UDESC

Teoria da Computação

Prof. Diego Buchinger
diego.buchinger@outlook.com
diego.buchinger@udesc.br

UDESC

Introdução à Complexidade

UDESC

Introdução

- Temos dois tipos de problemas:
  - Insolúveis
  - Solúveis
- Mas será que os problemas “Solúveis” são todos tratáveis na prática – i.e. é possível resolvê-los de forma eficiente?

UDESC

Introdução

- Temos dois tipos de problemas:
  - Insolúveis
  - Solúveis
- Mas será que os problemas "Solúveis" são todos tratáveis na prática – i.e. é possível resolvê-los de forma eficiente?
  - **No!** (ex: solução demora 1 século para computar)
  - Alguns consomem demais os recursos (tempo e/ou espaço)

- Vamos nos ater ao tempo primeiramente...
- Como calcular o tempo que uma MT, ou um autômato, ou mesmo um computador leva para computar um dado problema?

Introdução

- Considere a linguagem $A = \{0^k 1^k \mid k \geq 0\}$
  - Quanto tempo uma MT de uma única fita precisa para decidir A?
  - Quantos passos uma MT de uma fita precisa para decidir A?

M_1 = "Sobre a cadeia de entrada *w*:

1. Faça uma varredura na fita e *rejeite* se for encontrado algum 0 a direita de algum 1
2. Repita se existem ambos, 0s e 1s, na fita:
3. Faça uma varredura na fita, cortando um único 0 e um único 1
4. Se ainda permanecerem 0s após todos os 1s tiverem sido cortados ou se permanecerem 1s após todos os 0s tiverem sido cortados, *rejeite*. Caso contrário (sem 0s e 1s na fita), *aceite*.

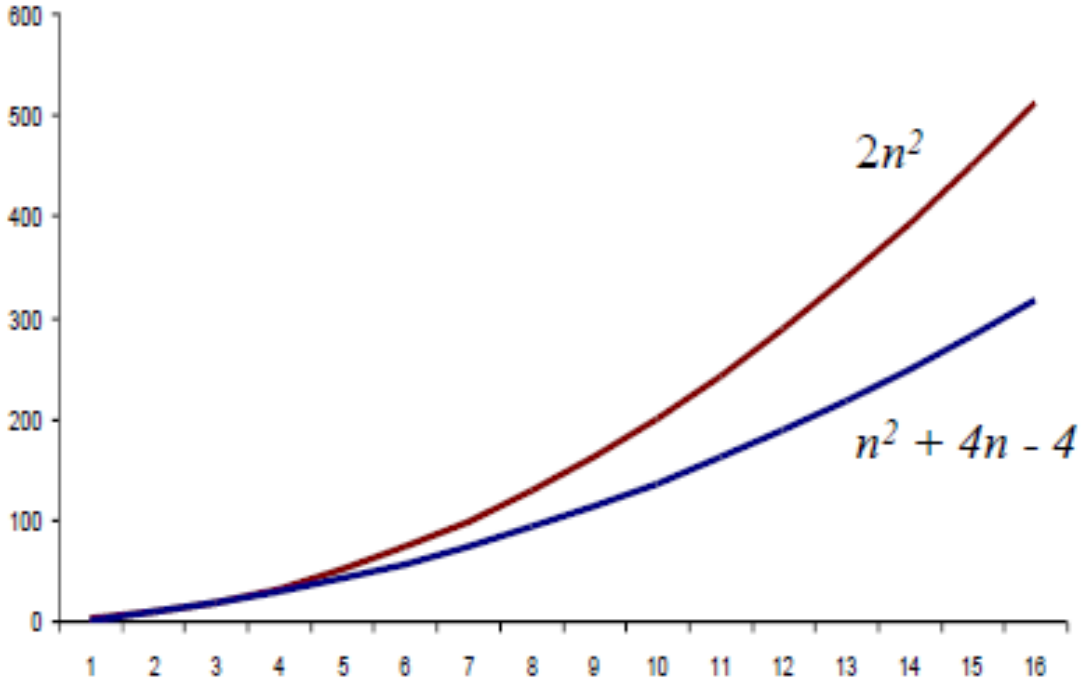
UDESC

Introdução

- O número de passos que um algoritmo usa sobre uma entrada específica pode depender de vários parâmetros:
  - Usualmente tamanho da entrada: vetor, texto, número grande;
  - A configuração inicial da entrada (fita);
  - Em um grafo: vértices, arestas, grau máximo do grafo.
- O número de passos também é influenciado pelo modelo de máquina utilizada!
- É comum analisarmos duas situações específicas
  - Análise pessimista ou do pior caso
  - Análise otimista ou do melhor caso

Notação Assintótica
(Notação O grande – Limite Superior)

