Algorithm Analysis

Fundamental Concepts

Dr. Bahadır Zeren

Algorithm

An algorithm is a procedure to accomplish a specific task and has a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time.

Algorithms’ properties

Input: there are zero or more quantities, which are externally supplied;

Output: at least one quantity is produced;

Definiteness: each instruction must be clear and unambiguous;

Finiteness: if we trace out the instructions of an algorithm, then for all cases the algorithm will terminate after a finite number of steps;

Effectiveness: every instruction must be sufficiently basic that it can in principle be carried out by a person using only pencil and paper. It is not enough that each operation be definite, but it must also be feasible.

Types of algorithms

Brute Force Algorithm: It is the simplest approach for a problem. A brute force algorithm is the first approach that comes to finding when we see a problem.

Recursive Algorithm: A recursive algorithm is based on recursion. In this case, a problem is broken into several sub-parts and called the same function again and again.

Backtracking Algorithm: The backtracking algorithm basically builds the solution by searching among all possible solutions. Using this algorithm, we keep on building the solution following criteria. Whenever a solution fails we trace back to the failure point and build on the next solution and continue this process till we find the solution or all possible solutions are looked after.

Searching Algorithm: Searching algorithms are the ones that are used for searching elements or groups of elements from a particular data structure. They can be of different types based on their approach or the data structure in which the element should be found.

Sorting Algorithm: Sorting is arranging a group of data in a particular manner according to the requirement. The algorithms which help in performing this function are called sorting algorithms. Generally sorting algorithms are used to sort groups of data in an increasing or decreasing manner.

Types of algorithms

Hashing Algorithm: Hashing algorithms work similarly to the searching algorithm. But they contain an index with a key ID. In hashing, a key is assigned to specific data.

Divide and Conquer Algorithm: This algorithm breaks a problem into sub-problems, solves a single sub-problem and merges the solutions together to get the final solution. It consists of the following three steps:

Divide
Solve
Combine

Greedy Algorithm: In this type of algorithm the solution is built part by part. The solution of the next part is built based on the immediate benefit of the next part. The one solution giving the most benefit will be chosen as the solution for the next part.

Dynamic Programming Algorithm: This algorithm uses the concept of using the already found solution to avoid repetitive calculation of the same part of the problem. It divides the problem into smaller overlapping subproblems and solves them.

Randomized Algorithm: In the randomized algorithm we use a random number so it gives immediate benefit. The random number helps in deciding the expected outcome.

Algorithms’ main performance metrics

Time Complexity: The time needed by an algorithm expressed as a function of the size of a problem is called the time complexity of the algorithm. The time complexity of a program is the amount of computer time it needs to run to completion.

Space Complexity: The space complexity of a program is the amount of memory it needs to run to completion.

Components of space complexity

Instruction space: Instruction space is the space needed to store the compiled version of the program instructions.

Data space: Data space is the space needed to store all constant and variable values. Data space has two components:

- Space needed by constants and simple variables in program.
- Space needed by dynamically allocated objects such as arrays and class instances.

Environment stack space: The environment stack is used to save information needed to resume execution of partially completed functions.

Instruction Space: The amount of instructions space that is needed depends on factors such as:

- The compiler used to complete the program into machine code.
- The compiler options in effect at the time of compilation
- The target computer.

Main algorithm design goals

- Try to save Time
- Try to save Space

Classifications of algorithms

If “*n*‟ is the number of data items to be processed:

1: Instructions of most programs are executed once or at most only a few times.

Log *n*: When the running time of a program is logarithmic, the program gets slightly slower as *n* grows. This running time commonly occurs in programs that solve a big problem by transforming it into a smaller problem, cutting the size by some constant fraction. Whenever *n* doubles, log n increases by a constant, but log *n* does not double until *n* increases to *n*^2.

n: When the running time of a program is linear, it is generally the case that a small amount of processing is done on each input element.

n*.log *n: This running time arises for algorithms that solve a problem by breaking it up into smaller sub-problems, solving then independently, and then combining the solutions. When *n* doubles, the running time more than doubles.

