Geometria Computacional

Professor:

Anselmo Montenegro
www.ic.uff.br/~anselmo

Conteúdo:

- Noções de análise de complexidade

Baseado em Notas de aula do Prof. Luiz Henrique Figueiredo – IMPA.

Roteiro

- Introdução
- Modelos de computação
- Árvores de decisão algébrica
- Complexidade de algoritmos
- Análise assintótica
- Notações O,Ω e Θ
- Complexidade de problemas
- Caso de estudo: problema de ordenação
- Redução

- A análise de complexidade tem dois objetivos fundamentais:
  - Comparar a eficiência de algoritmos diferentes para um mesmo problema
  - Analisar a complexidade inerente a um problema, independentemente dos algoritmos que tenham sido desenvolvidos ou não para ele

Geometria Computacional: modelos de computação

- A análise sobre a eficiência de um algoritmo requer a especificação de um modelo computacional

- Um modelo computacional é um modelo que define quais algoritmos são válidos e quais os custos das suas operações básicas

- Um modelo computacional nos permite inferir propriedades matemáticas sobre algoritmos

Geometria Computacional: árvores de decisões algébricas

- Um dos modelos computacionais mais conhecidos é o modelo baseado em Árvores de Decisões Algébricas
- Nesse modelo, os algoritmos procedem avaliando expressões algébricas e tomando decisões baseadas no sinal desses valores

Geometria Computacional: árvores de decisões algébricas

- Outro modelo de computação é o *Real RAM – Real Random Access Machine:*
- Toda posição de memória armazena um único número real
- São válidas as seguintes operações com custo unitário:

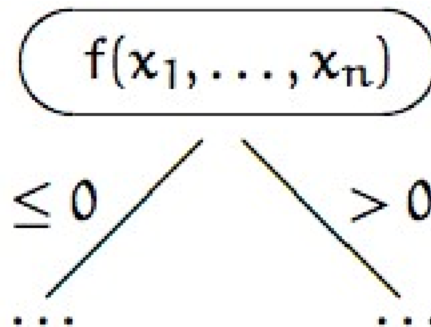
  a) operações aritméticas
  b) operações de comparação entre reais
  c) acesso indireto a memória
  d) raízes de ordem k, exponenciação, logaritmo, funções trigonométricas

Geometria Computacional: árvores de decisões algébricas

- Cada nó interno da árvore corresponde a uma abstração de um passo do algoritmo

- Em cada passo um polinômio em *n* variáveis $x_1,\ldots, x_n$ é avaliado

- Se o sinal for positivo, o algoritmo vai para o nó à direita, caso contrário segue para a esquerda



Geometria Computacional: árvores de decisões algébricas

- A complexidade de um algoritmo representado por uma árvore de decisões algébricas é medido pelo número de nós visitados na solução do problema, desde a raiz até uma folha

- Considera-se que as folhas contém as soluções do problema

- No modelo de árvores de decisões algébricas, as operações básicas (avaliação de um polinômio e comparação) — têm todas o mesmo custo (independente do grau do polinômio)

Geometria Computacional: árvores de decisões algébricas

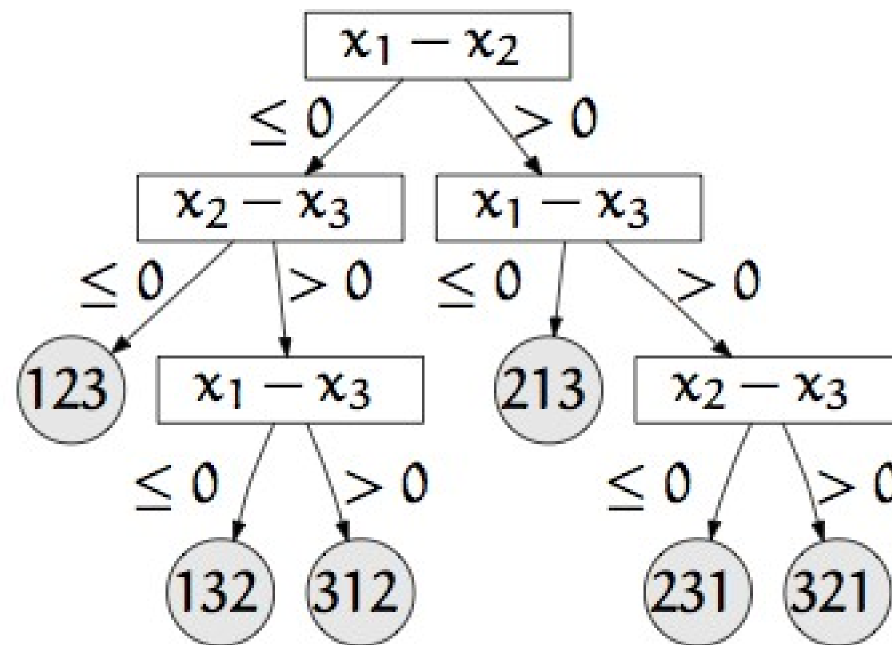
- Exemplo de problema analisado por uma ADA:

