Lista 1 - Resolução

1. Utilizando o Princípio de Indução Finita, prove que as igualdades abaixo são verdadeiras.

   (a) $1 + q + q^2 + \ldots + q^n = \frac{1-q^{n+1}}{1-q}$.

$$\begin{aligned} 1 + q + \ldots + q^n + q^{n+1} &= \frac{1-q^{n+1}}{1-q} + q^{n+1} \\ &= \frac{1-q^{n+1}}{1-q} + \frac{(1-q)q^{n+1}}{1-q} \\ &= \frac{1-q^{n+1}+q^{n+1}-q^{n+2}}{1-q} \\ &= \frac{1-q^{n+2}}{1-q} \square \end{aligned}$$

   (b) $\sum_{i=1}^{n} i^3 = \left[\frac{n(n+1)}{2}\right]^2$.

$$\begin{aligned} \sum_{i=1}^{n} i^3 + (n+1)^3 &= \left[\frac{n(n+1)}{2}\right]^2 + (n+1)^3 \\ &= \frac{n^2(n+1)^2}{2^2} + \frac{2^2(n+1)^3}{2^2} \\ &= \frac{n^2(n+1)^2 + 2^2(n+1)^2(n+1)}{2^2} \\ &= \frac{(n+1)^2[n^2 + 2^2(n+1)]}{2^2} \\ &= \frac{(n+1)^2(n^2+4n+4)}{2^2} \\ &= \frac{(n+1)^2(n+2)^2}{2^2} \\ &= \left[\frac{(n+1)(n+2)}{2}\right]^2 \square \end{aligned}$$

   (c) $n = 3a + 5b$, para $n \geq 8$.

   Base:

$$S(8) = 1 \times 3 + 1 \times 5$$

$$S(9) = 3 \times 3 + 5 \times 0$$

$$S(10) = 0 \times 3 + 2 \times 5$$

Indução (para n): para $n \geq 10$, os números podem ser escritos como uma combinação qualquer de a e b tal que $a \times 3 + b \times 5 = n$. Para $n = 10$: $0 \times 3 + 2 \times 5$.

Passo de indução (para $n + 1$): iremos provar que a indução é válida para $n + 1$. Para tal, vamos subtrair 3 e considerar o caso de $n - 2$. Como o valor mínimo de n é 10, temos a garantia que $n - 2$ é contemplado por um dos casos-base. Sendo assim, seja $n - 2 = a \times 3 + b \times 5$. Somando 3 dos dois lados, temos que

$$\begin{aligned} n + 1 &= a \times 3 + b \times 5 + 3 \\ n + 1 &= (a + 1) \times 3 + b \times 5 \square \end{aligned}$$

2. Considere um vetor V com n elementos inteiros. Considere também que uma operação de *swap* é definida como a troca de um par de elementos do vetor *em qualquer posição*. Descreva um algoritmo que, dado o vetor V, retorna o número *mínimo* de trocas necessárias para ordená-lo. Qual a complexidade desse algoritmo?

   Resposta: Para resolver essa questão, basta fazer a implementação do *Selection sort*. Apesar de ser um algoritmo com complexidade $O(n^2)$, o número de trocas é sempre o menor (o que ajuda a mostrar que o número de trocas não está relacionado à eficiência de tempo).

3. Considere um vetor V com n elementos inteiros. Considere também que uma operação de *swap* é definida como *remover um elemento de sua posição inicial e colocá-lo no final do vetor*. Descreva um algoritmo que, dado o vetor V, retorna o número *mínimo* de trocas necessárias para ordená-lo. Qual a complexidade desse algoritmo? Em qual caso especial essa complexidade se torna $O(n)$?

   Resposta: A melhor complexidade para o caso geral é $O(n \log n)$. Para tal, é necessário criar uma cópia do vetor, ordená-lo usando algum algoritmo de complexidade $O(n \log n)$ e então percorrer os dois vetores buscando quantos elementos estão fora de ordem (isso é feito em tempo linear, e o resultado dessa verificação nos dá quantos *swaps* serão necessários). O caso especial em que a complexidade se torna $O(n)$ é quando sabemos de antemão exatamente quais números estão no vetor e como ele deve ficar ordenado (por exemplo, um vetor com n elementos inteiros consecutivos). Nesse caso, basta apenas fazer a verificação explicada anteriormente, que é linear.