Uma função $g(n)$ domina assintoticamente outra função $f(n)$ se existem duas constantes positivas c e n_0 tais que, para $n > n_0$, temos $|f(n)| \leq c.|g(n)|$ → $f(n) = O(g(n))$



$n^2 + 4n - 4 = O(n^2)$

$2n^2 = O(n^2)$

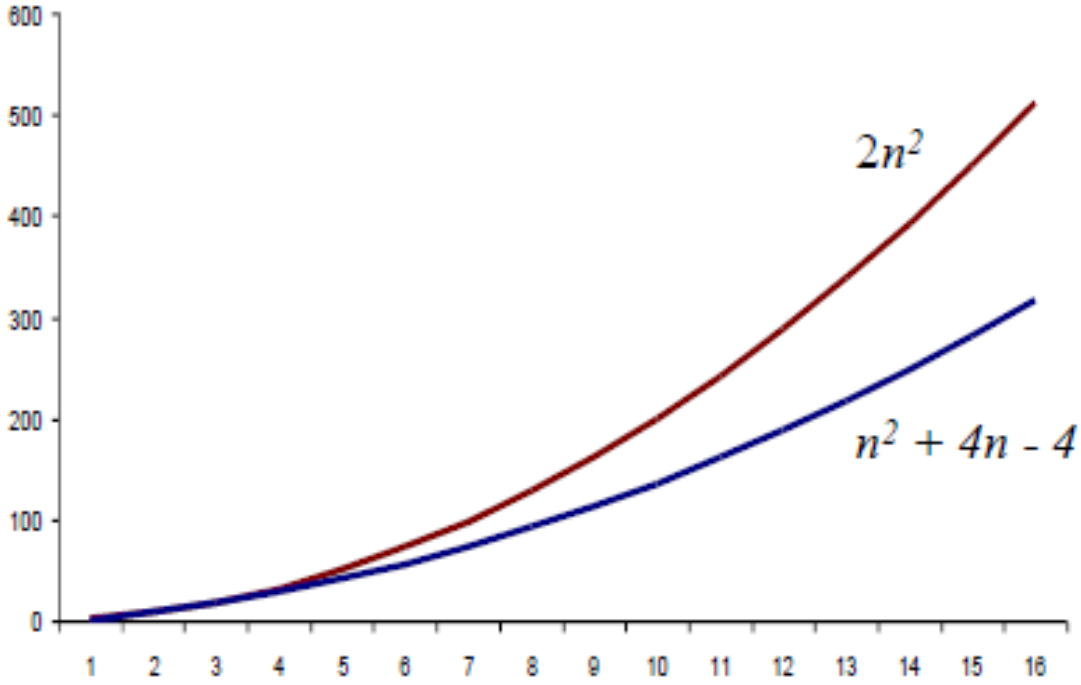
Notação Assintótica
(Notação O grande – Limite Superior)

- $f(n) = n^2 + 4n - 4$ g(n) = $O(n^2)$
- $f(n) = 2n^2$ g(n) = $O(n^2)$ [O(n) ?]

$f(n) / c$ & n_0	$c=1$ $n_0=3$	$c=1$ $n_0=50$	$c=2$ $n_0=1$	$c=2$ $n_0=10$	$c=3$ $n_0=2$	$c=3$ $n_0=10$
$2n^2$					-	-
$c(n^2)$						
$n^2 + 4n - 4$						
$c(n)$						

Notação Assintótica
(Notação O pequeno – *Little-O*)

Uma função $g(n)$ domina assintoticamente outra função $f(n)$ se existem duas constantes positivas c e n_0 tais que, para $n > n_0$, temos $|f(n)| < c.|g(n)|$ → $f(n) = o(g(n))$