  Problema :ordenar três números reais

  Entrada: $x_1, x_2, x_3 \in R$

  Saída: uma permutação π do conjunto de índices {1, 2, 3} tal que $x_{\pi(1)} \leq x_{\pi(2)} \leq x_{\pi(3)}$

Geometria Computacional: árvores de decisões algébricas



Geometria Computacional: árvores de decisões algébricas

- A complexidade é definida como o número máximo de nós visitados no pior caso
- Seja *P* um problema, *A* um algoritmo que resolve *P* e *I* uma instância de *P*
- Denominamos por $T_A(I)$ o custo de *A* para resolver a instância *I* de *P*
- $T_A(I)$ é dado pelo número de nós da ADA, que representa *A*, que são visitados na solução de *P* para uma instância *I*

Geometria Computacional: árvores de decisões algébricas

- Um dos objetivos da análise de complexidade é comparar a eficiência de algoritmos para um dado problema
- Porém, $TA(I)$ varia com a entrada (ver exemplo)
- Além disso, dois algoritmos *A* e *B* podem se comportar de modo diferente para diferentes instâncias de um problema
- Como resolver este impasse?

Geometria Computacional: árvores de decisões algébricas

- Solução: expressar o custo em função da instância *I* mais desfavorável dentre um conjunto de instâncias $\{I_0, I_1, ..., I_m\}$ de um mesmo tamanho n
- Tal análise é denominada Análise de Complexidade de Pior Caso

Geometria Computacional: árvores de decisões algébricas

- Assim, a complexidade de pior caso T_A de um algoritmo A é dada por

$$T_A(n) = \max_{n=|I|} T_A(n).$$

Geometria Computacional: árvores de decisões algébricas

- Um algoritmo que resolve um problema para instâncias de qualquer tamanho é representado por uma família de árvores, uma para cada tamanho de instância

- O objetivo da análise de complexidade é analisar como a profundidade de uma árvore representando instâncias de um mesmo tamanho crescem em função do tamanho *n* do problema

Geometria Computacional: análise assintótica

- A análise de pior caso pode não ser tão informativa em alguns casos (Por que?)
- Neste caso busca-se a análise de caso médio
- A análise de caso médio é muito difícil em muitos casos pois requer a definição de uma distribuição de probabilidades para as instâncias do problema

Geometria Computacional: análise assintótica

- A análise de pior caso é extremamente útil se considerarmos como T_A(n) se comporta para valores grandes de *n*
- Tal análise é denominada Análise Assintótica
- Deste modo é possível comparar dois algoritmos A e B comparando $T_A(n)$ com $T_B(n)$
- A análise de complexidade assintótica independe de avanços tecnológicos

Geometria Computacional: Análise assintótica

- A análise assintótica de TA(n) é feita estimando a ordem de grandeza do crescimento de TA(n) em função de n
- As notações O,Θ e Ω(f) expressam este tipo de estimativa

Geometria Computacional: Análise assintótica

- Sejam f, g: $R \rightarrow R$ duas funções positivas. Dizemos que:
  - g é O(f) quando existem $N_0 \in \mathbf{N}$ e $c \in R$,$c>0$, tais que $g(n) \leq cf(n)$ para todo $n \geq N_0$
  - g é Ω(f) quando existem $N_0 \in \mathbf{N}$ e $c \in R$, $c>0$, tais que $g(n) \geq cf(n)$ para todo $n \geq N_0$
  - g é Θ(f) quando g é O(f) e g é Ω(f)

Geometria Computacional: análise assintótica

- Na prática abusa-se da notação “=“ e escreve-se

  g = O(f)
  g = Ω(f)
  g = Θ(f)

- Exemplo de crescimento assintótico:

$$1 \prec \log n \prec n^{1/k} \prec \sqrt{n} \prec n \prec n\log n \prec n^2 \prec n^k \prec 2^n \prec n! \prec n^n$$

Geometria Computacional: Complexidade de problemas

- Qual a melhor complexidade que se pode esperar de um algoritmo que resolva um dado problema?
- É possível comparar problemas diferentes?
- Existem problemas intrinsicamente mais difíceis do que outros?

Geometria Computacional: Complexidade de problemas

•Definimos a complexidade de um problema *P* como sendo o melhor que algum algoritmo que resolve *P* consegue fazer em termos de custo computacional:

$$T_P(n) = \min_{A} T_A(n).$$

•*A* é o conjunto de todos os algoritmos que resolvem *P*, quer sejam conhecidos ou não

Geometria Computacional: complexidade de problemas

•Se A resolve P então $T_A = \Omega(T_P)$

•Isto é, nenhum algoritmo que resolva P pode ser mais rápido do que a complexidade de P

Geometria Computacional: complexidade de problemas

•Exercício: com base no slide anterior é verdade que $T_P = O(T_A)$? Por que?

