Semana 2

Análise assintótica de eficiência

Analisar significa dividir algo em partes menores, e estudar estas partes separadamente. No caso de algoritmos, estamos interessados em analisar sua **eficiência** em termos de **tempo** (consumo de CPU) e **espaço** (consumo de memória). Nosso **foco** será na **análise de tempo**, pois nos computadores atuais a análise de tempo tem potencial muito maior de trazer benefícios que a análise de espaço, embora ela também seja importante em situações com limitação severa de memória.

2.1 Complexidade de algoritmos

Temos uma noção intuitiva de que o tempo de execução de um algoritmo depende do tamanho das estruturas de dados que são fornecidas como entrada. A **complexidade de tempo** é uma função que fornece o "tempo de execução" para cada "tamanho de entrada". Da mesma forma, a **complexidade de espaço** é uma função que fornece o "consumo de memória" para cada "tamanho de entrada". Precisamos então de uma definição precisa de **como medir** o **"tempo de execução"** e o **"tamanho da entrada"**.

2.1.1 Medição do tamanho da entrada

A forma mais geral de medir o tamanho da entrada é determinar o **total de bits** utilizados na sua representação binária, pois é desta forma que a informação é armazenada na memória do computador. Por exemplo, um número inteiro de valor n utiliza $b = \lfloor \log_2 n \rfloor + 1$ bits. Isto significa que o maior valor representado em b bits cresce exponencialmente com o aumento de b, pois n vale no máximo $2^b - 1$.

Quando a entrada é uma **coleção de elementos** com um número **máximo de bits por elemento**, podemos usar o **número de elementos** como tamanho da entrada. Por exemplo, se a entrada é um array com 100 inteiros de 32 bits, então o tamanho da entrada é $100 \cdot 32 = 3200$ bits, ou simplesmente 100 elementos, pois cada inteiro no array é limitado a 32 bits. Veremos na análise assintótica que podemos **ignorar fatores multiplicativos constantes**, e a diferença entre o total de bits e o número de elementos é apenas um fator multiplicativo. Utilizar o número de elementos é **mais simples**, pois nos poupa do trabalho de calcular o número exato de bits.

Quando a entrada **contém mais de uma estrutura**, todas elas devem ser consideradas na determinação do tamanho da entrada. Por exemplo, no problema da mochila recebemos um array de pesos com n números, um array de preços com n números, e um número T representando o tamanho da mochila. Se cada peso e cada preço tem um número máximo de bits, o tamanho da entrada depende de n. Como não fixamos um número máximo de bits para T, o tamanho da entrada depende também de $\log_2 T$ (tamanho da representação de T em binário).

Nos problemas envolvendo **grafos**, o tamanho da entrada depende da representação empregada, mas costuma-se utilizar **número de vértices** e o **número de arestas**. Vamos adotar esta simplificação na disciplina.

2.1.2 Medição do tempo de execução

Se utilizarmos uma medida de tempo convencional, como segundos, o valor medido vai depender da velocidade do processador, do sistema operacional e da linguagem de programação utilizada. Para avaliar apenas o algoritmo, é necessário **abstrair estas características do ambiente** onde o algoritmo é executado.

Então uma abordagem melhor é **contar o total de instruções executadas**, ou seja, determinamos quantas vezes cada instrução é executada em função do tamanho da entrada, e somamos estas contagens para todas as instruções do algoritmo. Observe que neste caso estamos abstraindo também o fato de que algumas instruções consomem mais tempo que outras, mas isto não é um problema visto que esta diferença está limitada por um fator multiplicativo constante.

Para simplificar a contagem de instruções, **contamos apenas** o número de execuções das instruções mais importantes, chamadas **operações básicas**. As operações básicas são aquelas que determinam a taxa de crescimento da complexidade, ou seja, aquelas que claramente são as mais executadas no algoritmo. Por exemplo, se o algoritmo consiste de vários comandos de repetição aninhados, as operações básicas estão dentro da repetição mais interna. Outras vezes as operações básicas são aquelas que estamos mais interessados, independente do algoritmo. Por exemplo, acessos ao banco de dados, comparações de elementos em algoritmos de ordenação, operações aritméticas em problemas numéricos.

2.1.3 Ordem de crescimento

Como o tempo de execução aumenta com o tamanho da entrada, estamos interessados em determinar a **taxa de crescimento da complexidade**, ou seja, o quão rápido ela cresce com o aumento do tamanho da entrada.

Na tabela abaixo temos o tempo de execução para cada tamanho de entrada n. Nas colunas temos funções de complexidade que ocorrem com frequência em análise de algoritmos. Para calcular os tempos, estamos assumindo um computador que executa 1 bilhão (10^9) de instruções por segundo. Notação: s para segundos, m para minutos, d para dias, a para anos, $1ns = 10^{-9}s$, $1\mu s = 10^{-6}s$, e $1ms = 10^{-3}s$.

$$\text{Tempo (segundos)} = \frac{\text{Complexidade (instruções)}}{\text{Velocidade (instruções/segundo)}}$$

n	$\lg n$	n	$n \lg n$	n^2	n^3	2^n	$n!$
10	$3ns$	$10ns$	$33ns$	$100ns$	$1\mu s$	$1\mu s$	$4ms$
10^2	$7ns$	$100ns$	$7\mu s$	$10\mu s$	$1ms$	$10^{13}a$	$10^{141}a$
10^3	$10ns$	$1\mu s$	$10\mu s$	$1ms$	$1s$		
10^4	$13ns$	$10\mu s$	$130\mu s$	$100ms$	$17m$		
10^5	$17ns$	$100\mu s$	$2ms$	$10s$	$12d$		
10^6	$20ns$	$1ms$	$20ms$	17 min	$32a$		