Classifications of algorithms

***n*^2**: When the running time of an algorithm is quadratic, it is practical for use only on relatively small problems. Quadratic running times typically arise in algorithms that process all pairs of data items (perhaps in a double nested loop) whenever *n* doubles, the running time increases four fold.

***n*^3**: Similarly, an algorithm that process triples of data items (perhaps in a triple–nested loop) has a cubic running time and is practical for use only on small problems. Whenever *n* doubles, the running time increases eight fold.

2^*n*: Exponential time complexity arises for functions like enumerating all subsets of n items. Whenever *n* doubles, the running time squares.

n!: Factorial time complexity arises with functions like generating all permutations or orderings of n items is needed.

Understanding time complexity O(1)

- Accessing Array Index (int a = ARR[5];)
- Inserting a node in Linked List
- Pushing and Poping on Stack
- Insertion and Removal from Queue
- Finding out the parent or left/right child of a node in a tree stored in Array
- Jumping to Next/Previous element in Doubly Linked List

```
function swap(a, b):
   temp = a              1
   a = b                 1
   b = temp              1

       T(N)=3
```

Understanding time complexity O(n)

- Traversing an array
- Traversing a linked list
- Linear Search
- Deletion of a specific element in a Linked List (Not sorted)
- Comparing two strings
- Checking for Palindrome
- Counting/Bucket Sort and here too you can find a million more such examples....

```
for (int i = 0; i < n; i++)     n+1
   op();                        n
```

T(N)=2n+1

Understanding time complexity O(log n)

- Binary Search
- Finding largest/smallest number in a binary search tree
- Certain Divide and Conquer Algorithms based on Linear functionality
- Calculating Fibonacci Numbers - Best Method The basic premise here is NOT using the complete data, and reducing the problem size with every iteration

```
for (int i = 1; i <= n; i = 2*i)      log(n)+2
   op();                              log(n)+1
```

```
T(N)=2log(n)+3
```

Understanding time complexity O(n.log n)

- Merge Sort
- Heap Sort
- Quick Sort
- Certain Divide and Conquer Algorithms based on optimizing O(n^2) algorithms

```
for (int i = 1; i <= n; i++)         n+1
   for (int j = 1; j <= n; j = 2*j)  n.(log(n)+2)
      op();                          n.log(n)+n
```

T(N)=2nlog(n)+4n+1

Understanding time complexity O(n^2)

- Bubble Sort
- Insertion Sort
- Selection Sort
- Traversing a simple 2D array

```
for (int i = 0; i < n; i++)          n+1
   for (int j = i; j < n; j++)       n.(n+1)/2+n
      op();                          n.(n+1)/2
```

T(N)=n^2+3n+1

Understanding time complexity O(n^3)

- enumerate all triples
- Floyd–Warshall

```
for (int i = 0; i < n; i++)
   for (int j = i+1; j < n; j++)
      for (int k = j+1; k < n; k++)
         op();
```

Understanding time complexity O(2^n)

```
function powerset(n = '') {
  const array = Array.from(n);
  const base = [''];

  const results = array.reduce((previous, element) => {
    const previousPlusElement = previous.map(el => {
      return `${el}${element}`;
    });
    return previous.concat(previousPlusElement);
  }, base);

  return results;
}
```

```
powerset('') // ...
// n = 0, f(n) = 1;
powerset('a') // , a...
// n = 1, f(n) = 2;
powerset('ab') // , a, b, ab...
// n = 2, f(n) = 4;
powerset('abc') // , a, b, ab, c, ac, bc, abc...
// n = 3, f(n) = 8;
powerset('abcd') // , a, b, ab, c, ac, bc, abc, d, ad, bd, abd, cd, acd, bcd...
// n = 4, f(n) = 16;
powerset('abcde') // , a, b, ab, c, ac, bc, abc, d, ad, bd, abd, cd, acd, bcd...
// n = 5, f(n) = 32;
```

Understanding time complexity O(n!)