Geometria Computacional: Complexidade de problemas

- A existência de um algoritmo *A* que resolve *P* nos dá um limite superior, $T_P = O(T_A)$, mas nunca um limite inferior

- Para obter um limite inferior é necessário analisar o problema *P* e descobrir propriedades de todos os algoritmos que resolvem *P*

Geometria Computacional: Complexidade de problemas

- Dizemos que um algoritmo *A* é ótimo quando $T_A = O(T_P)$, ou seja, quando $T_A = \Theta(T_P)$ ($T_P = \Theta(T_A)$)
- Isso quer dizer que *A* consegue resolver *P* no melhor tempo possível
- É importante conhecer T_P mesmo que não se conheçam algoritmos ótimos

Geometria Computacional: Complexidade de problemas

•Encontrar limites superiores para a complexidade de um problema, isto é, estimativas do tipo $O(f)$, é fácil

•Encontrar limites inferiores, isto é, estimativas do tipo $\Omega(f)$, é uma tarefa bem mais difícil

Geometria Computacional: Complexidade de problemas

•Não é difícil, porém, encontrar limites inferiores triviais

•Um limite inferior para o problema de ordenação é $\Omega(n)$ (Por que?)

•Por outro lado, o que se procura são limites inferiores não triviais

Geometria Computacional: Complexidade de problemas

- A prática de análise da complexidade de um problema *P* consiste em encontrar o maior limite inferior e o menor limite superior possível

- Em outras palavras, queremos reduzir o intervalo em torno de TP

$$\Omega(\mathrm{f}) \prec \mathrm{T_P}(n) \prec O(g)$$

- O intervalo se fecha quando encontramos um algoritmo ótimo

Geometria Computacional: Complexidade do problema de ordenação

•Problema :ordenar *n* números reais

•Entrada: $x_1, x_2, x_3, \ldots, x_n \in R$

•Saída: uma permutação π do conjunto de índices {1, 2, 3,...,n} tal que $x_{\pi(1)} \leq x_{\pi(2)} \leq x_{\pi(3)} \leq \ldots \leq x_{\pi(n)}$

Geometria Computacional: Complexidade do problema de ordenação

•Importância do problema de ordenação para Geometria Computacional:

- Ordenação compõe parte da solução de muitos problemas
- A complexidade de muitos problemas de GC pode ser analisada com base em resultados de complexidade para o problema de ordenação

Geometria Computacional: Complexidade do problema de ordenação – limite inferior

- Seja um algoritmo A que resolve o problema de ordenação de n números

- A ADA que representa A tem pelo menos *n*! folhas (Por que?)

- Por outro lado a árvore binária de profundidade p associada a ADA tem 2^p folhas (Por que?)

Geometria Computacional: Complexidade do problema de ordenação – limite inferior

- Mostre porque uma árvore binária de profundidade p associada tem 2^p folhas

Geometria Computacional: Complexidade do problema de ordenação – limite inferior

- Logo $2^p \geq n!$
- Consequentemente,

$$TA(n) = p = \log_2(2^p) \geq \log_2 n!$$

- Deste modo:

$$TA(n) = \log_2 n!$$

Geometria Computacional: Complexidade do problema de ordenação – limite inferior

- Como estimar o valor de $\log_2 n!$:

$$(n!)^2 \geq n^n$$
$$(n!)^2 = (1\cdot 2\cdot 3\ldots\cdot(n-2)\cdot(n-1)\cdot n)^2 \geq n^n$$
$$(n!)^2 = (1\cdot 2\cdot 3\ldots\cdot(n-2)\cdot(n-1)\cdot n)(n\cdot(n-1)\cdot(n-2)\cdot\ldots\cdot 3\cdot 2\cdot 1) \geq n^n$$
$$(n!)^2 = (1\cdot n)(2\cdot(n-1))(3\cdot(n-2))\ldots((n-2)\cdot 3)((n-1)2)(n\cdot 1) \geq n^n$$

$$a\cdot b > n$$
$$a+b = n+1$$
$$a \geq 2, b \geq \frac{n}{2} \text{ ou } b \geq 2, a \geq \frac{n}{2}$$

- Assim

$$(n!)^2 \geq n^n$$
$$\log_2 n! \geq \frac{1}{2}\log_2 n^n = \frac{1}{2} n \log_2 n$$
$$T_A(n) = \Omega(n \log n)$$

Geometria Computacional: Complexidade do problema de ordenação – limite superior

- Para determinar um limite superior basta encontrar um algoritmo que resolva o problema

