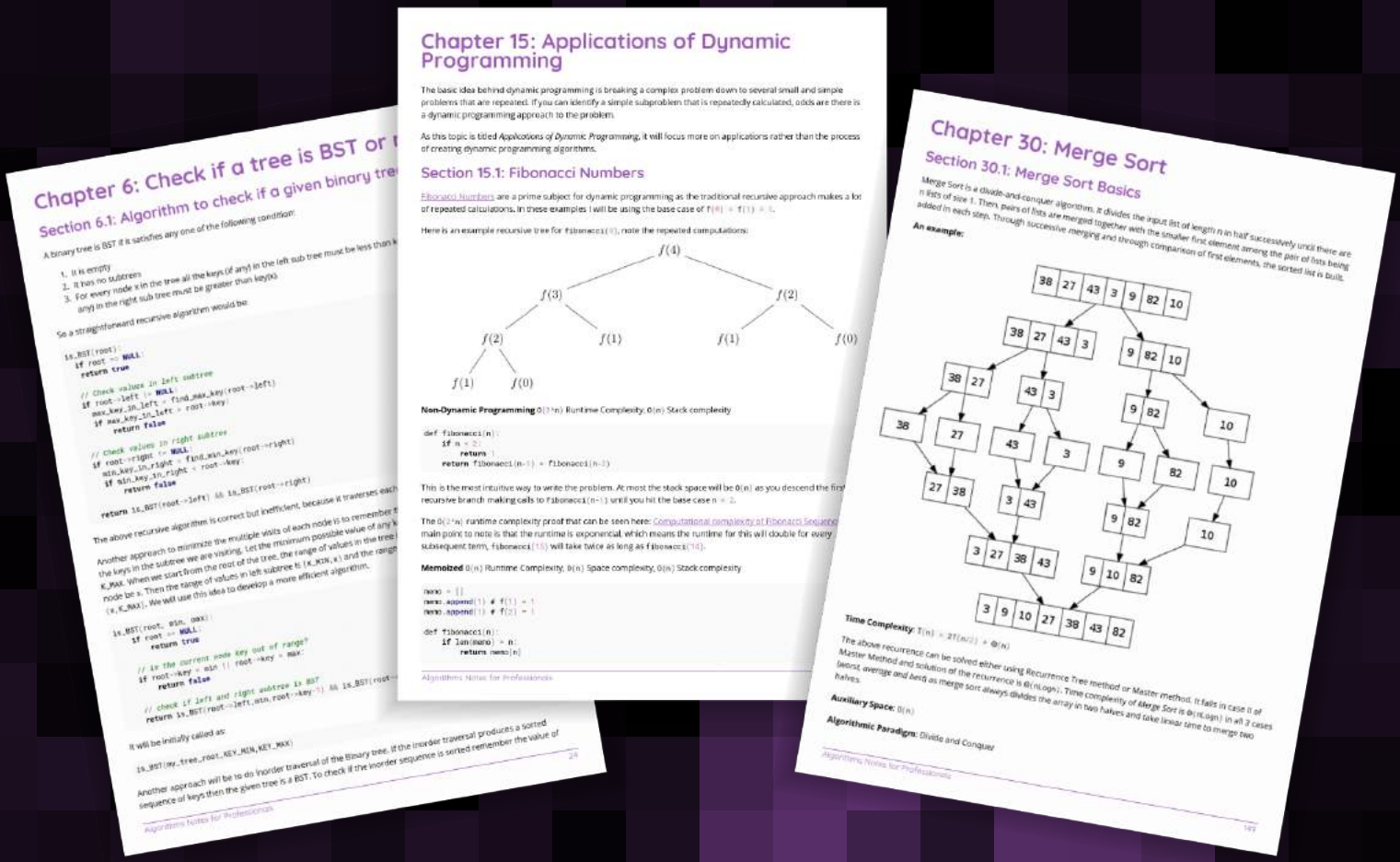
Algorithms

Notes for Professionals



200+ pages

of professional hints and tricks

Contents

About

Please feel free to share this PDF with anyone for free,
latest version of this book can be downloaded from:
https://goalkicker.com/AlgorithmsBook

Please send feedback and corrections to web@petercv.com

Chapter 1: Getting started with algorithms

Section 1.1: A sample algorithmic problem

An algorithmic problem is specified by describing the complete set of *instances* it must work on and of its output after running on one of these instances. This distinction, between a problem and an instance of a problem, is fundamental. The algorithmic *problem* known as *sorting* is defined as follows: [Skiena:2008:ADM:1410219]

- Problem: Sorting
- Input: A sequence of *n* keys, `a_1, a_2, ..., a_n`.
- Output: The reordering of the input sequence such that `a'_1 <= a'_2 <= ... <= a'_{n-1} <= a'_n`

An *instance* of sorting might be an array of strings, such as `{ Haskell, Emacs }` or a sequence of numbers such as `{ 154, 245, 1337 }`.

Section 1.2: Getting Started with Simple Fizz Buzz Algorithm in Swift

For those of you that are new to programming in Swift and those of you coming from different programming bases, such as Python or Java, this article should be quite helpful. In this post, we will discuss a simple solution for implementing swift algorithms.

Fizz Buzz

You may have seen Fizz Buzz written as Fizz Buzz, FizzBuzz, or Fizz-Buzz; they're all referring to the same thing. That "thing" is the main topic of discussion today. First, what is FizzBuzz?

