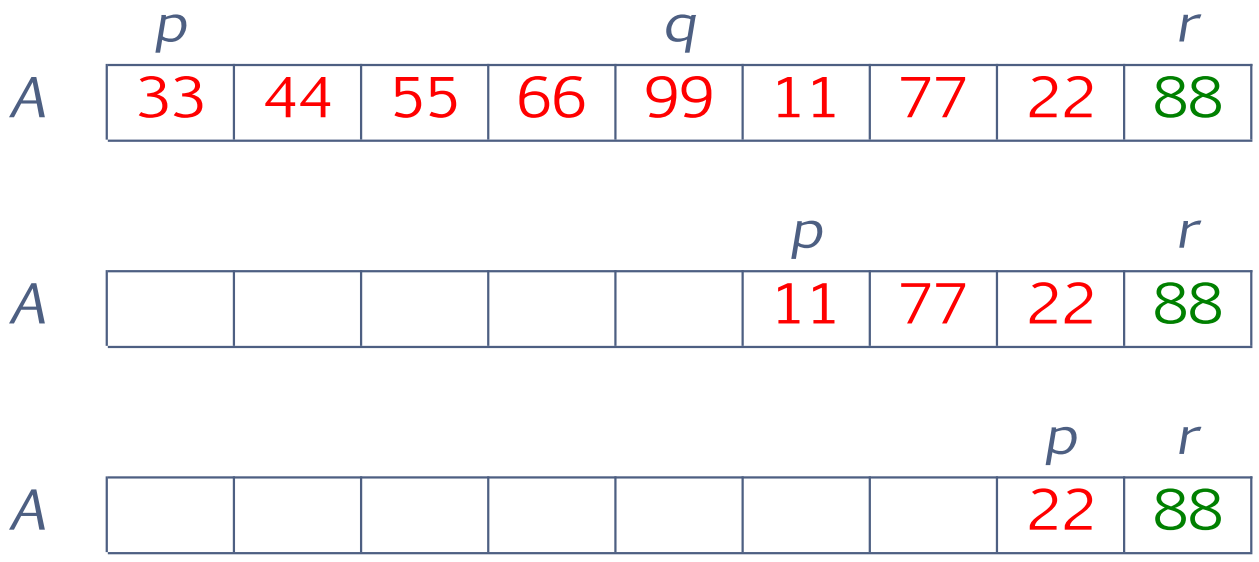
Ordenando por intercalação



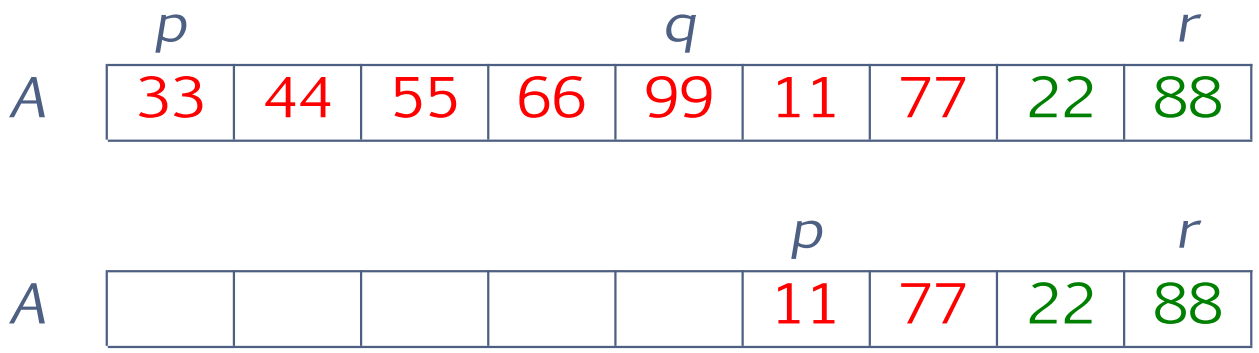
Ordenando por intercalação



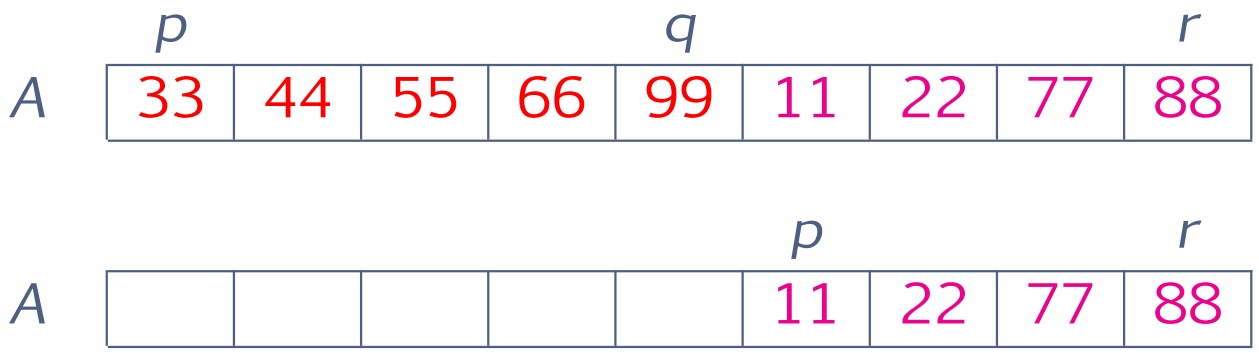
Ordenando por intercalação



Ordenando por intercalação

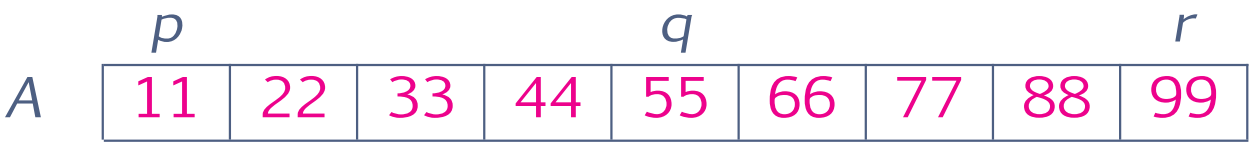


Ordenando por intercalação



Ordenando por intercalação

Ordenando por intercalação



Ordenando por intercalação

	p				q				r
A	11	22	33	44	55	66	77	88	99

Complexidade de Merge-Sort

- Tamanho da entrada: $n = r - p + 1$
- Seja $T(n)$ o número de instruções executadas no pior caso

Complexidade de MERGE-SORT

```
MERGE-SORT(A, p, r)
1   se p < r
2       então q ← ⌊(p + r)/2⌋
3             MERGE-SORT(A, p, q)
4             MERGE-SORT(A, q + 1, r)
5             INTERCALA(A, p, q, r)
```

- Tamanho da entrada: $n = r - p + 1$
- Seja $T(n)$ o número de instruções executadas no pior caso

Complexidade de MERGE-SORT

```
MERGE-SORT(A, p, r)
1   se p < r
2       então q ← ⌊(p + r)/2⌋
3             MERGE-SORT(A, p, q)
4             MERGE-SORT(A, q + 1, r)
5             INTERCALA(A, p, q, r)
```

- Tamanho da entrada: $n = r - p + 1$
