1. Show that $7n^2 - 2n + 2$ is not $O(n)$.
   Solution: We prove by contradiction. If $7n^2 - 2n + 2$ is $O(n)$, then there exist constants k and C such that "if $n > k$, then $7n^2 - 2n + 2 \le Cn$. When $n > 0$, we can divide both sides of the inequality by n and we get: $7n - 2 + \frac{2}{n} \le C$. Since C is a constant, $7n + \frac{2}{n} \le C + 2$ cannot hold for all $n > k$. This contradiction shows that $7n^2 - 2n + 2$ is not $O(n)$.

- **[Big-Omega Notation]** Let f and g be functions from the set of integers or the set of real numbers to the set of real numbers. We say that $f(x)$ is $\Omega(g(x))$ if there are positive constants C and k such that
  $$|f(x)| \ge C|g(x)|$$
  whenever $x > k$. This is read as "$f(x)$ is big − Omega of $g(x)$."
  - In other words, when x is large enough, $f(x)$ is lower bounded by a constant time $g(x)$.
  - Big-Omega notation is called the ***asymptotic lower bound*** of a function.

- **[Big-Theta Notation]** Let f and g be functions from the set of integers or the set of real numbers to the set of real numbers. We say that $f(x)$ is $\Theta(g(x))$ if $f(x)$ is $O(g(x))$ and $f(x)$ is $\Omega(g(x))$. When $f(x)$ is $\Theta(g(x))$, we say that "$f(x)$ is big − Theta of $g(x)$".
  - Big-Theta notation is called the ***asymptotic tight bound*** of a function.

Comparing the growing speed of two functions

- If $f(x)$ is $O(g(x))$, we say that $g(x)$ grows at least as fast as $f(x)$.
  - In other words, $g(x)$ either grows faster than $f(x)$ or of the same order as $f(x)$.
- If $f(x)$ is not $O(g(x))$, we say that $f(x)$ grows strictly faster than $g(x)$.
  - In other words, when x is large, $f(x)$ cannot be upper bounded by any constant time of $g(x)$.
- Similarly, if $f(x)$ is $\Omega(g(x))$, we say that$f(x)$ grows at least as fast as $g(x)$; if $f(x)$ is not $\Omega(g(x))$, we say that $g(x)$ grows strictly faster than $f(x)$.
- If $f(x)$ is $\Theta(g(x))$, then $f(x)$ and $g(x)$ are of the same order (they grow at "similar" speed: only differ by some constant times).

- **[Property]** If $f(x)$ is $O(g(x))$, then $f(x) + g(x) = \Theta(g(x))$.

2. Show that $f(n) = n^5 + 10^{20} n$ is $\Theta(n^5)$.

   Solution: we can use the property above. $10^{20} n$ is $O(n^5)$ since when $n > 10^5$, $10^{20} n \le n^5$. Thus, $n^5 + 10^{20} n = \Theta(n^5)$.

- We have the following observations:
  - A polynomial of higher degree grows faster than a polynomial of lower degree.
  - An exponential (the constant is greater than 1) grows faster than a polynomial.
  - A polynomial (degree > 0) grows faster than a logarithm.
  - A logarithm grows faster than constant.

3. True or false.

a.	n^2 is $\Theta(2n)$	False, n^2 is $\Omega(2n)$ but not $O(2n)$
b.	$\log n$ is $\Omega(\log\log n)$	$\log n = \log_{10} n$. Let $k = \log n$, k is $\Omega(\log k)$, true

c. $\log n$ is $\Theta(\lg n)$ — $\lg n = \log_2 n$. True. $\log n = \frac{\lg n}{\lg 10}$

d. 2^n is $O(n^n)$ — True. $2 \times 2 \times 2 \ldots \times 2 < n \times n \times n \ldots \times n$

e. $n!$ is $\Omega(2^n)$ — True. LHS $= 1 \times 2 \times 3 \ldots \times (n-1) \times n$, RHS$= 2 \times 2 \times 2 \ldots \times 2$, LHS$\times\, 2 > RHS$

f. 4000 is $\Theta(1)$ — True

g. $2^{\log n}$ is $\Theta(2^{\lg n})$ — $a^{\log_b c} = c^{\log_b a}$, LHS$= n^{\log 2} < n$, RHS$= n^{\lg 2} = n$, False