This is a common question that comes up in job interviews.

Imagine a series of a number from 1 to 10.

```
1 2 3 4 5 6 7 8 9 10
```

Fizz and Buzz refer to any number that's a multiple of 3 and 5 respectively. In other words, if a number is divisible by 3, it is substituted with fizz; if a number is divisible by 5, it is substituted with buzz. If a number is simultaneously a multiple of 3 AND 5, the number is replaced with "fizz buzz." In essence, it emulates the famous children game "fizz buzz".

To work on this problem, open up Xcode to create a new playground and initialize an array like below:

```
// for example
let number  = [1,2,3,4,5]
// here 3 is fizz and 5 is buzz
```

To find all the fizz and buzz, we must iterate through the array and check which numbers are fizz and which are buzz. To do this, create a for loop to iterate through the array we have initialised:

```
for num in number {
  // Body and calculation goes here
}
```

After this, we can simply use the "if else" condition and module operator in swift ie - % to locate the fizz and buzz

```
for num in number {
  if num % 3 == 0 {
    print("\(num) fizz")
  } else {
    print(num)
  }
}
```

Great! You can go to the debug console in Xcode playground to see the output. You will find that the "fizzes" have been sorted out in your array.

For the Buzz part, we will use the same technique. Let's give it a try before scrolling through the article — you can check your results against this article once you've finished doing this.

```
for num in number {
  if num % 3 == 0 {
    print("\(num) fizz")
  } else if num % 5 == 0 {
    print("\(num) buzz")
  } else {
    print(num)
  }
}
```

Check the output!

It's rather straight forward — you divided the number by 3, fizz and divided the number by 5, buzz. Now, increase the numbers in the array

```
let number = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
```

We increased the range of numbers from 1-10 to 1-15 in order to demonstrate the concept of a "fizz buzz." Since 15 is a multiple of both 3 and 5, the number should be replaced with "fizz buzz." Try for yourself and check the answer!

Here is the solution:

```
for num in number {
  if num % 3 == 0 && num % 5 == 0 {
    print("\(num) fizz buzz")
  } else if num % 3 == 0 {
    print("\(num) fizz")
  } else if num % 5 == 0 {
    print("\(num) buzz")
  } else {
    print(num)
  }
}
```

Wait...it's not over though! The whole purpose of the algorithm is to customize the runtime correctly. Imagine if the range increases from 1-15 to 1-100. The compiler will check each number to determine whether it is divisible by 3 or 5. It would then run through the numbers again to check if the numbers are divisible by 3 and 5. The code would essentially have to run through each number in the array twice — it would have to runs the numbers by 3 first and then run it by 5. To speed up the process, we can simply tell our code to divide the numbers by 15 directly.

Here is the final code:

```
for num in number {
```

```
  if num % 15 == 0 {
    print("\(num) fizz buzz")
  } else if num % 3 == 0 {
    print("\(num) fizz")
  } else if num % 5 == 0 {
    print("\(num) buzz")
  } else {
    print(num)
  }
}
```

As Simple as that, you can use any language of your choice and get started

Enjoy Coding

Chapter 2: Algorithm Complexity

Section 2.1: Big-Theta notation

Unlike Big-O notation, which represents only upper bound of the running time for some algorithm, Big-Theta is a tight bound; both upper and lower bound. Tight bound is more precise, but also more difficult to compute.

The Big-Theta notation is symmetric: `f(x) = Θ(g(x)) <=> g(x) = Θ(f(x))`

An intuitive way to grasp it is that `f(x) = Θ(g(x))` means that the graphs of f(x) and g(x) grow in the same rate, or that the graphs 'behave' similarly for big enough values of x.

The full mathematical expression of the Big-Theta notation is as follows:
Θ(f(x)) = {g: N0 -> R and c1, c2, n0 > 0, where c1 < abs(g(n) / f(n)), for every n > n0 and abs is the absolute value }

An example

If the algorithm for the input `n` takes `42n^2 + 25n + 4` operations to finish, we say that is `O(n^2)`, but is also `O(n^3)` and `O(n^100)`. However, it is `Θ(n^2)` and it is not `Θ(n^3)`, `Θ(n^4)` etc. Algorithm that is `Θ(f(n))` is also `O(f(n))`, but not vice versa!

Formal mathematical definition

`Θ(g(x))` is a set of functions.

```
Θ(g(x)) = {f(x) such that there exist positive constants c1, c2, N such that 0 <= c1*g(x) <= f(x) <= c2*g(x) for all x > N}
```

Because `Θ(g(x))` is a set, we could write `f(x) ∈ Θ(g(x))` to indicate that `f(x)` is a member of `Θ(g(x))`. Instead, we will usually write `f(x) = Θ(g(x))` to express the same notion - that's the common way.

Whenever `Θ(g(x))` appears in a formula, we interpret it as standing for some anonymous function that we do not care to name. For example the equation `T(n) = T(n/2) + Θ(n)`, means `T(n) = T(n/2) + f(n)` where `f(n)` is a function in the set `Θ(n)`.

Let `f` and `g` be two functions defined on some subset of the real numbers. We write `f(x) = Θ(g(x))` as `x->infinity` if and only if there are positive constants `K` and `L` and a real number `x0` such that holds:

`K|g(x)| <= f(x) <= L|g(x)|` for all `x >= x0`.