- Seja $T(n)$ o número de instruções executadas no pior caso

Complexidade de Merge-Sort

```
MERGE-SORT(A, p, r)
1   se p < r
2       então q ← ⌊(p + r)/2⌋
3             MERGE-SORT(A, p, q)
4             MERGE-SORT(A, q + 1, r)
5             INTERCALA(A, p, q, r)
```

- Tamanho da entrada: $n = r - p + 1$
- Seja $T(n)$ o número de instruções executadas no pior caso

Complexidade de MERGE-SORT

```
MERGE-SORT(A, p, r)
1   se p < r
2       então q ← ⌊(p + r)/2⌋
3             MERGE-SORT(A, p, q)
4             MERGE-SORT(A, q + 1, r)
5             INTERCALA(A, p, q, r)
```

Linha	Tempo
1	?
2	?
3	?
4	?
5	?

$$T(n) = ?$$

Complexidade de MERGE-SORT

```
MERGE-SORT(A, p, r)
1   se p < r
2       então q ← ⌊(p + r)/2⌋
3             MERGE-SORT(A, p, q)
4             MERGE-SORT(A, q + 1, r)
5             INTERCALA(A, p, q, r)
```

Linha	Tempo
1	$\Theta(1)$
2	$\Theta(1)$
3	$T(\lceil n/2 \rceil)$
4	$T(\lfloor n/2 \rfloor)$
5	$\Theta(n)$

$$T(n) = ?$$

Complexidade de MERGE-SORT

```
MERGE-SORT(A, p, r)
1   se p < r
2       então q ← ⌊(p + r)/2⌋
3             MERGE-SORT(A, p, q)
4             MERGE-SORT(A, q + 1, r)
5             INTERCALA(A, p, q, r)
```