Para entrada pequena (tamanho $n = 10$), todas as complexidades produzem tempo de execução abaixo de 1 segundo. A complexidade $\lg n$ cresce tão lentamente que o algoritmo executa de forma praticamente instantânea para qualquer entrada de tamanho realista (não astronômico). As complexidades n e $n \lg n$ crescem tão lentamente que o tempo de execução permanece abaixo de 1 segundo mesmo para entrada de tamanho $n = 10^6$. Nesta maior entrada ($n = 10^6$), a complexidade n^2 ficou limitada a alguns minutos, e a complexidade n^3 consumiu alguns anos. Já as complexidades 2^n e $n!$ crescem tão rapidamente que o tempo de execução ultrapassa trilhões de anos mesmo para a entrada de tamanho 100. Obs.: a idade estimada da terra é $4{,}5 \cdot 10^9$ anos. Portanto, a tabela ilustra o grande **impacto do crescimento da função de complexidade na eficiência** do algoritmo.

2.1.4 Pior caso, melhor caso e caso médio

Além do tamanho da entrada, o tempo de execução pode **depender também dos valores recebidos como entrada**. Por exemplo, considere o algoritmo de busca sequencial descrito abaixo, e utilize como operação básica a comparação $A[i] \neq k$. Se o valor buscado k estiver na primeira posição de A, faremos apenas uma comparação. Por outro lado, se o valor k não estiver em A, faremos n comparações. Portanto, o tempo de execução depende do tamanho da entrada e dos valores armazenados em A e k.

Algoritmo: Busca_sequencial($A[1..n]$, k)

Entrada: Array $A[1..n]$, e um valor k.
Saída : Índice da 1a ocorrência de k em A, ou -1 se k não ocorre em A.

```
1 i = 1
2 while i ≤ n and A[i] ≠ k do
3     i = i + 1
4 if i ≤ n then return i
5 else return −1
```

A análise fica muito complicada se levar em conta os valores específicos recebidos como entrada. Para contornar este problema, limitamos a análise a alguns cenários. Na **análise de pior caso** consideramos a pior entrada possível de tamanho n, ou seja, a entrada que produz o maior tempo de execução dentre todas as entradas possíveis de tamanho n. Na **análise de melhor caso** temos a situação oposta, ou seja, consideramos a entrada que produz o menor tempo de execução dentre todas as entradas de tamanho n. No exemplo da busca

sequêncial, a pior entrada teria um valor k que não ocorre em A, e a melhor entrada teria o valor k na primeira posição de A. A **análise de pior caso é mais útil**, pois nos permite estabelecer um limite superior para o tempo de execução. Apesar disso, a análise de melhor caso pode revelar tipos de entrada úteis com bom desempenho. Por exemplo, a análise de melhor caso indica que, para arrays quase ordenados, o algoritmo de ordenação por inserção tem tempo linear.

Quando as entradas ruins são raras, a análise de pior caso pode fornecer uma complexidade muito pior que a observada em entradas "típicas". Neste caso, é mais informativo realizar uma **análise de caso médio**, que leva em conta entradas "aleatórias". Para realizar uma análise de caso médio precisamos fazer uma suposição adicional: a probabilidade de ocorrência de cada entrada possível. A suposição mais utilizada é considerar que **todas as entradas possíveis possuem a mesma chance** de ocorrer. No exemplo da busca sequêncial, vamos assumir que p é a chance do valor k estar no array A; e se k estiver no array, a chance da 1a ocorrência ocupar qualquer posição é a mesma $(1/n)$. Seja X a variável aleatória que representa o número de comparações $A[i] \neq k$ que o algoritmo realiza para uma entrada aleatória. Na análise de caso médio desejamos determinar o valor esperado $E[X]$.

$$E[X] = p \cdot \sum_{i=1}^{n} i \cdot \frac{1}{n} + (1-p) \cdot n = p \cdot \left(\frac{n+1}{2}\right) + (1-p) \cdot n.$$

Ou seja, sabendo que k está em A (fazendo $p = 1$), concluímos que em média testamos aproximadamente a metade do array ($(n+1)/2$); e sabendo que k não está em A (fazendo $p = 0$), concluímos que n comparações são feitas. Este exemplo ilustra como a **análise de pior caso é mais fácil que a análise de caso médio**. Nesta disciplina o foco será na análise de pior caso.

Na análise de algumas estruturas de dados encontramos também a **análise amortizada**, que consiste em considerar o **tempo de execução de um conjunto de operações**, ao invés de considerar o tempo de execução de operações isoladas. Esta abordagem é útil quando o pior tempo de execução de uma operação é alto, mas o pior tempo médio de um conjunto de operações é bem mais baixo. Por exemplo, suponha que uma operação de busca em uma estrutura de dados tem complexidade igual a n (onde n é o número de elementos armazenados), mas a complexidade para realizar n buscas é $n \lg n$, ou seja, em média executamos $\lg n$ instruções por busca.

2.2 Notação assintótica

Veja os vídeos **"crescimento assintótico"** e **"notação assintótica"**.

Leitura complementar

- Seções 2.2, 2.3, 2.4 e 2.9.2 do Skiena.
- Capítulo 14 do Fleck:

  `http://mfleck.cs.illinois.edu/building-blocks/updates-fa2017/big-o.pdf`

- Seção 3.1 do Cormen.