4. Sabemos que o algoritmo *Quicksort* possui complexidade $O(n \log n)$ no caso médio. No entanto, no pior caso a complexidade aumenta para $O(n^2)$. Descreva qual é o cenário do pior caso e para qual algoritmo clássico de ordenação o *Quicksort* "degrada" nessa situação.

Resposta: O pior cenário acontece quando o *Quicksort* escolhe pivôs ruins; isto é, pivôs que tornam as divisões desbalanceadas. Na pior das hipóteses, o *Quicksort* pode escolher sempre o menor (ou maior) elemento do vetor como pivô. Neste caso, ele "degrada" para o *Insertion sort*.

5. Sabemos que para calcular o n-ésimo elemento da sequência de Fibonacci, podemos usar tanto uma abordagem recursiva, com complexidade $O(2^n)$, quanto uma abordagem iterativa, com complexidade $O(n)$. Entre recursão e iteração, há sempre um ganho entre uma abordagem e outra? Mais ainda, todo algoritmo que possui uma versão recursiva possui também uma equivalente iterativa?

   Resposta: Nem sempre há ganho entre uma abordagem recursiva e iterativa do mesmo algoritmo (basta pensar no caso do fatorial - ambas as abordagens são $O(n)$). E sim, todo algoritmo que possui uma forma recursiva também possui uma forma iterativa. Intuitivamente, podemos imaginar que a recursão pode ser "aberta" em iterações, sendo o caso-base o critério de parada. A prova pode ser feita usando-se a tese de Church-Turing.

6. As seguintes equivalências são válidas? Prove.

   (a) $2^{n+1} = O(2^n)$.

$$\begin{aligned} 2^{n+1} &\leq c2^n \\ 2^n 2 &\leq c2^n \\ 2 &\leq c \end{aligned}$$

   Para $c \geq 2$ e $n \geq n_0 = 0$, a igualdade é válida. Logo, 2^{n+1} é $O(2^n)$.

   (b) $2^{2n} = O(2^n)$.

$$\begin{aligned} 2^{2n} &\leq c2^n \\ \frac{2^{2n}}{2^n} &\leq \frac{c2^n}{2^n} \\ 2^n &\leq c \end{aligned}$$

   Não existe uma constante c tal que $2^n \leq c$ para $n \geq n_0$, pois 2^n sempre irá crescer, ao contrário de c, que permanecerá sempre com o mesmo valor.

   (c) $n^3 \log n = \Omega(n^3)$.

$$\begin{aligned} n^3 \log n &\geq cn^3 \\ \log n &\geq c \end{aligned}$$

Para $n \geq n_0 = 2$ e $c = 1$, a igualdade é válida. Logo, $n^3 \log n$ é $\Omega(n^3)$.

7. Implemente os algoritmos de busca sequencial e busca binária. Qual a complexidade de cada abordagem? Quais as vantagens de uma em relação à outra? Ambas funcionam sem restrição quanto ao vetor de entrada?

   Resposta: A complexidade da busca sequencial é $O(n)$, e a da busca binária, $O(\log_2 n)$. A busca binária é muito mais rápida que a busca sequencial; no entanto, para a busca binária funcionar, os elementos precisam estar ordenados, o que não é necessário na busca sequencial. Mesmo assim, a busca binária normalmente é usada em situações onde muitas *queries* são feitas no mesmo conjunto de elementos. Sendo assim, vale mais a pena fazer a ordenação em complexidade $O(n \log n)$ e fazer consultas com complexidade $O(\log_2 n)$ do que ficar percorrendo o vetor com complexidade $O(n)$ a cada nova busca.

8. (Desafio) Resolva o seguinte problema: http://www.urionlinejudge.com.br/repository/UOJ_1563.html. Qual a complexidade final do seu algoritmo?

   Resposta: A complexidade do algoritmo pra ser aceito nesse problema é $O(\sqrt{n})$.