Linha	Tempo
1	$\Theta(1)$
2	$\Theta(1)$
3	$T(\lceil n/2 \rceil)$
4	$T(\lfloor n/2 \rfloor)$
5	$\Theta(n)$

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) + \Theta(2)$$

Recorrência

O tempo de MERGE-SORT é dado pela fórmula

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{se } n = 2, 3, 4, \ldots \end{cases}$$

Obtemos uma fórmula de recorrência

- é a descrição de uma função em termos de si mesma.
- o tempo de um algoritmo recursivo costuma se descrito por uma recorrência

Mas queremos uma fórmula fechada

- Nesse caso, $T(n) = \Theta(n \log n)$
- Aprenderemos a resolver recorrências depois

Recorrência

O tempo de MERGE-SORT é dado pela fórmula

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{se } n = 2, 3, 4, \ldots \end{cases}$$

Obtemos uma fórmula de recorrência

- é a descrição de uma função em termos de si mesma.
- o tempo de um algoritmo recursivo costuma se descrito por uma recorrência

Mas queremos uma fórmula fechada

- Nesse caso, $T(n) = \Theta(n \log n)$
- Aprenderemos a resolver recorrências depois

Recorrência

O tempo de MERGE-SORT é dado pela fórmula

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{se } n = 2, 3, 4, \ldots \end{cases}$$

Obtemos uma fórmula de recorrência

- é a descrição de uma função em termos de si mesma.
- o tempo de um algoritmo recursivo costuma se descrito por uma recorrência

Mas queremos uma fórmula fechada

- Nesse caso, $T(n) = \Theta(n \log n)$
- Aprenderemos a resolver recorrências depois

Recorrência

O tempo de MERGE-SORT é dado pela fórmula

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{se } n = 2, 3, 4, \ldots \end{cases}$$

Obtemos uma fórmula de recorrência

- é a descrição de uma função em termos de si mesma.
- o tempo de um algoritmo recursivo costuma se descrito por uma recorrência

Mas queremos uma fórmula fechada

- Nesse caso, $T(n) = \Theta(n \log n)$
- Aprenderemos a resolver recorrências depois

Recorrência

O tempo de MERGE-SORT é dado pela fórmula

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{se } n = 2, 3, 4, \ldots \end{cases}$$

Obtemos uma fórmula de recorrência

- é a descrição de uma função em termos de si mesma.
- o tempo de um algoritmo recursivo costuma se descrito por uma recorrência

Mas queremos uma fórmula fechada

- Nesse caso, $T(n) = \Theta(n \log n)$
- Aprenderemos a resolver recorrências depois

Recorrência

O tempo de MERGE-SORT é dado pela fórmula

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{se } n = 2, 3, 4, \ldots \end{cases}$$

Obtemos uma fórmula de recorrência

- é a descrição de uma função em termos de si mesma.
- o tempo de um algoritmo recursivo costuma se descrito por uma recorrência

Mas queremos uma fórmula fechada

- Nesse caso, $T(n) = \Theta(n \log n)$
- Aprenderemos a resolver recorrências depois

Recorrência

O tempo de MERGE-SORT é dado pela fórmula

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{se } n = 2, 3, 4, \ldots \end{cases}$$

Obtemos uma fórmula de recorrência

- é a descrição de uma função em termos de si mesma.
- o tempo de um algoritmo recursivo costuma se descrito por uma recorrência

Mas queremos uma fórmula fechada

- Nesse caso, $T(n) = \Theta(n \log n)$
- Aprenderemos a resolver recorrências depois

Recorrência

O tempo de MERGE-SORT é dado pela fórmula

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{se } n = 2, 3, 4, \ldots \end{cases}$$

Obtemos uma fórmula de recorrência

- é a descrição de uma função em termos de si mesma.
- o tempo de um algoritmo recursivo costuma se descrito por uma recorrência

Mas queremos uma fórmula fechada

- Nesse caso, $T(n) = \Theta(n \log n)$
- Aprenderemos a resolver recorrências depois

Projetando algoritmos

- Entendendo a importância de algoritmos eficientes

Algoritmos bons e ruins

Que diferença faz ter um algoritmo $\Theta(n^2)$ ou $\Theta(n \log n)$?