- Solving the travelling salesman problem via brute-force search
- generating all unrestricted permutations of a partially ordered set;
- finding the determinant with Laplace expansion
- enumerating all partitions of a set

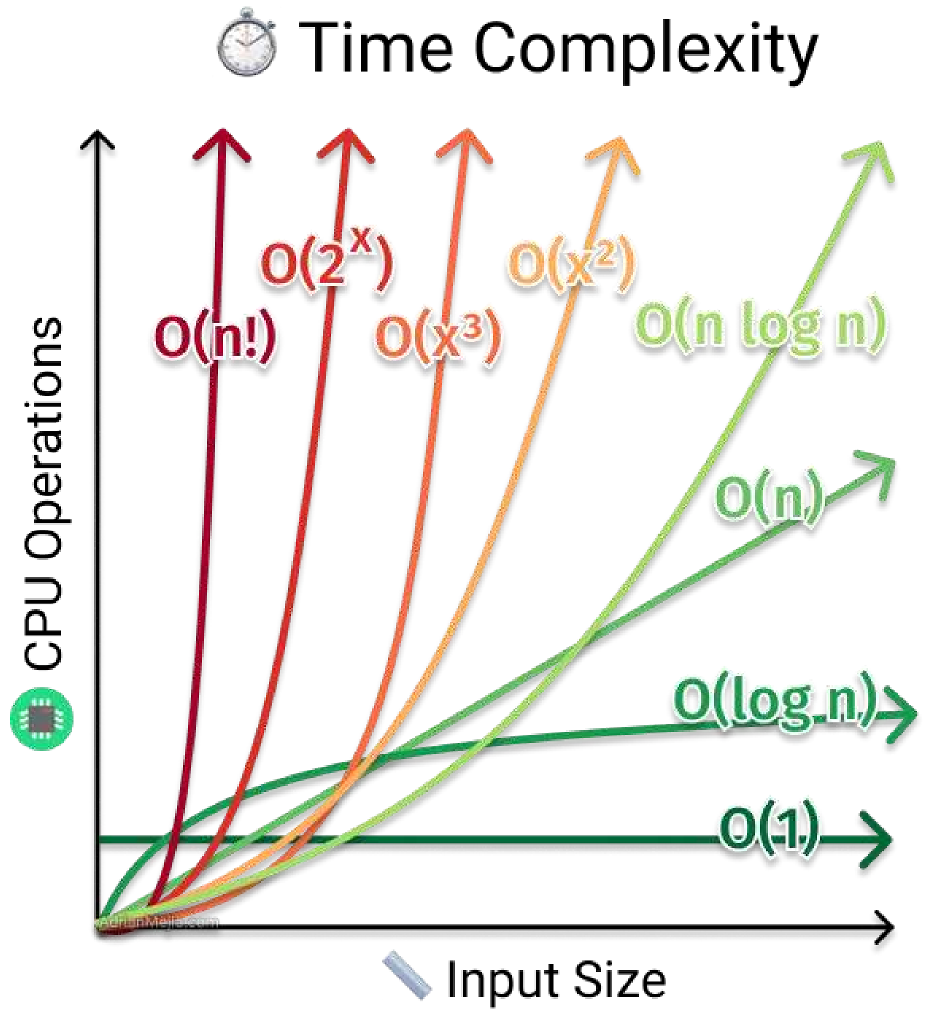
```
function getPermutations(string, prefix = '') {
  if(string.length <= 1) {
    return [prefix + string];
  }

  return Array.from(string).reduce((result, char, index) => {
    const reminder = string.slice(0, index) + string.slice(index+1);
    result = result.concat(getPermutations(reminder, prefix + char));
    return result;
  }, []);
}
```

```
getPermutations('ab') // ab, ba...
// n = 2, f(n) = 2;
getPermutations('abc') // abc, acb, bac, bca, cab, cba...
// n = 3, f(n) = 6;
getPermutations('abcd') // abcd, abdc, acbd, acdb, adbc, adcb, bacd...
// n = 4, f(n) = 24;
getPermutations('abcde') // abcde, abced, abdce, abdec, abecd, abedc, acbde...
// n = 5, f(n) = 120;
```