The definition is equal to:

`f(x) = O(g(x)) and f(x) = Ω(g(x))`

A method that uses limits

if `limit(x->infinity) f(x)/g(x) = c ∈ (0,∞)` i.e. the limit exists and it's positive, then `f(x) = Θ(g(x))`

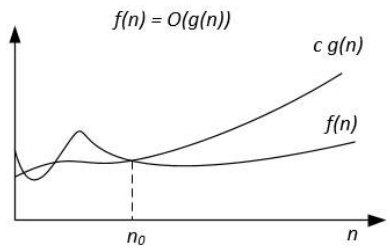
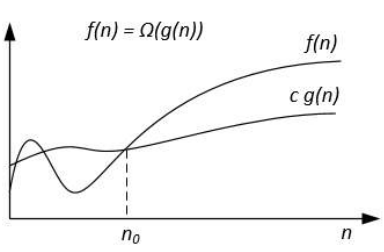
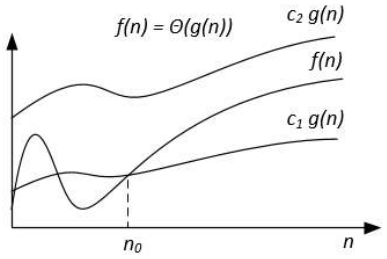
Common Complexity Classes

Name	Notation	n = 10	n = 100
Constant	Θ(1)	1	1
Logarithmic	Θ(log(n))	3	7
Linear	Θ(n)	10	100

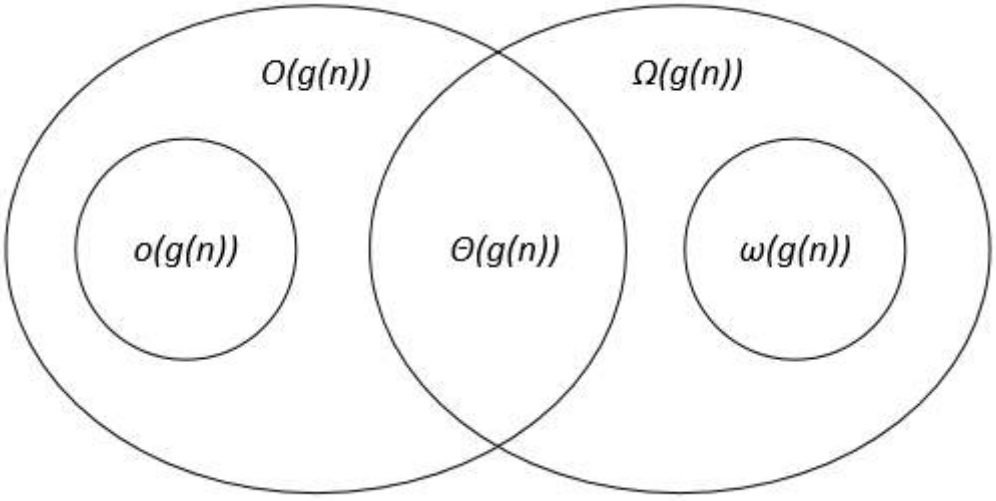
Linearithmic	Θ(n*log(n))	30	700
Quadratic	Θ(n^2)	100	10 000
Exponential	Θ(2^n)	1 024	1.267650e+ 30
Factorial	Θ(n!)	3 628 800	9.332622e+157

Section 2.2: Comparison of the asymptotic notations

Let `f(n)` and `g(n)` be two functions defined on the set of the positive real numbers, `c`, `c1`, `c2`, `n0` are positive real constants.

Notation	f(n) = O(g(n))	f(n) = Ω(g(n))	f(n) = Θ(g(n))	f(n) = o(g(n))	f(n) = ω(g(n))
Formal definition	`∃ c > 0, ∃ n0 > 0 : ∀ n ≥ n0, 0 ≤ f(n) ≤ c g(n)`	`∃ c > 0, ∃ n0 > 0 : ∀ n ≥ n0, 0 ≤ c g(n) ≤ f(n)`	`∃ c1, c2 > 0, ∃ n0 > 0 : ∀ n ≥ n0, 0 ≤ c1 g(n) ≤ f(n) ≤ c2 g(n)`	`∀ c > 0, ∃ n0 > 0 0 : ∀ n ≥ n0, 0 ≤ f(n) < c g(n)`	`∀ c > 0, ∃ n0 > 0 : ∀ n ≥ n0, 0 ≤ c g(n) < f(n)`
Analogy between the asymptotic comparison of `f`, `g` and real numbers `a`, `b`	`a ≤ b`	`a ≥ b`	`a = b`	`a < b`	`a > b`
Example	`7n + 10 = O(n^2 + n - 9)`	`n^3 - 34 = Ω(10n^2 - 7n + 1)`	`1/2 n^2 - 7n = Θ(n^2)`	`5n^2 = o(n^3)`	`7n^2 = ω(n)`
Graphic interpretation					

The asymptotic notations can be represented on a Venn diagram as follows:



Links

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Introduction to Algorithms.

Section 2.3: Big-Omega Notation

Ω-notation is used for asymptotic lower bound.

Formal definition

Let `f(n)` and `g(n)` be two functions defined on the set of the positive real numbers. We write `f(n) = Ω(g(n))` if there are positive constants `c` and `n0` such that:

`0 ≤ c g(n) ≤ f(n) for all n ≥ n0`.