$n^2 + 4n - 4 \neq o(n^2)$

$n^2 \neq o(n^2)$

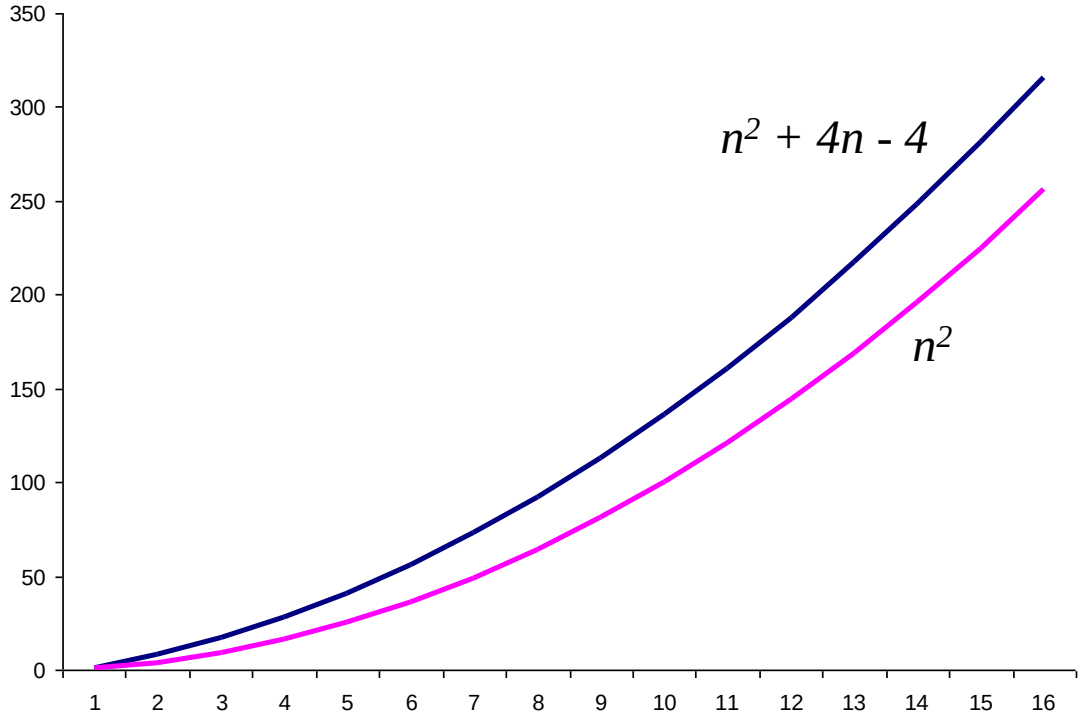
$4n - 8 = o(n^2)$

$n = o(n^2)$

Notação Assintótica

Notação Omega Grande – Limite Inferior

Uma função *f(n)* é o limite inferior de outra função *g(n)* se existem duas constantes positivas *c* e n_0 tais que, para $n > n_0$, temos $|g(n)| \geq c.|f(n)|$, $g(n) = \Omega(f(n))$.



$$n^2 = \Omega(\log n)$$

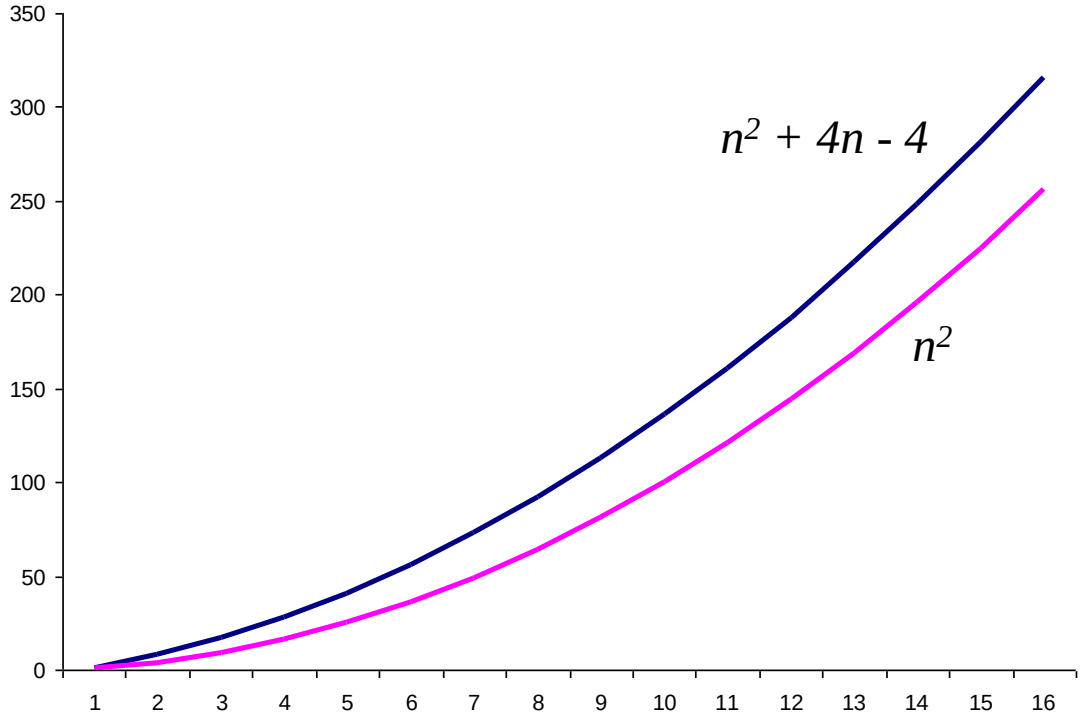
$$n^2 + 4n - 4 = \Omega(n^2)$$

$$n^2 = \Omega(n^2)$$

Notação Assintótica

Notação Omega Pequeno – Limite-Omega

Uma função *f(n)* é o limite inferior de outra função *g(n)* se existem duas constantes positivas *c* e n_0 tais que, para $n > n_0$, temos $|g(n)| > c.|f(n)|$, $g(n) = \omega(f(n))$.