Suponha que queremos ordenar um vetor de n elementos

1. Usando um computador A com velocidade $1\,GHz$ e com um algoritmo que executa $2n^2$ instruções.
2. Usando um computador B com velocidade $10\,MHz$ e com um algoritmo que executa $50n \log n$ instruções.

Comparando as implementações:

- O computador A é 100 vezes mais rápido do que B.
- A constante multiplicativa do segundo é bem maior:
  - pode ser devido a uma linguagem de mais alto nível
  - ou devido a um programador menos experiente

Algoritmos bons e ruins

Que diferença faz ter um algoritmo $\Theta(n^2)$ ou $\Theta(n \log n)$?

Suponha que queremos ordenar um vetor de n elementos

1. Usando um computador A com velocidade $1GHz$ e com um algoritmo que executa $2n^2$ instruções.
2. Usando um computador B com velocidade $10MHz$ e com um algoritmo que executa $50n \log n$ instruções.

Comparando as implementações:

- O computador A é 100 vezes mais rápido do que B.
- A constante multiplicativa do segundo é bem maior:
  - pode ser devido a uma linguagem de mais alto nível
  - ou devido a um programador menos experiente

Algoritmos bons e ruins

Que diferença faz ter um algoritmo $\Theta(n^2)$ ou $\Theta(n \log n)$?

Suponha que queremos ordenar um vetor de n elementos

1. Usando um computador A com velocidade $1GHz$ e com um algoritmo que executa $2n^2$ instruções.
2. Usando um computador B com velocidade $10MHz$ e com um algoritmo que executa $50n \log n$ instruções.

Comparando as implementações:

- O computador A é 100 vezes mais rápido do que B.
- A constante multiplicativa do segundo é bem maior:
  - pode ser devido a uma linguagem de mais alto nível
  - ou devido a um programador menos experiente

Algoritmos bons e ruins

Que diferença faz ter um algoritmo $\Theta(n^2)$ ou $\Theta(n \log n)$?

Suponha que queremos ordenar um vetor de n elementos

1. Usando um computador A com velocidade $1GHz$ e com um algoritmo que executa $2n^2$ instruções.
2. Usando um computador B com velocidade $10MHz$ e com um algoritmo que executa $50n \log n$ instruções.

Comparando as implementações:

- O computador A é 100 vezes mais rápido do que B.
- A constante multiplicativa do segundo é bem maior:
  - pode ser devido a uma linguagem de mais alto nível
  - ou devido a um programador menos experiente

Algoritmos bons e ruins

Que diferença faz ter um algoritmo $\Theta(n^2)$ ou $\Theta(n \log n)$?

Suponha que queremos ordenar um vetor de n elementos

1. Usando um computador A com velocidade $1GHz$ e com um algoritmo que executa $2n^2$ instruções.
2. Usando um computador B com velocidade $10MHz$ e com um algoritmo que executa $50n \log n$ instruções.

Comparando as implementações:

- O computador A é 100 vezes mais rápido do que B.
- A constante multiplicativa do segundo é bem maior:
  - pode ser devido a uma linguagem de mais alto nível
  - ou devido a um programador menos experiente

Algoritmos bons e ruins

Que diferença faz ter um algoritmo $\Theta(n^2)$ ou $\Theta(n \log n)$?

Suponha que queremos ordenar um vetor de n elementos

1. Usando um computador A com velocidade $1GHz$ e com um algoritmo que executa $2n^2$ instruções.
2. Usando um computador B com velocidade $10MHz$ e com um algoritmo que executa $50n \log n$ instruções.

Comparando as implementações:

- O computador A é 100 vezes mais rápido do que B.
- A constante multiplicativa do segundo é bem maior:
  - pode ser devido a uma linguagem de mais alto nível
  - ou devido a um programador menos experiente

Algoritmos bons e ruins

Que diferença faz ter um algoritmo $\Theta(n^2)$ ou $\Theta(n \log n)$?

Suponha que queremos ordenar um vetor de n elementos

1. Usando um computador A com velocidade $1GHz$ e com um algoritmo que executa $2n^2$ instruções.
2. Usando um computador B com velocidade $10MHz$ e com um algoritmo que executa $50n \log n$ instruções.