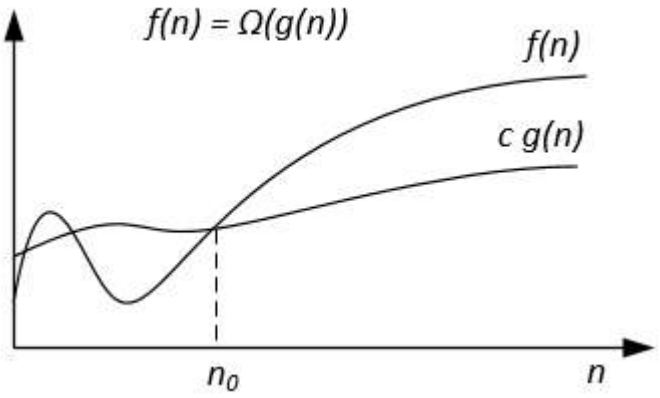
Notes

`f(n) = Ω(g(n))` means that `f(n)` grows asymptotically no slower than `g(n)`. Also we can say about `Ω(g(n))` when algorithm analysis is not enough for statement about `Θ(g(n))` or / and `O(g(n))`.

From the definitions of notations follows the theorem:

For two any functions `f(n)` and `g(n)` we have `f(n) = Θ(g(n))` if and only if `f(n) = O(g(n)) and f(n) = Ω(g(n))`.

Graphically Ω-notation may be represented as follows:



For example lets we have `f(n) = 3n^2 + 5n - 4`. Then `f(n) = Ω(n^2)`. It is also correct `f(n) = Ω(n)`, or even `f(n) = Ω(1)`.

Another example to solve perfect matching algorithm : If the number of vertices is odd then output "No Perfect Matching" otherwise try all possible matchings.

We would like to say the algorithm requires exponential time but in fact you cannot prove a `Ω(n^2)` lower bound using the usual definition of Ω since the algorithm runs in linear time for n odd. We should instead define `f(n)=Ω(g(n))` by saying for some constant `c>0`, `f(n)≥ c g(n)` for infinitely many `n`. This gives a nice correspondence between upper and lower bounds: `f(n)=Ω(g(n))` iff `f(n) != o(g(n))`.

References

Formal definition and theorem are taken from the book "Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Introduction to Algorithms".

Chapter 3: Big-O Notation

Definition

The Big-O notation is at its heart a mathematical notation, used to compare the rate of convergence of functions. Let `n -> f(n)` and `n -> g(n)` be functions defined over the natural numbers. Then we say that `f = O(g)` if and only if `f(n)/g(n)` is bounded when n approaches infinity. In other words, `f = O(g)` if and only if there exists a constant A, such that for all n, `f(n)/g(n) <= A`.

Actually the scope of the Big-O notation is a bit wider in mathematics but for simplicity I have narrowed it to what is used in algorithm complexity analysis : functions defined on the naturals, that have non-zero values, and the case of n growing to infinity.

What does it mean ?

Let's take the case of `f(n) = 100n^2 + 10n + 1` and `g(n) = n^2`. It is quite clear that both of these functions tend to infinity as n tends to infinity. But sometimes knowing the limit is not enough, and we also want to know the *speed* at which the functions approach their limit. Notions like Big-O help compare and classify functions by their speed of convergence.

Let's find out if `f = O(g)` by applying the definition. We have `f(n)/g(n) = 100 + 10/n + 1/n^2`. Since `10/n` is 10 when n is 1 and is decreasing, and since `1/n^2` is 1 when n is 1 and is also decreasing, we have `f(n)/g(n) <= 100 + 10 + 1 = 111`. The definition is satisfied because we have found a bound of `f(n)/g(n)` (111) and so `f = O(g)` (we say that f is a Big-O of `n^2`).

This means that f tends to infinity at approximately the same speed as g. Now this may seem like a strange thing to say, because what we have found is that f is at most 111 times bigger than g, or in other words when g grows by 1, f grows by at most 111. It may seem that growing 111 times faster is not "approximately the same speed". And indeed the Big-O notation is not a very precise way to classify function convergence speed, which is why in mathematics we use the equivalence relationship when we want a precise estimation of speed. But for the purposes of separating algorithms in large speed classes, Big-O is enough. We don't need to separate functions that grow a fixed number of times faster than each other, but only functions that grow *infinitely* faster than each other. For instance if we take `h(n) = n^2*log(n)`, we see that `h(n)/g(n) = log(n)` which tends to infinity with n so h is *not* O(n^2), because h grows *infinitely* faster than n^2.

Now I need to make a side note : you might have noticed that if `f = O(g)` and `g = O(h)`, then `f = O(h)`. For instance in our case, we have `f = O(n^3)`, and `f = O(n^4)`... In algorithm complexity analysis, we frequently say `f = O(g)` to mean that `f = O(g)` *and* `g = O(f)`, which can be understood as "g is the smallest Big-O for f". In mathematics we say that such functions are Big-Thetas of each other.

How is it used ?

When comparing algorithm performance, we are interested in the number of operations that an algorithm performs. This is called *time complexity*. In this model, we consider that each basic operation (addition, multiplication, comparison, assignment, etc.) takes a fixed amount of time, and we count the number of such operations. We can usually express this number as a function of the size of the input, which we call n. And sadly, this number usually grows to infinity with n (if it doesn't, we say that the algorithm is O(1)). We separate our algorithms in big speed classes defined by Big-O : when we speak about a "O(n^2) algorithm", we mean that the number of operations it performs, expressed as a function of n, is a O(n^2). This says that our algorithm is approximately as fast as an algorithm that would do a number of operations equal to the square of the size of its input, *or faster*. The "or faster" part is there because I used Big-O instead of Big-Theta, but usually people will say Big-O to mean Big-Theta.

