Universidade Federal de Viçosa

Universidade Federal de Viçosa
Departamento de Informática
Centro de Ciências Exatas e Tecnológicas

INF 112 – Programação 2
Aula 4

Introdução à Análise de Algoritmos 2

Aula baseada nos slides disponibilizados por Anany Levitin (autor do livro: *The Design and Analysis of Algorithms*) e nas aulas do Prof. Fabio Ribeiro

Introdução à Análise de Algoritmos

- Notação assintótica
- Como visto, a análise de eficiência de algoritmos normalmente se concentra na “ordem de complexidade” da operação básica do algoritmo.
- Para comparar essa ordem, são utilizadas três notações: O, Ω, Θ
- Vamos estudar tais notações de maneira informal.

Introdução à Análise de Algoritmos

- Notação assintótica: O ("o" maiúsculo), Ω, Θ
  - $O(g(n))$: conjunto de todas as funções com ordem de crescimento menor ou igual a $g(n)$.
  - $\Omega(g(n))$: conjunto de todas as funções com ordem de crescimento maior ou igual a $g(n)$.
  - $\Theta(g(n))$: conjunto de todas as funções com mesma ordem de crescimento que $g(n)$.
- Vamos nos concentrar na notação O.

Introdução à Análise de Algoritmos

- O(g(n)) : conjunto de todas as funções com ordem de crescimento menor ou igual a g(n).
  - $n \in O(n^3)$
  - $n \in O(n^2)$
  - $n \in O(n)$
  - $100n + 999999999 \in O(n)$
  - $100n + 999999999 \in O(n^2)$
  - $2n^3 \in O(n^3)$

Introdução à Análise de Algoritmos

- $O(g(n))$: conjunto de todas as funções com ordem de crescimento menor ou igual a $g(n)$.
  - $999999999999n \in O(n)$
  - $n^3 \notin O(n^2)$
  - $n^4 + 3n^2 + 100n \notin O(n^2)$
  - $0.000000000000000000000001n^3 \notin O(n^2)$

Introdução à Análise de Algoritmos

- Definição: uma função t(n) está em O(g(n)), denotando-se por t(n) $\in$ O(g(n)), se t(n) é limitada superiormente por algum múltiplo constante de g(n) para todo n grande. Ou seja, se existem alguma constante positiva c e um inteiro não negativo n_0 tais que:
  - t(n) ≤ cg(n), para todo $n \geq n_0$

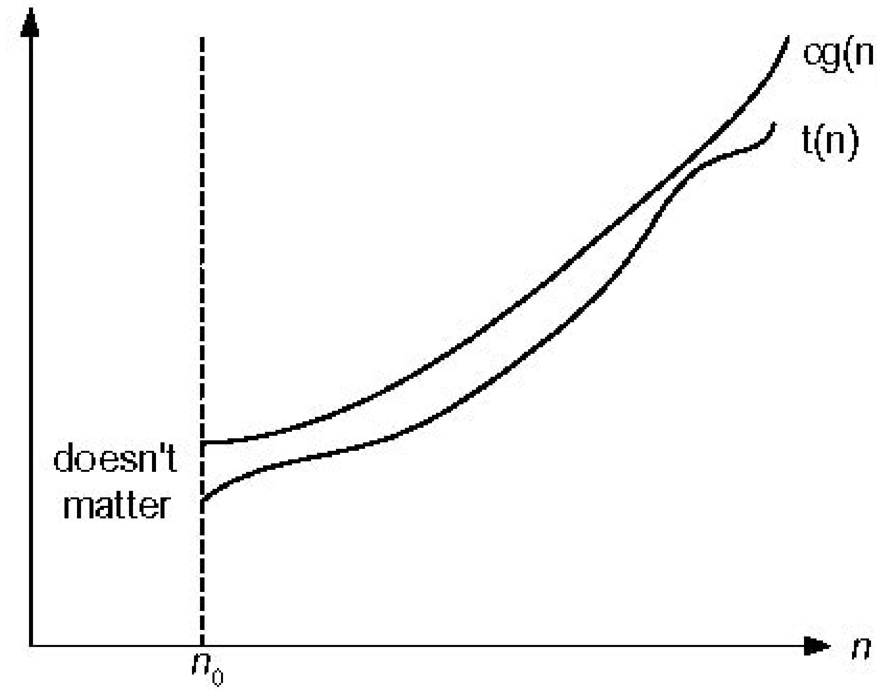

Figure 2.1 Big-oh notation: $t(n) \in O(g(n))$

Introdução à Análise de Algoritmos

- Propriedades da notação assintótica:
- $t(n) \in O(t(n))$
  - Ex: $n^3 \in O(n^3)$
- Se $t(n) \in O(g(n))$ e $g(n) \in O(h(n))$, então $t(n) \in O(h(n))$.
  - Ex: $n^2 \in O(n^3)$ e $n^3 \in O(n^4) \rightarrow n^2 \in O(n^4)$
- Se $t_1(n) \in O(g_1(n))$ e $t_2(n) \in O(g_2(n))$, então $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$
  - Ex: $n^2 \in O(n^2)$, $n^3 \in O(n^3) \rightarrow n^2 + n^3 \in O(\max\{n^2, n^3\}) = O(n^3)$