Comparando as implementações:

- O computador A é 100 vezes mais rápido do que B.
- A constante multiplicativa do segundo é bem maior:
  - pode ser devido a uma linguagem de mais alto nível
  - ou devido a um programador menos experiente

Algoritmos bons e ruins

Que diferença faz ter um algoritmo $\Theta(n^2)$ ou $\Theta(n \log n)$?

Suponha que queremos ordenar um vetor de n elementos

1. Usando um computador A com velocidade $1\,GHz$ e com um algoritmo que executa $2n^2$ instruções.
2. Usando um computador B com velocidade $10\,MHz$ e com um algoritmo que executa $50n \log n$ instruções.

Comparando as implementações:

- O computador A é 100 vezes mais rápido do que B.
- A constante multiplicativa do segundo é bem maior:
  - pode ser devido a uma linguagem de mais alto nível
  - ou devido a um programador menos experiente

Algoritmos bons e ruins

Que diferença faz ter um algoritmo $\Theta(n^2)$ ou $\Theta(n \log n)$?

Suponha que queremos ordenar um vetor de n elementos

1. Usando um computador A com velocidade $1\,GHz$ e com um algoritmo que executa $2n^2$ instruções.
2. Usando um computador B com velocidade $10\,MHz$ e com um algoritmo que executa $50n \log n$ instruções.

Comparando as implementações:

- O computador A é 100 vezes mais rápido do que B.
- A constante multiplicativa do segundo é bem maior:
  - pode ser devido a uma linguagem de mais alto nível
  - ou devido a um programador menos experiente

Algoritmos bons e ruins (cont)

Que implementação ordena um milhão de elementos primeiro?

1. A máquina A leva

$$\frac{2 \cdot (10^6)^2 \text{instruções}}{10^9 \text{instruções /segundo}} \approx 2000 \text{segundos}$$

2. A máquina B leva

$$\frac{50 \cdot (10^6 \log 10^6) \text{instruções}}{10^7 \text{instruções /segundo}} \approx 100 \text{segundos}$$

- A máquina B foi vinte vezes mais rápida do que A
- Se fossem 10 milhões de elementos, a razão seria de 2,3 dias para 20 minutos

Algoritmos bons e ruins (cont)

Que implementação ordena um milhão de elementos primeiro?

1. A máquina A leva

$$\frac{2 \cdot (10^6)^2 \text{instruções}}{10^9 \text{instruções /segundo}} \approx 2000 \text{segundos}$$

2. A máquina B leva

$$\frac{50 \cdot (10^6 \log 10^6) \text{instruções}}{10^7 \text{instruções /segundo}} \approx 100 \text{segundos}$$

- A máquina B foi vinte vezes mais rápida do que A
- Se fossem 10 milhões de elementos, a razão seria de 2,3 dias para 20 minutos

Algoritmos bons e ruins (cont)

Que implementação ordena um milhão de elementos primeiro?

1. A máquina A leva

$$\frac{2 \cdot (10^6)^2 \text{instruções}}{10^9 \text{instruções /segundo}} \approx 2000 \text{segundos}$$

2. A máquina B leva

$$\frac{50 \cdot (10^6 \log 10^6) \text{instruções}}{10^7 \text{instruções /segundo}} \approx 100 \text{segundos}$$

- A máquina B foi vinte vezes mais rápida do que A
- Se fossem 10 milhões de elementos, a razão seria de 2,3 dias para 20 minutos

Algoritmos bons e ruins (cont)

Que implementação ordena um milhão de elementos primeiro?

1. A máquina A leva

$$\frac{2 \cdot (10^6)^2 \text{instruções}}{10^9 \text{instruções /segundo}} \approx 2000 \text{segundos}$$

2. A máquina B leva

$$\frac{50 \cdot (10^6 \log 10^6) \text{instruções}}{10^7 \text{instruções /segundo}} \approx 100 \text{segundos}$$

- A máquina B foi vinte vezes mais rápida do que A
- Se fossem 10 milhões de elementos, a razão seria de 2,3 dias para 20 minutos

Algoritmos bons e ruins (cont)

Que implementação ordena um milhão de elementos primeiro?