When counting operations, we usually consider the worst case: for instance if we have a loop that can run at most n times and that contains 5 operations, the number of operations we count is 5n. It is also possible to consider the average case complexity.

Quick note : a fast algorithm is one that performs few operations, so if the number of operations grows to infinity *faster*, then the algorithm is *slower*: O(n) is better than O(n^2).

We are also sometimes interested in the *space complexity* of our algorithm. For this we consider the number of bytes in memory occupied by the algorithm as a function of the size of the input, and use Big-O the same way.

Section 3.1: A Simple Loop

The following function finds the maximal element in an array:

```
int find_max(const int *array, size_t len) {
    int max = INT_MIN;
    for (size_t i = 0; i < len; i++) {
        if (max < array[i]) {
            max = array[i];
        }
    }
    return max;
}
```

The input size is the size of the array, which I called `len` in the code.

Let's count the operations.

```
int max = INT_MIN;
size_t i = 0;
```

These two assignments are done only once, so that's 2 operations. The operations that are looped are:

```
if (max < array[i])
i++;
max = array[i]
```

Since there are 3 operations in the loop, and the loop is done n times, we add `3n` to our already existing 2 operations to get `3n + 2`. So our function takes `3n + 2` operations to find the max (its complexity is `3n + 2`). This is a polynomial where the fastest growing term is a factor of n, so it is O(n).

You probably have noticed that "operation" is not very well defined. For instance I said that `if (max < array[i])` was one operation, but depending on the architecture this statement can compile to for instance three instructions : one memory read, one comparison and one branch. I have also considered all operations as the same, even though for instance the memory operations will be slower than the others, and their performance will vary wildly due for instance to cache effects. I also have completely ignored the return statement, the fact that a frame will be created for the function, etc. In the end it doesn't matter to complexity analysis, because whatever way I choose to count operations, it will only change the coefficient of the n factor and the constant, so the result will still be O(n). Complexity shows how the algorithm scales with the size of the input, but it isn't the only aspect of performance!

Section 3.2: A Nested Loop

The following function checks if an array has any duplicates by taking each element, then iterating over the whole array to see if the element is there

```
_Bool contains_duplicates(const int *array, size_t len) {
    for (int i = 0; i < len - 1; i++) {
        for (int j = 0; j < len; j++) {
            if (i != j && array[i] == array[j]) {
                return 1;
            }
        }
    }
    return 0;
}
```

The inner loop performs at each iteration a number of operations that is constant with `n`. The outer loop also does a few constant operations, and runs the inner loop `n` times. The outer loop itself is run `n` times. So the operations inside the inner loop are run `n^2` times, the operations in the outer loop are run `n` times, and the assignment to `i` is done one time. Thus, the complexity will be something like `an^2 + bn + c`, and since the highest term is `n^2`, the O notation is `O(n^2)`.

As you may have noticed, we can improve the algorithm by avoiding doing the same comparisons multiple times. We can start from `i + 1` in the inner loop, because all elements before it will already have been checked against all array elements, including the one at index `i + 1`. This allows us to drop the `i == j` check.

```
_Bool faster_contains_duplicates(const int *array, size_t len) {
    for (int i = 0; i < len - 1; i++) {
        for (int j = i + 1; j < len; j++) {
            if (array[i] == array[j]) {
                return 1;
            }
        }
    }
    return 0;
}
```

Obviously, this second version does less operations and so is more efficient. How does that translate to Big-O notation? Well, now the inner loop body is run `1 + 2 + ... + n - 1 = n(n-1)/2` times. This is *still* a polynomial of the second degree, and so is still only `O(n^2)`. We have clearly lowered the complexity, since we roughly divided by 2 the number of operations that we are doing, but we are still in the same complexity *class* as defined by Big-O. In order to lower the complexity to a lower class we would need to divide the number of operations by something that *tends to infinity* with `n`.

Section 3.3: O(log n) types of Algorithms

Let's say we have a problem of size n. Now for each step of our algorithm(which we need write), our original problem becomes half of its previous size(n/2).

So at each step, our problem becomes half.

Step	Problem
1	n/2
2	n/4
3	n/8
4	n/16

When the problem space is reduced(i.e solved completely), it cannot be reduced any further(n becomes equal to 1) after exiting check condition.

