SIN5013 - Análise de Algoritmos e Estruturas de Dados

Apresentação da Disciplina e Introdução

Prof. Flávio Luiz Coutinho
flcoutinho@usp.br

Objetivos da disciplina

Familiarizar os/as estudantes com as várias **estruturas de dados**, **técnicas de programação** e a **análise da complexidade assintótica de algoritmos** a fim de que possam contar com esses recursos no desenvolvimento de suas atividades em Sistemas de Informação.

Conteúdo

- Análise de algoritmos
- Notação assintótica
- Recorrências
- Técnicas de Programação
- Algoritmos de ordenação
- Listas
- Filas
- Pilhas
- Árvores
- Grafos

Avaliação

- Duas provas: P1 e P2
- Média Final: MF = P1 * 0.4 + P2 * 0.6
- Conceitos:

MF >= 9.0 -----> A

7.0 <= MF < 9.0 -----> B

5.0 <= MF < 7.0 -----> C

MF < 5.0 -----> R

Análise de algoritmos

Prever recursos computacionais necessários para a execução de um programa, ou parte(s) de um programa:

- Tempo
- Memória
- Outros (quantidade de acessos a disco, dados trafegados em rede em um sistema distribuído, etc)

Análise de algoritmos

Prever recursos computacionais necessários para a execução de um programa, ou parte(s) de um programa:

- Tempo*
- Memória
- Outros (quantidade de acessos a disco, dados trafegados em rede em um sistema distribuído, etc)

Análise de algoritmos

Prever recursos computacionais necessários para a execução de um programa, ou parte(s) de um programa:

- Tempo*
- Memória
- Outros (quantidade de acessos a disco, dados trafegados em rede em um sistema distribuído, etc)

Dados dois algoritmos A e B, que realizam a mesma tarefa, o processo de análise permite demonstrar formalmente qual o algoritmo mais adequado.

Não é mais fácil testar os algoritmos candidatos?

Não, pois...

- Precisamos implementar todos os algoritmos candidatos.
- Testar para “entradas pequenas” pode levar a conclusões incorretas.
- Testar para “entradas grandes” pode ser demorado demais.
- Há muitas variáveis sobre as quais não temos tanto controle: arquitetura, memória disponível, sistema operacional, linguagem de programação usada, existência de outros programas dividindo os mesmo recursos computacionais de forma simultânea.

Não é mais fácil testar os algoritmos candidatos?

Não, pois...

- Precisamos implementar todos os algoritmos candidatos.
- Testar para “entradas pequenas” pode levar a conclusões incorretas.
- Testar para “entradas grandes” pode ser demorado demais.
- Há muitas variáveis sobre as quais não temos tanto controle: arquitetura, memória disponível, sistema operacional, linguagem de programação usada, existência de outros programas dividindo os mesmo recursos computacionais de forma simultânea.

Testes não dão como resultado uma função matemática que prevê o consumo de um certo recurso para entradas de tamanho arbitrário.

Exemplo: ordenar uma sequência de números

Problema: dada uma sequência de valores numéricos, gerar uma permutação dos valores de modo que todo valor na sequência resultante seja menor ou igual ao valor que o sucede.

Exemplo:

- Sequência de entrada: 11, 7, 3, 28, 19, 14, 5, 21
- Sequência resultante: 3, 5, 7, 11, 14, 19, 21, 28

Vamos discutir dois algoritmos diferentes para realizar tal tarefa (foco no princípio de funcionamento, sem nos preocuparmos tanto com implementação).

Algoritmo 1: *selection sort*

- Fazer uma varredura na sequência para encontrar o menor valor.
- Mover o menor valor encontrado para a sequência resposta.
- Repetir os passos anteriores até não restar valores na sequência original.
- Exemplo das duas primeiras varreduras:

{ 11, 7, 3, 28, 19, 14, 5, 21 }	{ }	(encontrar menor)
{ 11, 7, 28, 19, 14, 5, 21 }	{ 3 }	(mover menor)
{ 11, 7, 28, 19, 14, 5, 21 }	{ 3 }	(encontrar menor)
{ 11, 7, 28, 19, 14, 21 }	{ 3, 5 }	(mover menor)

Algoritmo 2: *merge sort*

- Definir subsequências unitárias para cada valor da sequência:

{ 11 } { 7 } { 3 } { 28 } { 19 } { 14 } { 5 } { 21 }

Algoritmo 2: *merge sort*

- Definir subsequências unitárias para cada valor da sequência:

  { 11 } { 7 } { 3 } { 28 } { 19 } { 14 } { 5 } { 21 }

- Juntar, duas a duas, as subsequências, tomando o cuidado de gerar novas subsequências ordenadas:

  { 7, 11 } { 3, 28 } { 14, 19 } { 5, 21 }   (como juntar de forma esperta?)