Classifications of algorithms

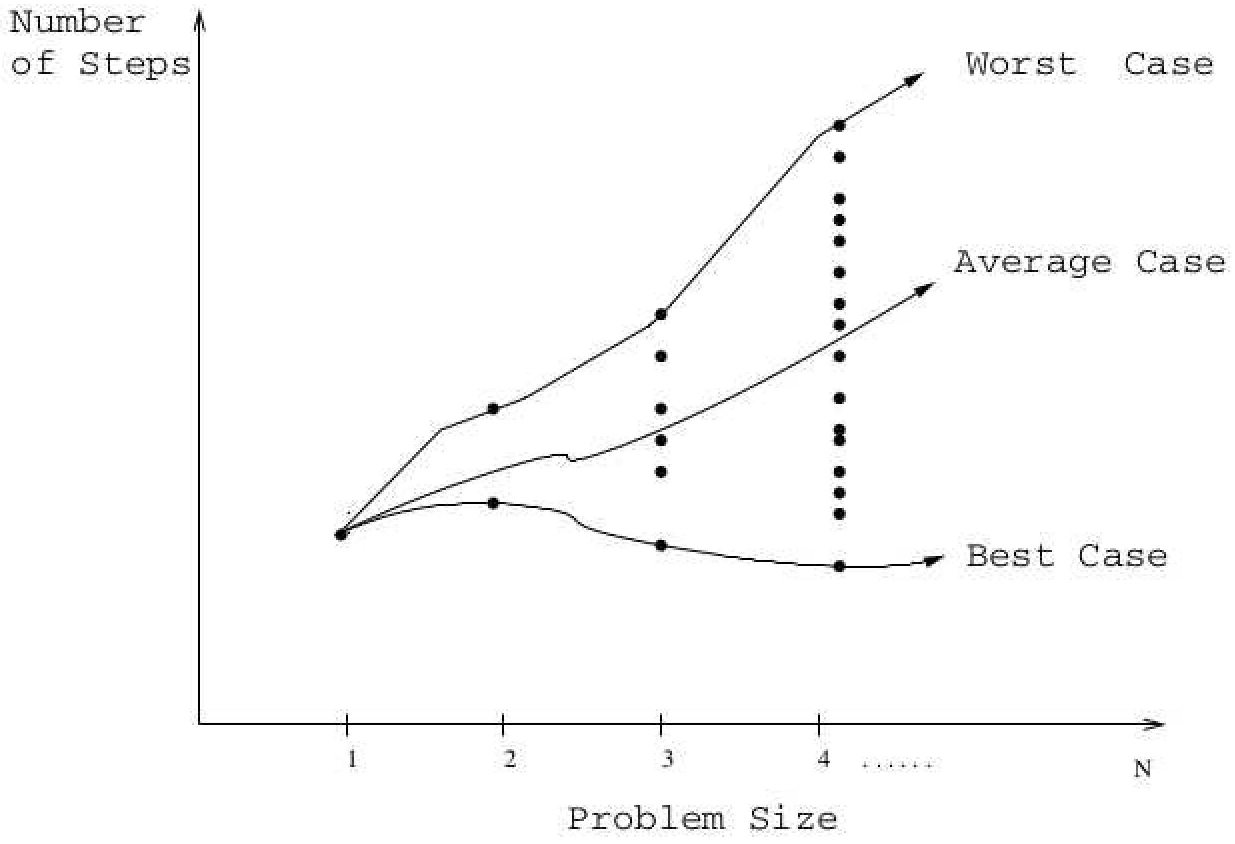
$O(1) < O(\log_2 n) < O(n) < O(n.\log_2 n) < O(n^2) < O(n^3) < O(2^n) < n! < n^n$



Complexity cases of algorithms

If the function *f(n)*, gives the running time of an algorithm:

Best case : The minimum possible value of *f(n)* is called the best case.
Average case : The expected value of *f(n)*.
Worst case : The maximum value of *f(n)* for any key possible input.