1. Let's say at kth step or number of operations:

   problem-size = 1

2. But we know at kth step, our problem-size should be:

   problem-size = n/2k

3. From 1 and 2:

   n/2k = 1 or

   n = 2k

4. Take log on both sides

   loge n = k loge2

   or

   k = loge n / loge 2

5. Using formula **logx m / logx n = logn m**

   k = log2 n

   or simply k = log n

> Now we know that our algorithm can run maximum up to log n, hence time complexity comes as O(log n)

A very simple example in code to support above text is :

```
for(int i=1; i<=n; i=i*2)
{
    // perform some operation
}
```

So now if some one asks you if n is 256 how many steps that loop(or any other algorithm that cuts down it's problem size into half) will run you can very easily calculate.

k = log2 256

k = log2 2 8 (=> logaa = 1)

k = 8

Another very good example for similar case is **Binary Search Algorithm**.

```
int bSearch(int arr[],int size,int item){
 int low=0;
 int high=size-1;

 while(low<=high){
     mid=low+(high-low)/2;
     if(arr[mid]==item)
         return mid;
     else if(arr[mid]<item)
         low=mid+1;
     else  high=mid-1;
     }
  return –1;// Unsuccessful result
}
```

Section 3.4: An O(log n) example

Introduction

Consider the following problem:

`L` is a sorted list containing `n` signed integers (`n` being big enough), for example `[-5, -2, -1, 0, 1, 2, 4]` (here, `n` has a value of 7). If `L` is known to contain the integer 0, how can you find the index of 0 ?

Naïve approach

The first thing that comes to mind is to just read every index until 0 is found. In the worst case, the number of operations is `n`, so the complexity is O(n).

This works fine for small values of `n`, but is there a more efficient way ?

Dichotomy

Consider the following algorithm (Python3):

```
a = 0
b = n-1
while True:
  h = (a+b)//2 ## // is the integer division, so h is an integer
  if L[h] == 0:
    return h
  elif L[h] > 0:
    b = h
  elif L[h] < 0:
    a = h
```

`a` and `b` are the indexes between which 0 is to be found. Each time we enter the loop, we use an index between `a` and `b` and use it to narrow the area to be searched.

In the worst case, we have to wait until `a` and `b` are equal. But how many operations does that take? Not n, because each time we enter the loop, we divide the distance between `a` and `b` by about two. Rather, the complexity is O(log n).

Explanation

Note: When we write "log", we mean the binary logarithm, or log base 2 (which we will write "log_2"). As O(log_2 n) = O(log n) (you can do the math) we will use "log" instead of "log_2".

Let's call x the number of operations: we know that 1 = n / (2^x).

So 2^x = n, then x = log n

Conclusion

When faced with successive divisions (be it by two or by any number), remember that the complexity is logarithmic.

Chapter 4: Trees

Section 4.1: Typical anary tree representation

Typically we represent an anary tree (one with potentially unlimited children per node) as a binary tree, (one with exactly two children per node). The "next" child is regarded as a sibling. Note that if a tree is binary, this representation creates extra nodes.

We then iterate over the siblings and recurse down the children. As most trees are relatively shallow - lots of children but only a few levels of hierarchy, this gives rise to efficient code. Note human genealogies are an exception (lots of levels of ancestors, only a few children per level).

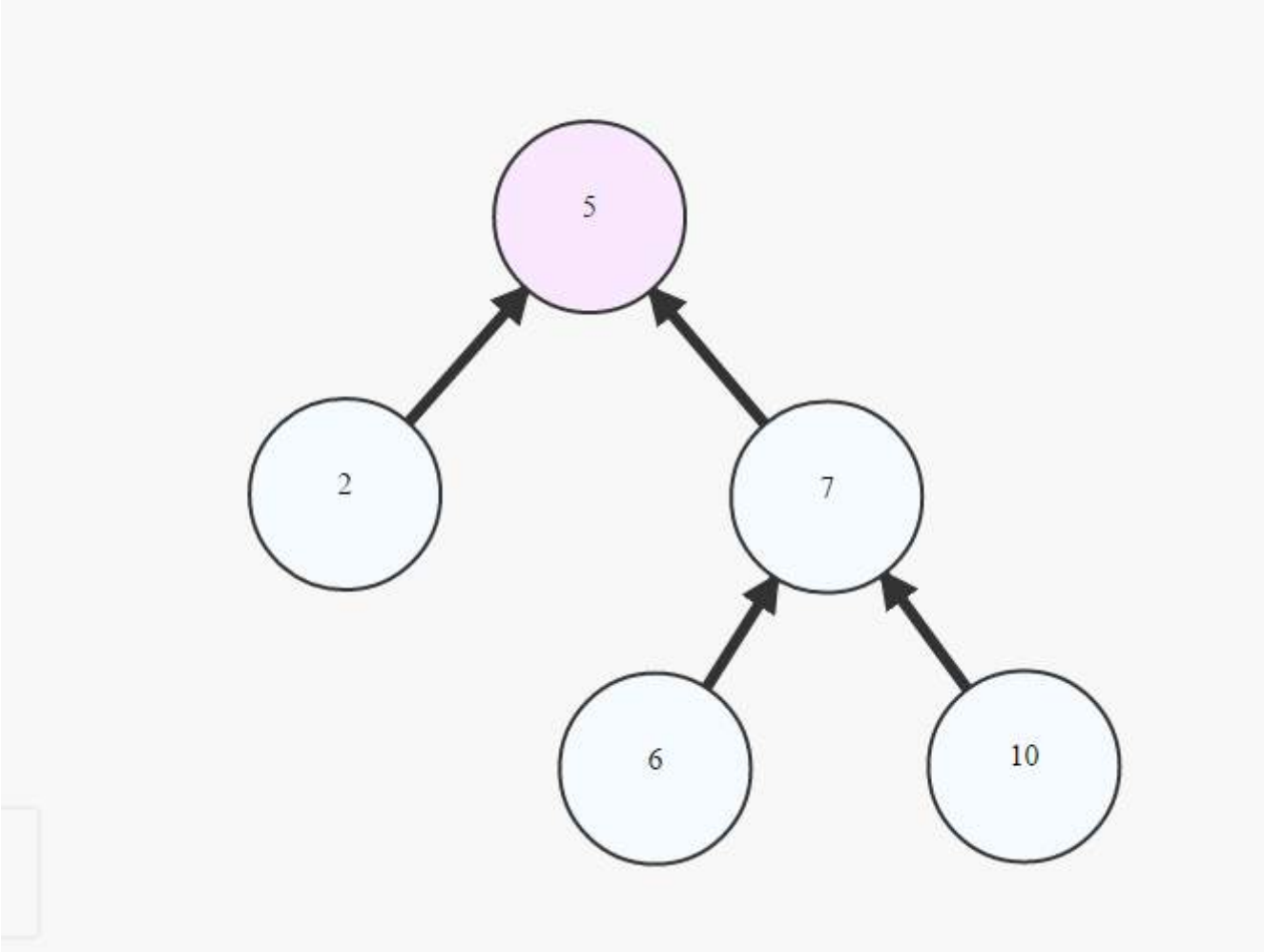
If necessary back pointers can be kept to allow the tree to be ascended. These are more difficult to maintain.