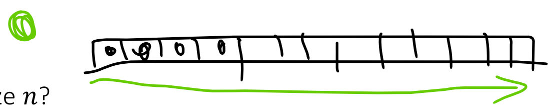
h. 2^n is $\Theta(4^n)$ — False. $4^n = (2^n)^2$, let $k = 2^n$, then k is NOT $\Theta(k^2)$

i. n^3 is $\Omega(3n^3 + 3n^2 + 10000n)$ — True. $3n^3 + 3n^2 + 10000n$ is $\Theta(n^3)$, then n^3 is also $\Theta(3n^3 + 3n^2 + 10000n)$

j. $n^2 + n \log n$ is $\Theta(n^2)$ — if we compare n^2 and $n \log n$, we are actually comparing n and $\log n$. True

Searching and Sorting

- **Searching** is a method that is available in most of the data structures. Given data in a structure and a target element, the method either finds the location of the element (or tells the user that such element does not exist).
- **Linear search** is a searching algorithm that is used in a "linear" data structure. It checks each element in the structure one by one.



4. What is the time complexity of linear search in a list/array of size n?

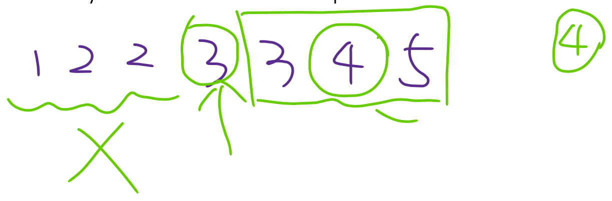
- If we are really unlucky, we will need to go through the whole list to find out that the given element is at the last spot or even doesn't exist in the list, thus, in the **worst case**, linear search needs n operations, which is $\Theta(n)$.

- The above time complexity analysis is a **worst-case** analysis. By the worst-case performance of an algorithm, we mean the largest number of operations needed to solve the given problem using this algorithm on input of specified size. Worst-case analysis tells us how many operations an algorithm requires to guarantee that it will produce a solution.
- There is also **best-case** analysis. The best-case performance of an algorithm is the minimum possible number of steps an algorithm needed to produce a solution.
- Another important type of complexity analysis, besides worst-case analysis, is called **average-case** analysis. The average number of operations used to solve the problem over all possible inputs of a given size is found in this type of analysis. Average case time complexity analysis is usually much more complicated than worst-case analysis.

- Note that, it is WRONG to relate worst-case with $O(.)$, average-case with $\Theta(.)$ and best-case with $\Omega(.)$. For example, the worst-case time complexity of linear search can be $\Theta(n)$.

$3\ 3\ 4\ 5\ 1\ 2\ 2 \rightarrow 1\ 2\ 2\ 3\ 3\ 4\ 5$

- **Sorting** in a list/array is to rearrange the elements into a certain order (usually non-decreasing on some value).
- Sorting brings us a lot of convenience. For example, after sorting, we can use **binary search:** it is like how we look up a word in a dictionary, if we find the value of the target is too large compared to the element at current spot, then every element before this spot is also too small.



```
**BINARY-SEARCH** $(t, A[r \ldots p])$
$low = r$
$high = p$
**while** $(low < high)$
        $mid = \lfloor (low + high)/2 \rfloor$
        **if** $t > A[mid]$
                $low = mid + 1$
        **else** $high = mid$
**if** $A[high] == t$
        **return** $high$
**else** **return** $-1$
```

5. What is the worst-case time complexity of binary search in a list of size n?

- Before the first iteration of the while loop, we have $high - low = n - 1$, and after the while loop, we have $high - low = 0$. And in each iteration of the while loop, what value of $high - low$ is halved. Thus, there are $\lg n$ iterations. Since there are constant operations in each iteration, the worst-case time complexity of binary search is $\Theta(\lg n)$.

size left: $\frac{n}{1}$ $=2^0$ $\quad \frac{n}{2}$ $=2^1$ $\quad \frac{n}{4}$ $=2^2$ $\quad \frac{n}{8}$ $=2^3$ $\quad \ldots\ldots \quad \frac{n}{2^{j-1}} \quad 1 = \frac{n}{n}$

$n = 2^{\lg n}$

iteration: $1 \quad 2 \quad 3 \quad 4 \quad \ldots \quad j \quad \boxed{?} \Leftarrow \lg n + 1$