Introdução à Análise de Algoritmos

- Qual a relação destas propriedades com os algoritmos?
- A última propriedade afirma que a eficiência geral do algoritmo é determinada pela parte com maior ordem de complexidade (a menos eficiente).
- Assim, se um algoritmo possui dois trechos, sendo que o primeiro executa a operação básica $3n^2$ vezes e a segunda executa a operação básica n^3 vezes, então o algoritmo terá complexidade $O(n^3)$.

Introdução à Análise de Algoritmos

- Mais propriedades:
- Toda função logarítmica pertence à classe O(log(n)), independente da base (>1) do logarítimo.
- Todos os polinômios de grau k pertencem à classe $O(n^k)$. Ex: $n^5 + 5n^2 + 1000n \in O(n^5)$

Introdução à Análise de Algoritmos

- Classes básicas de eficiência
- Mesmo com a notação assintótica agrupando funções que diferem por múltiplos constantes, ainda assim há infinitas "classes de complexidade" (ex: funções do tipo a^n pertencem a diferentes classes dependendo do valor de a).
- Porém, a MAIORIA dos algoritmos se enquadra em poucas classes básicas de eficiência.

Classe	Nome	
1	constante	Polinomiais
log(n)	logarítmica	
n	linear	
n log(n)	"n-log-n"	
n^2	quadrática	
n^3	cúbica	
2^n	exponencial	Exponenciais
n!	fatorial	

Introdução à Análise de Algoritmos

- Em geral, um algoritmo de classe mais baixa (O(n)) é muito mais eficiente do que um algoritmo de uma classe mais alta (ex: $O(n^2)$).
- Porém, pode haver algumas exceções:
  - Por exemplo, um algoritmo que executa 10^6n^2 operações básicas poderia ser menos eficiente do que um algoritmo que realiza n^3 operações básicas para $n \leq 10^6$.
  - Porém, essas "anomalias" são MUITO RARAS. Em geral, pode-se assumir que um algoritmo $O(n^2)$ é mais eficiente do que um $O(n^3)$ mesmo para entradas de tamanho moderado.

Introdução à Análise de Algoritmos

- Como analisar um algoritmo e descobrir qual a sua classe de complexidade?

  1. Identificar a operação básica do algoritmo.

Introdução à Análise de Algoritmos

- Como analisar um algoritmo e descobrir qual a sua classe de complexidade?

  1. Identificar a operação básica do algoritmo.
  2. Identificar a melhor forma de se definir o tamanho da entrada.

Introdução à Análise de Algoritmos

- Como analisar um algoritmo e descobrir qual a sua classe de complexidade?

  1. Identificar a operação básica do algoritmo.
  2. Identificar a melhor forma de se definir o tamanho da entrada.
  3. Verificar se o número de vezes que ela é executada pode variar para diferentes entradas de um mesmo tamanho. Se for o caso, deve-se analisar o melhor caso, pior caso e caso médio.

Introdução à Análise de Algoritmos

- Como analisar um algoritmo e descobrir qual a sua classe de complexidade?

  1. Identificar a operação básica do algoritmo.
  2. Identificar a melhor forma de se definir o tamanho da entrada.
  3. Verificar se o número de vezes que ela é executada pode variar para diferentes entradas de um mesmo tamanho. Se for o caso, deve-se analisar o melhor caso, pior caso e caso médio.
  4. Determinar uma função f(n) que representa o número de vezes que a função básica é executada em função do tamanho da entrada.

Introdução à Análise de Algoritmos

- Como analisar um algoritmo e descobrir qual a sua classe de complexidade?
  1. Identificar a operação básica do algoritmo.
  2. Identificar a melhor forma de se definir o tamanho da entrada.
  3. Verificar se o número de vezes que ela é executada pode variar para diferentes entradas de um mesmo tamanho. Se for o caso, deve-se analisar o melhor caso, pior caso e caso médio.
  4. Determinar uma função f(n) que representa o número de vezes que a função básica é executada em função do tamanho da entrada.
  5. Determinar a qual classe de complexidade f(n) pertence.