Note that it is typical to have one function to call on the root and a recursive function with extra parameters, in this case tree depth.

```
struct node
{
   struct node *next;
   struct node *child;
   std::string data;
}

void printtree_r(struct node *node, int depth)
{
   int i;

   while(node)
   {
       if(node->child)
       {
          for(i=0;i<depth*3;i++)
              printf(" ");
          printf("{\n"):
          printtree_r(node->child, depth +1);
          for(i=0;i<depth*3;i++)
              printf(" ");
          printf("{\n"):

          for(i=0;i<depth*3;i++)
             printf(" ");
           printf("%s\n", node->data.c_str());

           node = node->next;
        }
    }
}

void printtree(node *root)
{
   printree_r(root, 0);
}
```

Section 4.2: Introduction

Trees are a sub-type of the more general node-edge graph data structure.



To be a tree, a graph must satisfy two requirements:

- **It is acyclic.** It contains no cycles (or "loops").
- **It is connected.** For any given node in the graph, every node is reachable. All nodes are reachable through one path in the graph.

The tree data structure is quite common within computer science. Trees are used to model many different algorithmic data structures, such as ordinary binary trees, red-black trees, B-trees, AB-trees, 23-trees, Heap, and tries.

it is common to refer to a Tree as a `Rooted Tree` by:

```
choosing 1 cell to be called `Root`
painting the `Root` at the top
creating lower layer for each cell in the graph depending on their distance from the root -the
bigger the distance, the lower the cells (example above)
```

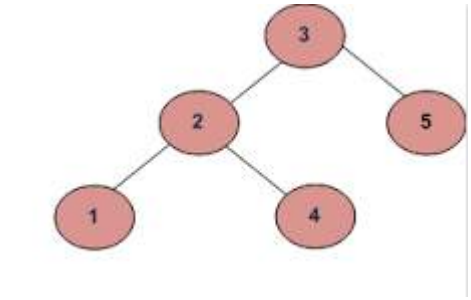
common symbol for trees: `T`

Section 4.3: To check if two Binary trees are same or not

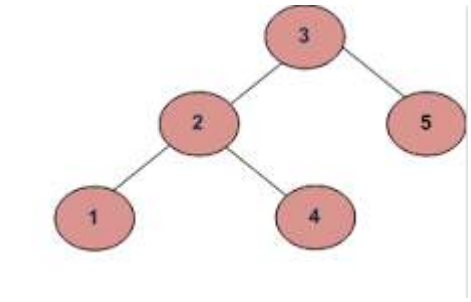
1. For example if the inputs are:

Example:1

a)



b)

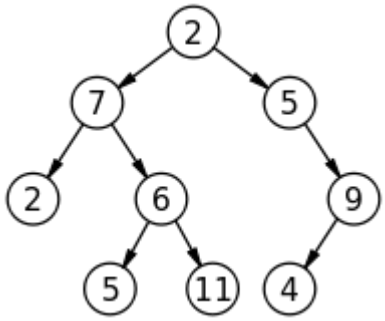


Output should be true.

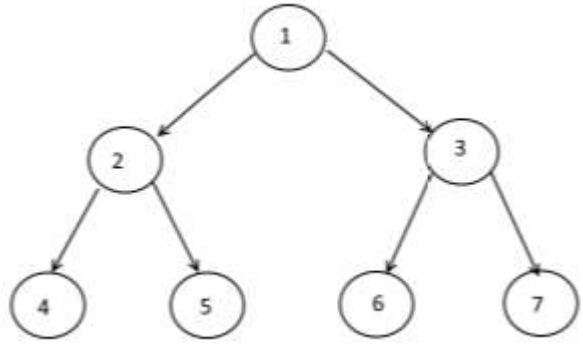
Example:2

If the inputs are:

a)



b)



Output should be false.

Pseudo code for the same:

```
boolean sameTree(node root1, node root2){
```

```
if(root1 == NULL && root2 == NULL)
return true;

if(root1 == NULL || root2 == NULL)
return false;

if(root1->data == root2->data
     && sameTree(root1->left,root2->left)
        && sameTree(root1->right, root2->right))
return true;

}
```

Chapter 5: Binary Search Trees

Binary tree is a tree that each node in it has maximum of two children. Binary search tree (BST) is a binary tree which its elements positioned in special order. In each BST all values(i.e key) in left sub tree are less than values in right sub tree.