- Nem sempre um problema admite um algoritmo (por exemplo, o Problema da Parada)

- Um algoritmo trivial é um algoritmo por força bruta onde geramos as $n!$ permutações π e testamos se cada uma delas satisfaz

$$x_{\pi(1)} \leq x_{\pi(2)} \leq x_{\pi(3)} \leq \ldots \leq x_{\pi(n)}$$

Geometria Computacional: Complexidade do problema de ordenação – limite superior

- Um algoritmo trivial é um algoritmo por força bruta onde geramos as $n!$ permutações π dos índices de uma entrada $x_1,x_2,...,x_n$ e testamos se cada uma delas satisfaz

$$x_{\pi(1)} \leq x_{\pi(2)} \leq x_{\pi(3)} \leq ... \leq x_{\pi(n)}$$

- É possível entretanto fazer muito melhor que isso!

- Vejamos dois algoritmos para o problema de ordenação e suas complexidades assintóticas de pior caso

Geometria Computacional: Ordenação por seleção direta

```
Para i=1,...,n-1 faça
   m = i;
      Para j=i+1,...,n faça
         Se (v[i]<v[m]) então
            m = j;
         Fim_Se
      Fim_Para
      Trocar(v[i],v[m]);
Fim_Para
```

$$n + (n-2) + \ldots + 2 = \frac{n(n+1)}{2} - 1 = O(n^2)$$

Geometria Computacional: MergeSort

```
MergeSort(x,n)
        Se n<2 então retorne;
        Fim_Se;
        m = n/2;
        l = x[1..m];
        r = x[m+1..n];
        MergeSort(l,m);
        MergeSort(r,n-m);
        i = 1;
        j = 1;
        Para k=1..n faça
           Se l[i]<r[j] então
              x[k] = l[i]; i = i+1;
           Senão
              x[k] = r[j]; j = j+1;
        Fim_Para
```

$$T(n) = 2T(n/2) + kn$$
$$T(n/2) = 2T(n/4) + kn/2$$
$$T(n) = 4T(n/4) + 2kn$$
$$T(n) = 8T(n/8) + 3kn$$

...

$$T(n) = 2^p T(n/2^p) + pkn = nT(1) + k(\log_2 n)n \therefore$$
$$T(n) = O(n \log n)$$

Geometria Computacional: Mergesort

•Podemos observar que o algoritmo MergeSort resolve o problema em T_A=O($n$log$n$) = T_P

•Logo o algoritmo é ótimo

•Assim concluímos que TP é Θ($n$log$n$) e concluímos a análise de complexidade de pior caso

Geometria Computacional: Redução

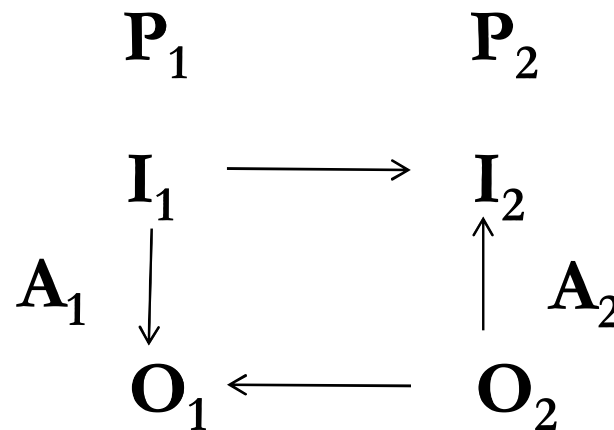
•Uma das ferramentas mais poderosas para análise de complexidade de problemas é a redução

•A redução permite que possamos definir em muitos casos limites inferiores e superiores não triviais para problemas

Geometria Computacional: Redução

- Definição: um problema P_1 é redutível em tempo linear a um problema P_2, se dada uma instância I_1 de P_1 for possível construir, em tempo O(n), uma instância I_2 de P_2 tal que a solução O_1 de P_1 possa ser construída, em tempo O(n), a partir da solução O_2 de P_2

- Dizemos que $P_1 \propto_{O(n)} P_2$



Geometria Computacional: Redução

• Intuitivamente podemos afirmar que P_1 é tão fácil de resolver quanto P_2

•Teorema (da redução) – Sejam dois problemas P_1 e P_2,tais que $P_1 \propto_{O(n)} P_2$ então

a) Se existe um algoritmo que resolve P_2 em $O(T_{P2})$ então existe um algoritmo que resolve P_1 em $O(T_{P2})$

b) Se qualquer algoritmo que resolve P_1 requer $\Omega(T_{P1})$ passos então qualquer algoritmo que resolve P_2 também requerirá $\Omega(T_{P1})$ passos

Referências:

• ***Notas de aula*** **do Prof. Luiz Henrique Figueiredo – IMPA.**