Algoritmo 2: *merge sort*

- Definir subsequências unitárias para cada valor da sequência:

  { 11 } { 7 } { 3 } { 28 } { 19 } { 14 } { 5 } { 21 }

- Juntar, duas a duas, as subsequências, tomando o cuidado de gerar novas subsequências ordenadas:

  { 7, 11 } { 3, 28 } { 14, 19 } { 5, 21 }   (como juntar de forma esperta?)

- Repetir o processo até que todos os valores estejam reunidos novamente em uma única sequência. Executando o passo mais uma vez, teremos:

  { 3, 7, 11, 28 } { 5, 14, 19, 21 }   (precisa repetir mais uma vez)

Qual algoritmo é melhor?

- *Merge sort* é certamente menos intuitivo que o *selection sort*!!!
- Mas qual é o melhor??? Depende do que definimos como melhor...
- O melhor algoritmo é aquele que realiza menos comparações entre valores:
    - *Selection sort* faz comparações no processo de varredura.
    - *Merge sort* faz comparações no processo de junção.

- Mas qual dos algoritmos ordena a sequência fazendo menos comparações???

- Simulação.

Qual algoritmo é melhor?

- *Selection sort*: 28 comparações.
- *Merge sort*: 17 comparações.
- *Merge sort* é melhor? Provavelmente sim... mas quão melhor???
    - faz sempre 11 comparações a menos?
    - faz sempre 60.7% das comparações que o algoritmo 1 faria?
- Ter o resultado de teste para um cenário específico (uma sequência particular de 8 elementos) não é muito informativo...

Qual algoritmo é melhor?

- Precisamos generalizar o resultado para sequências de tamanho arbitrário.
- Queremos determinar f(n) que dá o número de comparações necessárias para ordenar uma sequência de tamanho n.
- Olhando a simulação com mais atenção, somos capazes de fazer tal generalização.

- *Selection sort*: f(n) = 0 + 1 + 2 + 3 + … + (n - 1) = (n * (n - 1)) / 2 ~ (n ^ 2) / 2

Qual algoritmo é melhor?

- *Merge sort*: f(n) = ???   (não é tão fácil de generalizar a função)
- f(n) = (número de comparações por nível) x (quantidade de níveis)
- Quantidade de níveis (em que são realizadas comparações): log2(n)
- Número de comparações por nível:
  - Não é trivial determinar tal número...
  - Varia conforme o conteúdo das subsequências (melhor/pior caso).
  - Mas é fácil determinar um limite máximo!!!
  - Em um nível, nunca são feitas mais do que n comparações.

- *Merge sort*: f(n) <= n * log2(n)

Qual algoritmo é melhor?

- *Selection sort*: f(n) ~ (n ^ 2) / 2
- *Merge sort*: f(n) <= n * log2(n)

n	**n * log2(n)**	**(n^2)/2**
4	8	8
8	24	32
16	64	128
32	160	512
64	384	2048

n	**n * log2(n)**	**(n^2)/2**	**fator**
1 (= 2^0)	0	0.5	-
1K (~ 2^10)	~ 10K	~500K	50
1M (~ 2^20)	~ 20M	500000M	25K

Qual algoritmo é melhor?

- *Selection sort*: f(n) ~ (n ^ 2) / 2
- *Merge sort*: f(n) <= n * log2(n)

n	**n * log2(n)**	**(n^2)/2**
4	8	8
8	24	32
16	64	128
32	160	512
64	384	2048

n	**n * log2(n)**	**(n^2)/2**	**fator**
1 (= 2^0)	0	0.5	-
1K (~ 2^10)	~ 10K	~500K	50
1M (~ 2^20)	~ 20M	500000M	25K

Merge sort é, definitivamente, melhor!!!

Qual algoritmo é melhor?

- *Selection sort*: $f(n) \sim (n \wedge 2) / 2$
- *Merge sort*: $f(n) <= n * \log2(n)$

n	n * log2(n)	(n^2)/2
4	8	8
8	24	32
16	64	128
32	160	512
64	384	2048

n	n * log2(n)	(n^2)/2	fator
1 (= 2^0)	0	0.5	-
1K (~ 2^10)	~ 10K	~500K	50
1M (~ 2^20)	~ 20M	500000M	25K

Merge sort é, definitivamente, melhor!!!
E nem precisamos calcular a função f(n) associada a ele de forma exata!