Introdução à Análise de Algoritmos

- Exemplo:

```
int maxElement(int v[],int n) {
    int mx = v[0];
    for(int i=1;i<n;i++)
        if (v[i] > mx)
            mx = v[i];
    return mx;
}
```

- Operação básica:
- Tamanho da entrada:
- O número de comparações é sempre o mesmo para arrays de mesmo tamanho.
- Número de comparações:
- Classe de complexidade:

Introdução à Análise de Algoritmos

- Exemplo:

```
int maxElement(int v[],int n) {
      int mx = v[0];
      for(int i=1;i<n;i++)
            if (v[i] > mx)
                  mx = v[i];
      return mx;
}
```

- Operação básica: comparação.
- Tamanho da entrada: número de elementos no array (n).
- O número de comparações é sempre o mesmo para arrays de mesmo tamanho.
- Número de comparações: $\sum_{1}^{n-1} 1$ = n-1
- Classe de complexidade: O(n)

Introdução à Análise de Algoritmos

- Exemplo:

```
int f ind(int v[],int n,int elem) {
    for(int i=0;i<n;i++)
        if (v[i] == elem)
            return i;
    return -1;
}
```

- Operação básica:
- Tamanho da entrada:
- O número de comparações varia dependendo do arranjo...
- Número de comparações no pior caso:
- Classe de complexidade:
- Exercício: como seria no melhor caso? e no caso médio (supondo que todo arranjo contém o elemento e que a probabilidade dele estar em uma posição é a mesma para todas posições)

Introdução à Análise de Algoritmos

- Exemplo:

```
int f ind(int v[],int n,int elem) {
	for(int i=0;i<n;i++)
		if (v[i] == elem)
			return i;
	return -1;
}
```

- Operação básica: comparação.
- Tamanho da entrada: número de elementos no array (n).
- O número de comparações varia dependendo do arranjo...
- Número de comparações no pior caso: $\sum_{0}^{n-1} 1$ = n
- Classe de complexidade: O(n)
- Exercício: como seria no melhor caso? e no caso médio (supondo que todo arranjo contém o elemento e que a probabilidade dele estar em uma posição é a mesma para todas posições) ?

Introdução à Análise de Algoritmos

- Exemplo:

```
int numDigitos(int n) { //suponha que n>0
	int ct  = 0;
	while( n != 0) {
		n/=10;
		ct++;
	}
	return ct;
}
```

- Operação básica:
- Tamanho da entrada:
- O “tamanho” de n é dividido por 10 em cada iteração... o número de comparações é aproximadamente:
- Classe de complexidade:

Introdução à Análise de Algoritmos

- Exemplo:

```
int numDigitos(int n) { //suponha que n>0
      int ct  = 0;
      while( n != 0) {
            n/=10;
            ct++;
      }
      return ct;
}
```

- Operação básica: divisão.
- Tamanho da entrada: o número n.
- O “tamanho” de n é dividido por 10 em cada iteração... o número de comparações é aproximadamente: log(n) (base 10)
- Classe de complexidade: O(log(n))

Introdução à Análise de Algoritmos

- Exemplo:

```
int matrixMult(int **a, int **b, int **c, int n) { //suponha que n>0
	for(int i=0;i<n;i++)
		for(int j=0;j<n;j++) {
			int resp = 0;
			for(int k=0;k<n;k++)
				resp += a[i][k]*b[k][j];
			c[i][j] = resp;
		}
}
```

- Operação básica:
- Tamanho da entrada:
- Número de multiplicações:
- Classe de complexidade:

Introdução à Análise de Algoritmos

- Exemplo:

```
int matrixMult(int **a, int **b, int **c, int n) { //suponha que n>0
	for(int i=0;i<n;i++)
		for(int j=0;j<n;j++) {
			int resp = 0;
			for(int k=0;k<n;k++)
				resp += a[i][k]*b[k][j];
			c[i][j] = resp;
		}
}
```

- Operação básica: multiplicação.
- Tamanho da entrada: o lado da matriz (n).
- Número de multiplicações: $\sum_{0}^{n-1}\sum_{0}^{n-1}\sum_{0}^{n-1} 1 = n^3$
- Classe de complexidade: $O(n^3)$

Introdução à Análise de Algoritmos

- No caso de algoritmos recursivos, a análise de complexidade é um pouco mais difícil de ser realizada.
- Estudos mais detalhados sobre este tema fogem do escopo desta disciplina (vamos trabalhar de forma muito informal a análise de algoritmos).
- Por enquanto, o importante é ter uma ideia de como funciona a análise de complexidade e reconhecer a complexidade de algoritmos simples.
- O conhecimento sobre a complexidade de algoritmos é muito importante: ajuda a encontrar gargalos em códigos, a escolher o algoritmo mais adequado para cada situação, etc.