Rate of growth (notations)

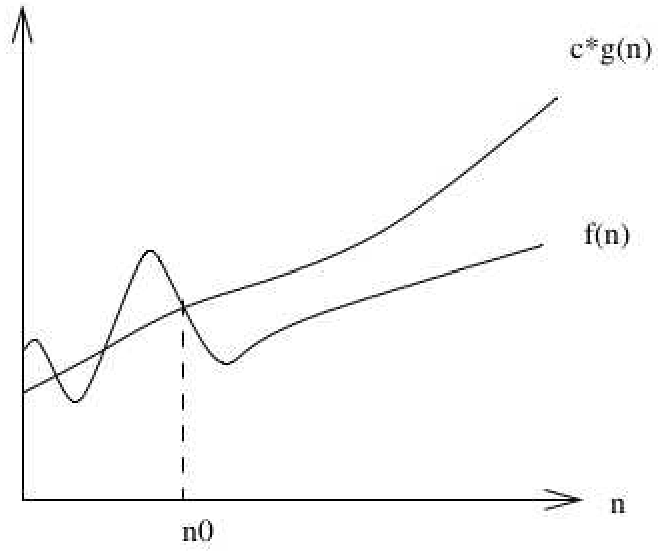
Big–OH (O): Worst case

Big–OMEGA (Ω): Best case

Big–THETA (Ө): Average case

Big-OH O (Upper bound)

f(n) = O(g(n)) means *c · g(n)* is an upper bound on *f(n)*. Thus there exists some constant *c* such that *f(n)* is always ≤ *c · g(n)*, for large enough *n*.



$T(n) = 12754.n^2 + 4353.n + 834.lg_2 n + 13546$

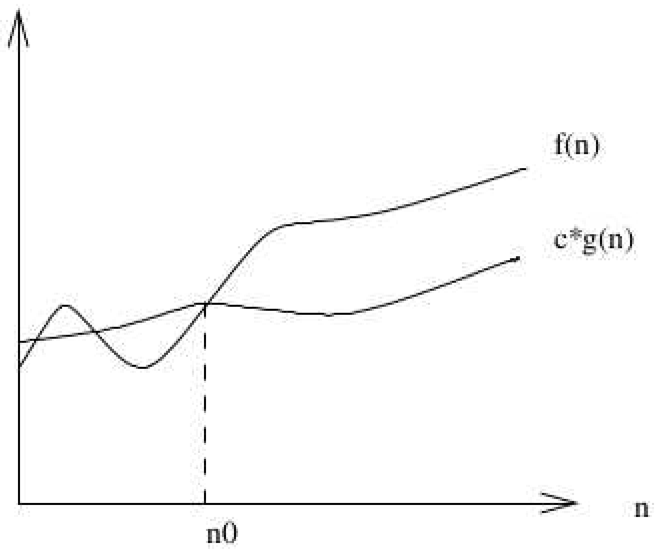
$$\Downarrow$$

$O(n^2)$

"the time grows quadratically with n."

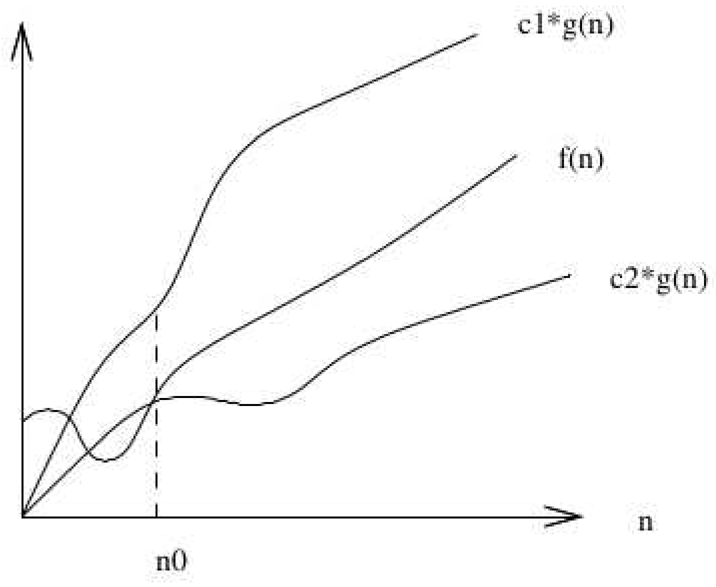
Big-Omega Ω (Lower bound)