1. A máquina A leva

$$\frac{2 \cdot (10^6)^2 \text{instruções}}{10^9 \text{instruções /segundo}} \approx 2000 \text{segundos}$$

2. A máquina B leva

$$\frac{50 \cdot (10^6 \log 10^6) \text{instruções}}{10^7 \text{instruções /segundo}} \approx 100 \text{segundos}$$

- A máquina B foi vinte vezes mais rápida do que A
- Se fossem 10 milhões de elementos, a razão seria de 2,3 dias para 20 minutos

Algoritmos e tecnologia

E se usarmos um supercomputador?

$f(n)$	Computador atual	100×mais rápido	1000×mais rápido
n	N_1	$100N_1$	$1000N_1$
n^2	N_2	$10N_2$	$31{,}6N_2$
n^3	N_3	$4{,}64N_3$	$10N_3$
n^5	N_4	$2{,}5N_4$	$3{,}98N_4$
2^n	N_5	$N_5 + 6{,}64$	$N_5 + 9{,}97$
3^n	N_6	$N_6 + 4{,}19$	$N_6 + 6{,}29$

Se só tivermos um algoritmo ruim, não iremos resolver problemas muito maiores.

Algoritmos e tecnologia

E se usarmos um supercomputador?

$f(n)$	Computador atual	100×mais rápido	1000×mais rápido
n	N_1	$100N_1$	$1000N_1$
n^2	N_2	$10N_2$	$31{,}6N_2$
n^3	N_3	$4{,}64N_3$	$10N_3$
n^5	N_4	$2{,}5N_4$	$3{,}98N_4$
2^n	N_5	$N_5 + 6{,}64$	$N_5 + 9{,}97$
3^n	N_6	$N_6 + 4{,}19$	$N_6 + 6{,}29$

Se só tivermos um algoritmo ruim, não iremos resolver problemas muito maiores.

Algoritmos e tecnologia

E se usarmos um supercomputador?

$f(n)$	Computador atual	100×mais rápido	1000×mais rápido
n	N_1	$100N_1$	$1000N_1$
n^2	N_2	$10N_2$	$31{,}6N_2$
n^3	N_3	$4{,}64N_3$	$10N_3$
n^5	N_4	$2{,}5N_4$	$3{,}98N_4$
2^n	N_5	$N_5 + 6{,}64$	$N_5 + 9{,}97$
3^n	N_6	$N_6 + 4{,}19$	$N_6 + 6{,}29$

Se só tivermos um algoritmo ruim, não iremos resolver problemas muito maiores.

Algoritmos e tecnologia

E se usarmos um supercomputador?

$f(n)$	Computador atual	100×mais rápido	1000×mais rápido
n	N_1	$100N_1$	$1000N_1$
n^2	N_2	$10N_2$	$31{,}6N_2$
n^3	N_3	$4{,}64N_3$	$10N_3$
n^5	N_4	$2{,}5N_4$	$3{,}98N_4$
2^n	N_5	$N_5 + 6{,}64$	$N_5 + 9{,}97$
3^n	N_6	$N_6 + 4{,}19$	$N_6 + 6{,}29$

Se só tivermos um algoritmo ruim, não iremos resolver problemas muito maiores.

Algoritmos e tecnologia

E se usarmos um supercomputador?

$f(n)$	Computador atual	100×mais rápido	1000×mais rápido
n	N_1	$100N_1$	$1000N_1$
n^2	N_2	$10N_2$	$31{,}6N_2$
n^3	N_3	$4{,}64N_3$	$10N_3$
n^5	N_4	$2{,}5N_4$	$3{,}98N_4$
2^n	N_5	$N_5 + 6{,}64$	$N_5 + 9{,}97$
3^n	N_6	$N_6 + 4{,}19$	$N_6 + 6{,}29$

Se só tivermos um algoritmo ruim, não iremos resolver problemas muito maiores.

Algoritmos e tecnologia

E se usarmos um supercomputador?

$f(n)$	Computador atual	100×mais rápido	1000×mais rápido
n	N_1	$100N_1$	$1000N_1$
n^2	N_2	$10N_2$	$31{,}6N_2$
n^3	N_3	$4{,}64N_3$	$10N_3$
n^5	N_4	$2{,}5N_4$	$3{,}98N_4$
2^n	N_5	$N_5 + 6{,}64$	$N_5 + 9{,}97$
3^n	N_6	$N_6 + 4{,}19$	$N_6 + 6{,}29$

Se só tivermos um algoritmo ruim, não iremos resolver problemas muito maiores.

Algoritmos e tecnologia

E se usarmos um supercomputador?