$$n^2 + 4n - 4 \neq \omega(n^2)$$

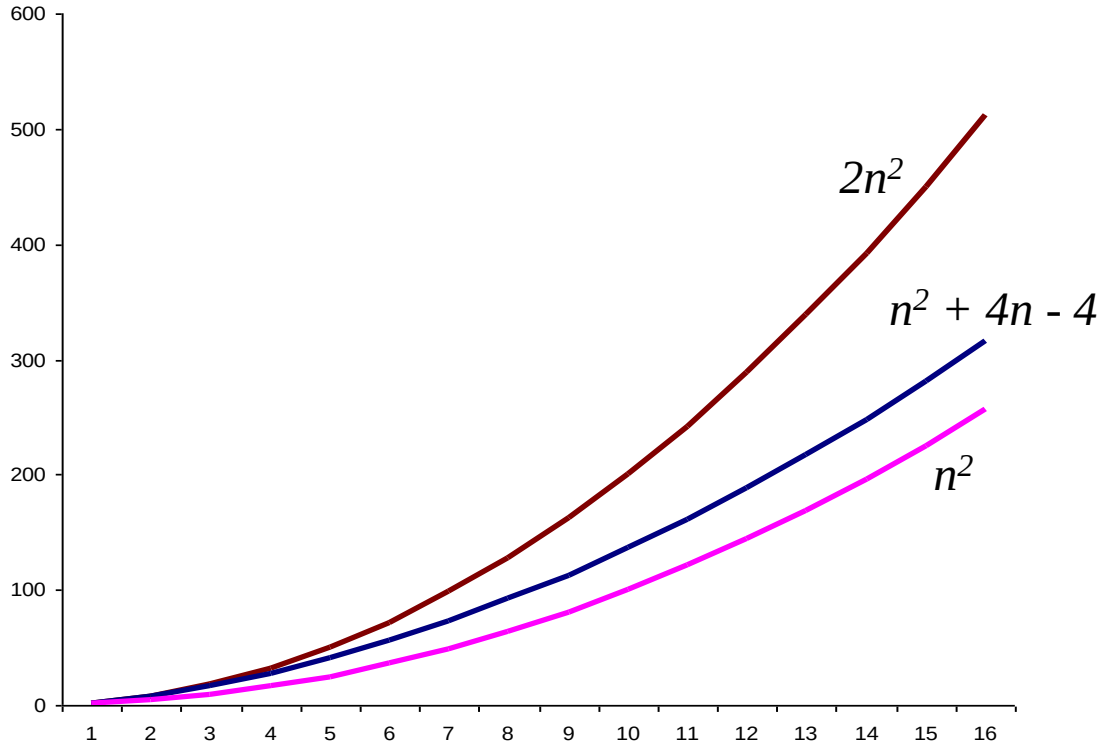
$$n^2 \neq \omega(n^2)$$

$$n^2 + 4n - 4 = \omega(n)$$

$$n^2 = \omega(n)$$

Notação Assintótica
Limite Firme (Notação Θ)

Uma função *f(n)* é o limite restrito (ou exato) de outra função *g(n)* se existem três constantes positivas c_1, c_2, e n_0 tais que, para $n > n_0$, temos $c_1.|f(n)| \geq |g(n)| \geq c_2.|f(n)|$, $g(n) = \Theta(f(n))$



[funções crescem com mesma rapidez]

$$n^2 + 4n - 4 = \Theta(n^2)$$

UDESC

Algumas Operações com Notação *O*

$c.O(f(n)) = O(f(n))$, onde c é uma constante.

$O(f(n)) + O(g(n)) = O(MAX(f(n), g(n))$

$n.O(f(n)) = O(n \cdot f(n))$

$O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$

UDESC

Funções de Complexidade

- Algumas funções são consideradas equivalentes assintoticamente – i.e. possuem um mesmo “peso”

- Quais são as funções genéricas que não são equivalentes?
  - Vamos testar alguns casos!?

Funções de Complexidade

Determine a quantidade de passos necessários para cada situação hipotética abaixo considerando os vários valores para *n*:

$f(n) / n$	*n=10*	*n=100*	*n=1.000*	*n=10.000*	*n=100.000*	*n=1.000.000*
n						
$log_{10} n$						
1						
n^2						
$3n$						
$\sqrt{n}$						
$n\ log_{10} n$						
2^n						
$n!$						

Funções de Complexidade

Podemos classificar os tempos de execução em:

$$1 < \log\log n < \log n < n^{\varepsilon} < n^{c} < n^{\log n} < c^{n}$$

Definição: definimos uma classe de complexidade de tempo denominada TIME(t(n)) como a coleção de todas as linguagens que são decidíveis por uma MT de tempo O(t(n))

Análise de Complexidade de Tempo

Voltemos ao nosso exemplo: $A = \{0^k 1^k \mid k \geq 0\}$

- Considere que a entrada é composta por n símbolos
- Qual é a complexidade de tempo do algoritmo apresentado?