f(n) = Ω(g(n)) means *c · g(n)* is a lower bound on *f(n)*. Thus there exists some constant *c* such that *f(n)* is always ≥ *c · g(n)*, for all *n* ≥ *n0*.



Big-Theta Ɵ (Same order)

f(n) = Ɵ(g(n)) means *c1 · g(n)* is an upper bound on *f(n)* and *c2 · g(n)* is a lower bound on *f(n)*, for all *n* ≥ *n0*. Thus there exist constants *c1* and *c2* such that *f(n)* ≤ *c1 · g(n)* and *f(n)* ≥ *c2 · g(n)*. This means that *g(n)* provides a nice, tight bound on *f(n)*.



Some questions ?

$3n^2 - 100n + 6 = O(n^2)$, because I choose $c = 3$ and $3n^2 > 3n^2 - 100n + 6$;

$3n^2 - 100n + 6 = O(n^3)$, because I choose $c = 1$ and $n^3 > 3n^2 - 100n + 6$ when $n > 3$;

$3n^2 - 100n + 6 \neq O(n)$, because for any c I choose $c \times n < 3n^2$ when $n > c$;

$3n^2 - 100n + 6 = \Omega(n^2)$, because I choose $c = 2$ and $2n^2 < 3n^2 - 100n + 6$ when $n > 100$;

$3n^2 - 100n + 6 \neq \Omega(n^3)$, because I choose $c = 3$ and $3n^2 - 100n + 6 < n^3$ when $n > 3$;