$f(n)$	Computador atual	100×mais rápido	1000×mais rápido
n	N_1	$100N_1$	$1000N_1$
n^2	N_2	$10N_2$	$31{,}6N_2$
n^3	N_3	$4{,}64N_3$	$10N_3$
n^5	N_4	$2{,}5N_4$	$3{,}98N_4$
2^n	N_5	$N_5 + 6{,}64$	$N_5 + 9{,}97$
3^n	N_6	$N_6 + 4{,}19$	$N_6 + 6{,}29$

Se só tivermos um algoritmo ruim, não iremos resolver problemas muito maiores.

Algoritmos e tecnologia

E se usarmos um supercomputador?

$f(n)$	Computador atual	100×mais rápido	1000×mais rápido
n	N_1	$100N_1$	$1000N_1$
n^2	N_2	$10N_2$	$31{,}6N_2$
n^3	N_3	$4{,}64N_3$	$10N_3$
n^5	N_4	$2{,}5N_4$	$3{,}98N_4$
2^n	N_5	$N_5 + 6{,}64$	$N_5 + 9{,}97$
3^n	N_6	$N_6 + 4{,}19$	$N_6 + 6{,}29$

Se só tivermos um algoritmo ruim, não iremos resolver problemas muito maiores.

Algoritmos e tecnologia

E se usarmos um supercomputador?

$f(n)$	Computador atual	100×mais rápido	1000×mais rápido
n	N_1	$100N_1$	$1000N_1$
n^2	N_2	$10N_2$	$31{,}6N_2$
n^3	N_3	$4{,}64N_3$	$10N_3$
n^5	N_4	$2{,}5N_4$	$3{,}98N_4$
2^n	N_5	$N_5 + 6{,}64$	$N_5 + 9{,}97$
3^n	N_6	$N_6 + 4{,}19$	$N_6 + 6{,}29$

Se só tivermos um algoritmo ruim, não iremos resolver problemas muito maiores.

Conclusões

Algumas conclusões:

- Projetar algoritmos melhores pode levar a ganhos extraordinários de desempenho.
- Isso é tão importante quanto o projeto de hardware.
- O ganho obtido ao melhorar a complexidade de um algoritmo não poderia ser obtida simplesmente com o avanço da tecnologia.
- Queremos estudar principalmente algoritmos fundamentais, que produzem avanços em outras componentes básicas das aplicações (pense nos compiladores, buscadores na internet, classificadores, etc).

Conclusões

Algumas conclusões:

- Projetar algoritmos melhores pode levar a ganhos extraordinários de desempenho.
- Isso é tão importante quanto o projeto de hardware.
- O ganho obtido ao melhorar a complexidade de um algoritmo não poderia ser obtida simplesmente com o avanço da tecnologia.
- Queremos estudar principalmente algoritmos fundamentais, que produzem avanços em outras componentes básicas das aplicações (pense nos compiladores, buscadores na internet, classificadores, etc).

Conclusões

Algumas conclusões:

- Projetar algoritmos melhores pode levar a ganhos extraordinários de desempenho.
- Isso é tão importante quanto o projeto de hardware.
- O ganho obtido ao melhorar a complexidade de um algoritmo não poderia ser obtida simplesmente com o avanço da tecnologia.
- Queremos estudar principalmente algoritmos fundamentais, que produzem avanços em outras componentes básicas das aplicações (pense nos compiladores, buscadores na internet, classificadores, etc).

Conclusões

Algumas conclusões:

- Projetar algoritmos melhores pode levar a ganhos extraordinários de desempenho.
- Isso é tão importante quanto o projeto de hardware.
- O ganho obtido ao melhorar a complexidade de um algoritmo não poderia ser obtida simplesmente com o avanço da tecnologia.
- Queremos estudar principalmente algoritmos fundamentais, que produzem avanços em outras componentes básicas das aplicações (pense nos compiladores, buscadores na internet, classificadores, etc).

Conclusões

Algumas conclusões:

- Projetar algoritmos melhores pode levar a ganhos extraordinários de desempenho.
- Isso é tão importante quanto o projeto de hardware.
- O ganho obtido ao melhorar a complexidade de um algoritmo não poderia ser obtida simplesmente com o avanço da tecnologia.
- Queremos estudar principalmente algoritmos fundamentais, que produzem avanços em outras componentes básicas das aplicações (pense nos compiladores, buscadores na internet, classificadores, etc).