M_1 = "Sobre a cadeia de entrada *w*:

1. Faça uma varredura na fita e *rejeite* se for encontrado algum 0 a direita de algum 1
2. Repita se existem ambos, 0
3. Faça uma varredura na f
4. Se ainda permanecerem 0s
   se permanecerem 1s após t
   Caso contrário (sem 0s e 1s

Note que não foi mencionado o reposicionamento do cabeçote no início da fita na descrição do passo 1!

A notação assintótica nos permite omitir detalhes da descrição da máquina que afetam o tempo de execução por, **no máximo**, um fator **constante**!

Análise de Complexidade de Tempo

Voltemos ao nosso exemplo: $A = \{0^k 1^k \mid k \geq 0\}$

- Considere que a entrada é composta por n símbolos
- Qual é a complexidade de tempo do algoritmo apresentado?

M_1 = "Sobre a cadeia de entrada *w*:

1. Faça uma varredura na fita e *rejeite* se for encontrado algum 0 a direita de algum 1
2. Repita se existem ambos, 0s e 1s, na fita:
3. Faça uma varredura na fita, cortando um único 0 e um único 1
4. Se ainda permanecerem 0s após todos os 1s tiverem sido cortados ou se permanecerem 1s após todos os 0s tiverem sido cortados, *rejeite*. Caso contrário (sem 0s e 1s na fita), *aceite*.

Análise de Complexidade de Tempo

Voltemos ao nosso exemplo: $A = \{0^k 1^k \mid k \geq 0\}$

- Utilizando M_1 mostramos que $A = O(n^2)$

  logo, $A \in TIME(n^2)$

- Mas será que $A = O(n^2)$ ou $A = o(n^2)$?

- Alguma ideia para construir uma máquina melhor?

  (=mais eficiente)

UDESC

Análise de Complexidade de Tempo

- Alguma ideia para construir uma máquina melhor?
  - Ao invés de cortar apenas um 1 e 0 cortar dois a cada passo
    - Isso ajuda?
    - Qual é a complexidade de tempo?

Análise de Complexidade de Tempo

- Alguma ideia para construir uma máquina melhor?
  - ~~Ao invés de cortar apenas um 1 e 0 cortar dois a cada passo~~
  - Proposta B:

M_2 = "Sobre a cadeia de entrada *w*:

1. Faça uma varredura na fita e *rejeite* se for encontrado algum 0 a direita de algum 1
2. Repita se existem ambos, 0s ou 1s, na fita:
3. Faça uma varredura na fita, verificando se o número total de 0s e 1s remanescentes é par ou ímpar. Se for ímpar, *rejeite*
4. Faça uma varredura novamente na fita, cortando alternadamente um 0 sim e outro não começando com o primeiro 0, e, então, cortando alternadamente um 1 sim e outro não começando com o primeiro 1.
5. Se nenhum 0 e nenhum 1 permanecerem na fita, *aceite*. Caso contrário, *rejeite*.

Análise de Complexidade de Tempo

- Alguma ideia para construir uma máquina melhor?
  - ~~Ao invés de cortar apenas um 1 e 0 cortar dois a cada passo~~
  - Proposta B – M_2 => O(n log n)

    Assim podemos dizer que $A \in TIME(n \log n)$

    Mas será que A = O(n log n) ou A = o(n log n) ?

Análise de Complexidade de Tempo

- Alguma ideia para construir uma máquina melhor?
  - ~~Ao invés de cortar apenas um 1 e 0 cortar dois a cada passo~~
  - Proposta B – M_2 => O(n log n)

    Assim podemos dizer que $A \in TIME(n \log n)$

    Mas será que A = O(n log n) ou A = o(n log n) ?

    No!!

    É possível provar que qualquer linguagem que pode ser decidida em tempo o(n log n) em uma MT de uma fita é regular!

UDESC

Relacionamentos de Complexidade entre Modelos Computacionais

UDESC

Desempenho Computacional

- Já mostramos que certas modificações em uma MT não aumentam a sua capacidade computacional.

- Mas será que essas modificações que definem um **modelo computacional** não melhoram o desempenho da computação (i.e. realizar a mesma coisa de maneira mais rápida assintoticamente)?

Desempenho Computacional

Voltemos ao nosso exemplo: $A = \{0^k 1^k \mid k \geq 0\}$

- Será que é possível construir uma MT de duas fitas que possui complexidade de tempo o(n log n)?