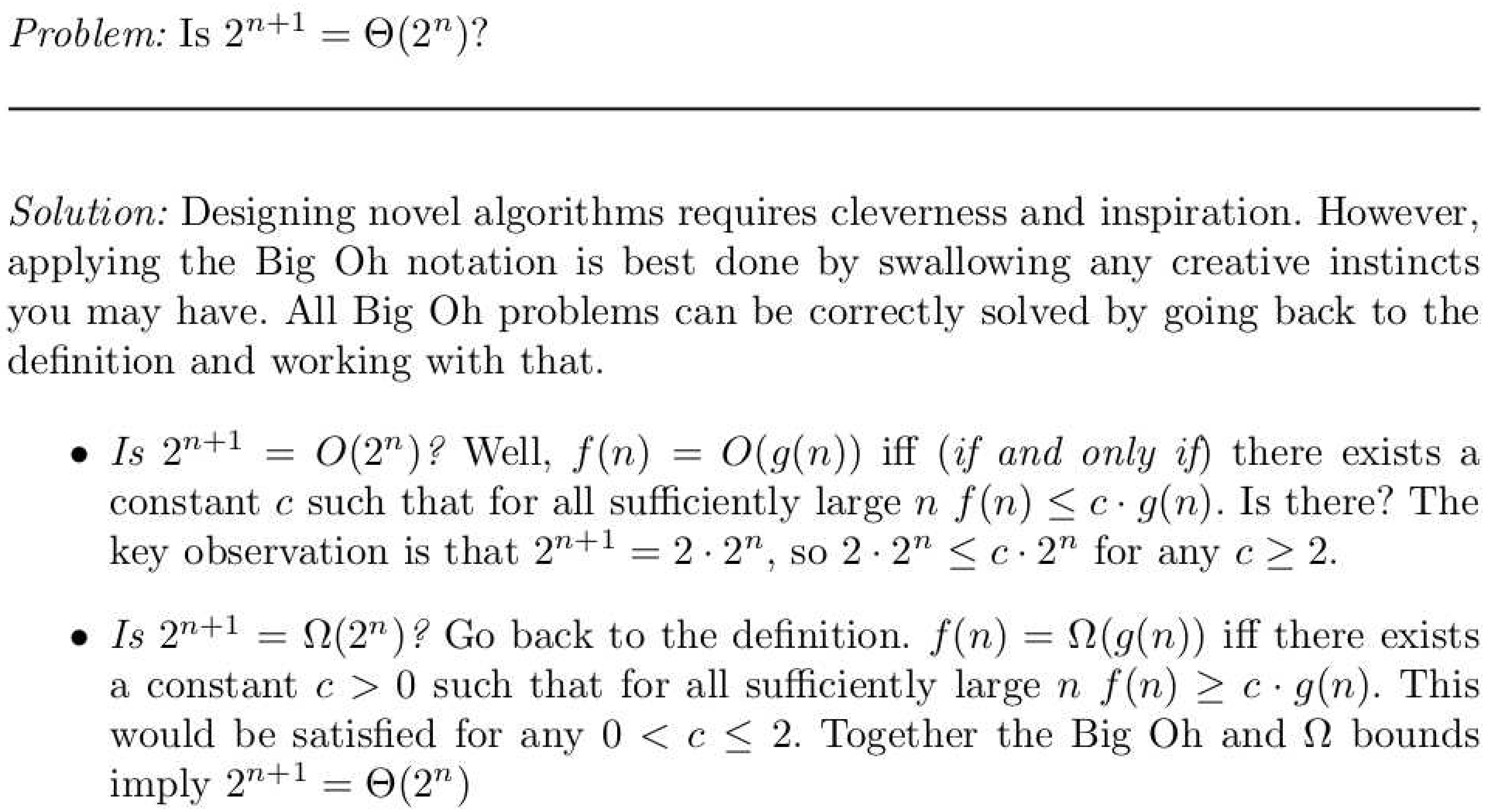
$3n^2 - 100n + 6 = \Omega(n)$, because for any c I choose $cn < 3n^2 - 100n + 6$ when $n > 100c$;

$3n^2 - 100n + 6 = \Theta(n^2)$, because both O and Ω apply;

$3n^2 - 100n + 6 \neq \Theta(n^3)$, because only O applies;

$3n^2 - 100n + 6 \neq \Theta(n)$, because only Ω applies.

Some questions ?