Desempenho Computacional

Voltemos ao nosso exemplo: $A = \{0^k 1^k \mid k \geq 0\}$

- Será que é possível construir uma MT de duas fitas que possui complexidade de tempo o(n log n)?
- Considere a magnifica MT de duas fitas M_3:

M_3 = "Sobre a cadeia de entrada *w*:

1. Faça uma varredura na fita e *rejeite* se algum 0 for encontrado à direita de algum 1.
2. Faça uma varredura nos 0s sobre a fita 1 até o primeiro 1. Ao mesmo tempo, copie os 0s para a fita 2.
3. Faça uma varredura nos 1s sobre a fita 1 até o final da entrada. Para cada 1 lido sobre a fita 1, corte um 0 sobre a fita 2. Se todos os 0s estiverem cortados antes que todos os 1s sejam lidos, *rejeite*.
4. Se todos os 0s tiverem sido cortados, *aceite*. Se restar algum 0, *rejeite*.

Desempenho Computacional

Voltemos ao nosso exemplo: $A = \{0^k 1^k \mid k \geq 0\}$

- Será que é possível construir uma MT de duas fitas que possui complexidade de tempo o(n log n)?
- Considere a magnifica MT de duas fitas M_3:

M_3 = "So

1. F à direita d
2. Faça uma varredura nos 0s sobre a fita 1 até o primeiro 1. Ao mesmo tempo, copie os 0s para a fita 2.
3. Faça uma varredura nos 1s sobre a fita 1 até o final da entrada. Para cada 1 lido sobre a fita 1, corte um 0 sobre a fita 2. Se todos os 0s estiverem cortados antes que todos os 1s sejam lidos, *rejeite*.
4. Se todos os 0s tiverem sido cortados, *aceite*. Se restar algum 0, *rejeite*.

Qual a **complexidade de tempo** da MT M_3?

Desempenho Computacional

- A MT de duas fitas M_3 possui complexidade de tempo: **O(n)**
- Mas será que $M_3 = O(n)$ ou $M_3 = o(n)$??

Desempenho Computacional

- A MT de duas fitas M_3 possui complexidade de tempo: **O(n)**
- Mas será que $M_3 = O(n)$ ou $M_3 = o(n)$
- É necessário *n* operações no mínimo para ler a entrada, então não tem como ser melhor do que isso!

Resumo da complexidade de tempo sobre A:

MT com uma fita: O(n log n)

MT com duas fitas: O(n)

A complexidade de A depende do modelo computacional utilizado!

Relacionamento de Complexidade

Teorema: seja $t(n)$ uma função onde $t(n) \geq n$. Então toda MT multifita de tempo $t(n)$ tem uma MT de uma fita equivalente de tempo $O(t^2(n))$.

Ideia da prova: lembrando-se que é possível converter qualquer MT multifita em uma MT de uma única fita que a simula, é possível perceber que simular cada passo da máquina multifita usa, no máximo, $O(t(n))$ passos na máquina de uma única fita. Logo, o tempo total usado é $O(t^2(n))$

Relacionamento de Complexidade

Teorema: seja $t(n)$ uma função onde $t(n) \geq n$. Então para toda MT não-determinística de uma fita de tempo $t(n)$, existe uma MT determinística de uma fita equivalente de tempo $2^{O(t(n))}$.

Ideia da prova: lembrando-se que é possível converter qualquer MT não-determinística em uma MT determinística de uma única fita que a simula, é possível perceber que é preciso explorar todo e cada ramificação da árvore de possibilidades gerada por uma MT não-determinística: $O(b^{t(n)})$ o que nos leva a uma complexidade de tempo de simulação de: $2^{O(t(n))}$.

Relacionamento de Complexidade

Teorema: seja $t(n)$ uma função onde $t(n) \geq n$. Então para toda MT não-determinística de uma fita de tempo $t(n)$, existe uma MT determinística de uma fita equivalente de tempo $2^{O(t(n))}$.

Ideia da prova: lembrando-se que é possível converter qualquer M... ca fi... e ca... T nã... le de tempo de simulação de: $2^{O(t(n))}$.

A definição do tempo de execução de uma MT não-determinística não tem o objetivo de corresponder a algum dispositivo de computação do mundo real. Ela é uma definição matemática útil que ajuda na caracterização da complexidade de uma classe importante de problemas computacionais!

Relacionamento de Complexidade

- De forma similar à classe $TIME(\ t(n)\ )$
- Temos uma classe análoga para categorizar complexidade de tempo em máquinas não-determinístico $\mathrm{N}TIME(\ t(n)\ )$

$\mathrm{N}TIME(\ t(n)\ )$ = {L | L é uma linguagem decidida por uma MT não-determinística de tempo O(t(n)) }

UDESC

Classes de Problemas

Introdução

Máquina de Turing de uma única fita

vs.

Máquina de Turing multifita

Diferença de, no máximo, uma potência quadrática ou polinomial entre complexidades de tempo [pequena]

Máquina de Turing determinista

vs.

Máquina de Turing não-determinista

Diferença no máximo exponencial entre complexidades de tempo [grande]

Introdução

Considere a diferença entre a taxa de crescimento polinomial $[n^3]$ e a taxa de crescimento exponencial $[2^n]$.

Suponha um valor pequeno para **n** e compare os resultados:

n = 1.000

– $[n^3]$: 1 bilhão – grande, mas viável

– $[2^n]$: maior do que a quantidade de átomos estimada no universo

Algoritmos de tempo polinomial costumam ser suficientemente rápidos para diversos propósitos.
Já algoritmos de tempo exponencial raramente são úteis.

Introdução

Mas de onde taxas de crescimento exponencial?

UDESC

Introdução

Mas de onde taxas de crescimento exponencial?

- Surgem de buscas exaustivas em um espaço de soluções, corriqueiramente chamadas de **busca por força bruta**

  ex: fatorar um número em seus primos constituintes buscando por todos os potenciais divisores

- Em alguns casos sabe-se que a força bruta pode ser evitada através de um entendimento mais profundo do problema, revelando um algoritmo de tempo polinomial

- Algoritmos de **programação dinâmica** são exemplo de maneiras de resolver certos problemas de forma mais eficiente

UDESC

Equivalência Polinomial

Todos os modelos computacionais determinísticos *razoáveis* são **polinomialmente equivalentes**.

- *Razoável* não é definido, mas é amplo o suficiente para incluir modelos que aproximam o tempo de execução em computadores reais
- Em teoria da computação mantemos nosso foco em descobrir propriedades fundamentais da computação
  - ❑ A primeira vista, desconsiderar diferenças polinomias parece absurdo – programadores se esforçam para deixar seus programas 2x mais rápido!
  - ❑ A equivalência apresentada é interessante para uma visão macro, na qual ignoramos o modelo específico de computação utilizado;
  - ❑ Isso não significa que considerar as diferenças polinomias não seja importante em alguns aspectos (analogia entre observar a floresta como um todo ou cada árvore individualmente).

Classe de Problemas P

Definição: P é a classe de linguagens que são decidíveis em tempo polinomial sobre uma MT determinística de uma única fita.

$$P = \bigcup_k TIME(n^k)$$

- Note que:
  - ❖ P é invariante para todos os modelos de computação polinomialmente equivalentes à MT. Determinísticas de uma única fita;
  - ❖ P corresponde aproximadamente à classe de problemas que são realisticamente solúveis em um computador;

É improvável que um tempo de execução na ordem de n^{100} tenha qualquer uso prático, mas esse tipo de ordem não é usual.

Classe de Problemas P

Como determinar complexidade polinomial com pseudocódigo

- Determinar uma quantidade polinomial de estágios
- Descrever cada estágio com um custo de tempo polinomial
- Resultado: polinômio x polinômio = polinômio

- Cuidado com a codificação utilizada!
  - Todo tipo de entrada (números, texto, grafos, equações etc.) deve ser codificada de forma polinomial
  - Ex: notação unária para representar o número 12: 111111111111

    Pode isso Arnaldo?
  - Uma codificação em binário é válida/viável? Qual é o seu custo em relação ao valor decimal utilizado?

UDESC

Classe de Problemas P

Alguém saberia citar exemplos de problemas que pertencem a P?

Categoria de Problemas

Problemas de Otimização: cada solução possível tem um valor associado e desejamos encontrar a solução com melhor valor

Problemas de Decisão: problema que tem como resposta sim ou não

- Problemas de Decisão são possivelmente "mais fáceis" do que problemas de otimização, mas com certeza "não mais difíceis"

Exemplo:

- Qual é o menor caminho de caixeiro viajante em um grafo *G*?
- Existe um caminho de caixeiro viajante com peso menor do que *k* entre *a* e *b*?

UDESC

Problemas Difíceis (?!)

Apesar de técnicas avançadas de programação, para certos problemas não se conhece uma solução mais eficiente do que uma busca exaustiva!

- Logo, não podemos afirmar que tais problemas pertencem a P
- Contudo, fica a seguinte dúvida:
  - Será que apenas ainda não descobrimos uma solução eficiente, ou um princípio que tornaria tal problema mais simples?
  - Ou será que existem problemas que são intrinsecamente difíceis?

- Entretanto, uma coisa é certa: se um problema A pode ser resolvido com uma busca exaustiva, então $A \in NTIME(n^k)$

Alguém saberia citar exemplos de problemas que não se conhece uma solução eficiente para resolvê-los?