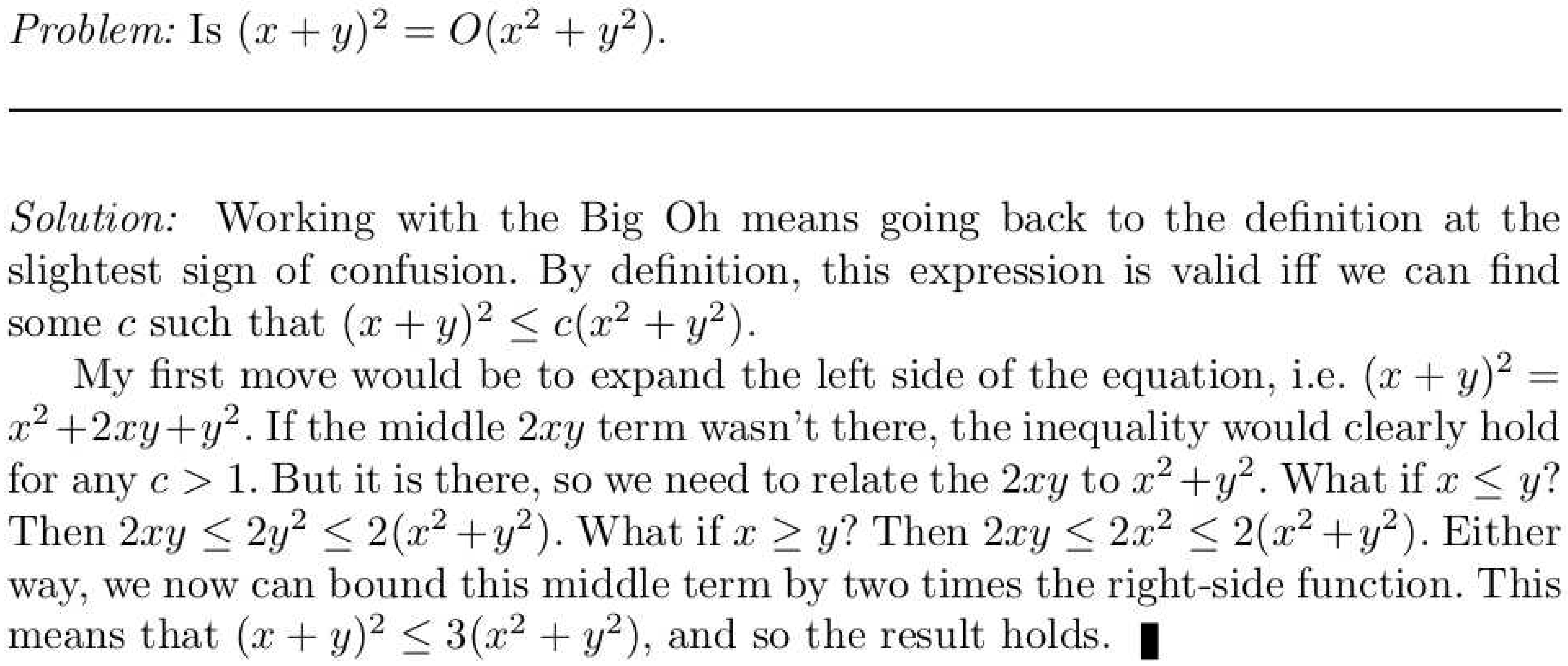
Problem: Is $2^{n+1} = \Theta(2^n)$?

Solution: Designing novel algorithms requires cleverness and inspiration. However, applying the Big Oh notation is best done by swallowing any creative instincts you may have. All Big Oh problems can be correctly solved by going back to the definition and working with that.

- *Is $2^{n+1} = O(2^n)$?* Well, $f(n) = O(g(n))$ iff (*if and only if*) there exists a constant c such that for all sufficiently large n $f(n) \leq c \cdot g(n)$. Is there? The key observation is that $2^{n+1} = 2 \cdot 2^n$, so $2 \cdot 2^n \leq c \cdot 2^n$ for any $c \geq 2$.

- *Is $2^{n+1} = \Omega(2^n)$?* Go back to the definition. $f(n) = \Omega(g(n))$ iff there exists a constant $c > 0$ such that for all sufficiently large n $f(n) \geq c \cdot g(n)$. This would be satisfied for any $0 < c \leq 2$. Together the Big Oh and Ω bounds imply $2^{n+1} = \Theta(2^n)$

Some questions ?



Problem: Is $(x+y)^2 = O(x^2+y^2)$.

Solution: Working with the Big Oh means going back to the definition at the slightest sign of confusion. By definition, this expression is valid iff we can find some c such that $(x+y)^2 \leq c(x^2+y^2)$.

My first move would be to expand the left side of the equation, i.e. $(x+y)^2 = x^2+2xy+y^2$. If the middle $2xy$ term wasn't there, the inequality would clearly hold for any $c > 1$. But it is there, so we need to relate the $2xy$ to x^2+y^2. What if $x \leq y$? Then $2xy \leq 2y^2 \leq 2(x^2+y^2)$. What if $x \geq y$? Then $2xy \leq 2x^2 \leq 2(x^2+y^2)$. Either way, we now can bound this middle term by two times the right-side function. This means that $(x+y)^2 \leq 3(x^2+y^2)$, and so the result holds. $\blacksquare$