Classe de Problemas NP

Definição: NP é a classe de linguagens que são decidíveis em tempo polinomial sobre uma MT não-determinística.

$$NP = \bigcup_k NTIME(n^k)$$

- Cuidado:
  - NP **NÃO SIGNIFICA** que o problema pode ser resolvido em tempo Não-Polinomial (por um algoritmo determinista)

- Propriedade:
  - Problemas NP possuem a característica de **verificabilidade polinomial**

Classe de Problemas NP

Definição: Um verificador para uma linguagem A é um algoritmo V onde:

$$A = \{w \mid V\ aceita\ \langle w, c\rangle\ para\ alguma\ cadeia\ c\}$$

O elemento c é o **certificado** ou **prova** - isto é, uma solução para o problema que possui comprimento polinomial em relação a *w*.

Medimos o tempo de um verificador em termos do comprimento de *w*, portanto um verificador de tempo polinomial roda em tempo polinomial no comprimento de *w*. Assim, uma linguagem é dita **polinomialmente verificável** se ela tem um verificador de tempo polinomial.

Ex: Caminho Hamiltoniano (passar por todos os vértices uma vez)

Composto = { x | x = pq, para inteiros p, q > 1}

UDESC

Classe de Problemas NP

Definição 2: NP é a classe de linguagens que possuem verificadores de tempo polinomial.

Podemos converter um verificador de tempo polinomial para uma MT não-determinística de tempo polinomial equivalente e vice-versa.

+ A MTN simula o verificador adivinhando o certificado.
+ O verificador simula a MTN usando o ramo de computação de aceitação como o certificado.

Existem problemas que não são polinomialmente verificáveis?

Classe de Problemas NP

Definição 2: NP é a classe de linguagens que possuem verificadores de tempo polinomial.

Existem problemas que não são polinomialmente verificáveis?

Sim!

$\overline{CAMHAM}$ - o complemento do problema CAMHAM

Determinar se um grafo não tem um caminho hamiltoniano

Não se conhece uma forma de permitir que alguém verifique sua não existência sem testar todos os possíveis caminhos

Verificar que algo *não está* presente parece ser mais difícil que verificar que algo *está* presente.

UDESC

Classe de Problemas coNP

- Verificar que algo *não está* presente parece ser mais difícil que verificar que algo *está* presente.
- Diversos problemas NP possuem como complemento problemas que não são membros óbvios de NP (e podem nem ser mesmo!)

Definição: coNP é a classe de linguagens que são complementos das linguagens em NP.

Não sabemos se $coNP \neq NP$

Exemplo problema caixeiro viajante:

Verificar se existe um caminho de caixeiro viajante < X

Verificar se não existe um caminho de caixeiro viajante < X

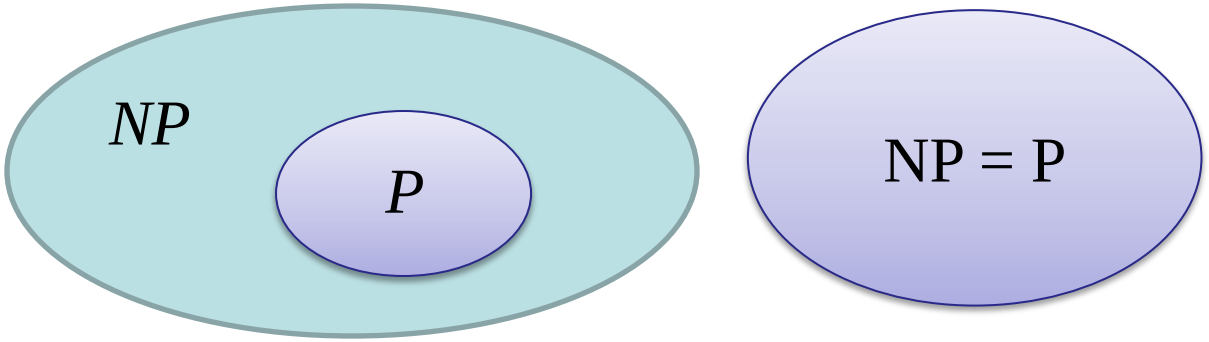
A questão P vs. NP

P **=** a classe das linguagens para as quais pertinência pode ser *decidida* em tempo polinomial (rapidamente)

NP **=** a classe das linguagens para as quais pertinência pode ser *verificada* em tempo polinomial (rapidamente)

Pergunta do milhão: P=NP ou P ≠NP?

Possíveis relações entre as classes:



A questão P vs. NP

A questão P = NP? É um dos maiores problemas não resolvidos em ciência da computação teórica e matemática contemporânea.

Se essas classes forem iguais, então todo problema verificável em tempo polinomial é também decidível em tempo polinomial.

Acredita-se que a relação mais provável é que P ≠ NP. Por quê?

Até o momento, podemos afirmar apenas que:

$$NP \subseteq EXPTIME = \bigcup_k TIME(2^{n^k})$$