Section 5.1: Binary Search Tree - Insertion (Python)

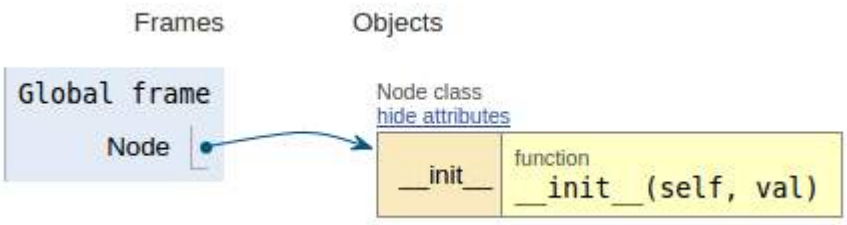
This is a simple implementation of Binary Search Tree Insertion using Python.

An example is shown below:

www.penjee.com

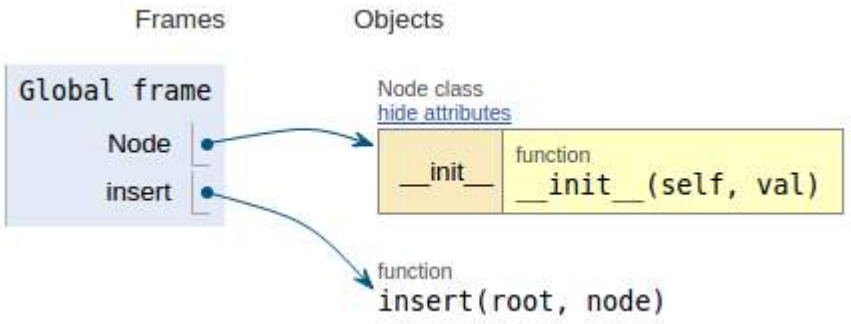
Following the code snippet each image shows the execution visualization which makes it easier to visualize how this code works.

```
class Node:
    def __init__(self, val):
        self.l_child = None
        self.r_child = None
        self.data = val
```

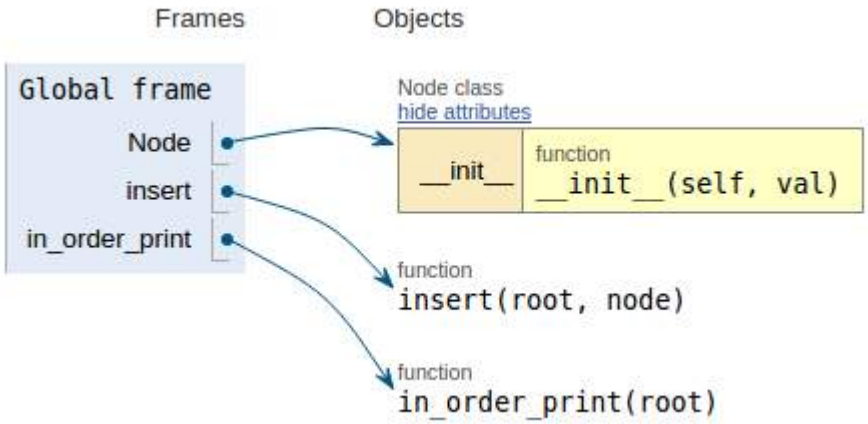



```
def insert(root, node):
    if root is None:
        root = node
    else:
        if root.data > node.data:
            if root.l_child is None:
                root.l_child = node
            else:
                insert(root.l_child, node)
        else:
            if root.r_child is None:
```

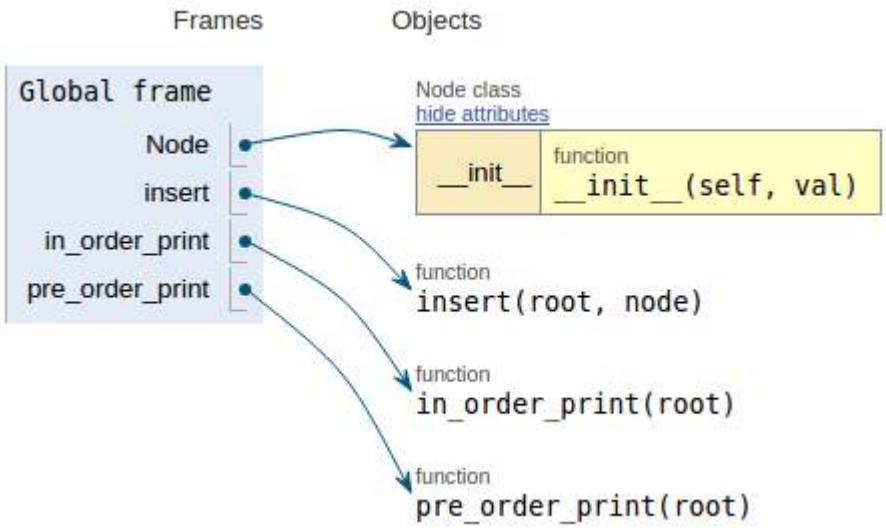
```
                root.r_child = node
            else:
                insert(root.r_child, node)
```




```
def in_order_print(root):
    if not root:
        return
    in_order_print(root.l_child)
    print root.data
    in_order_print(root.r_child)
```



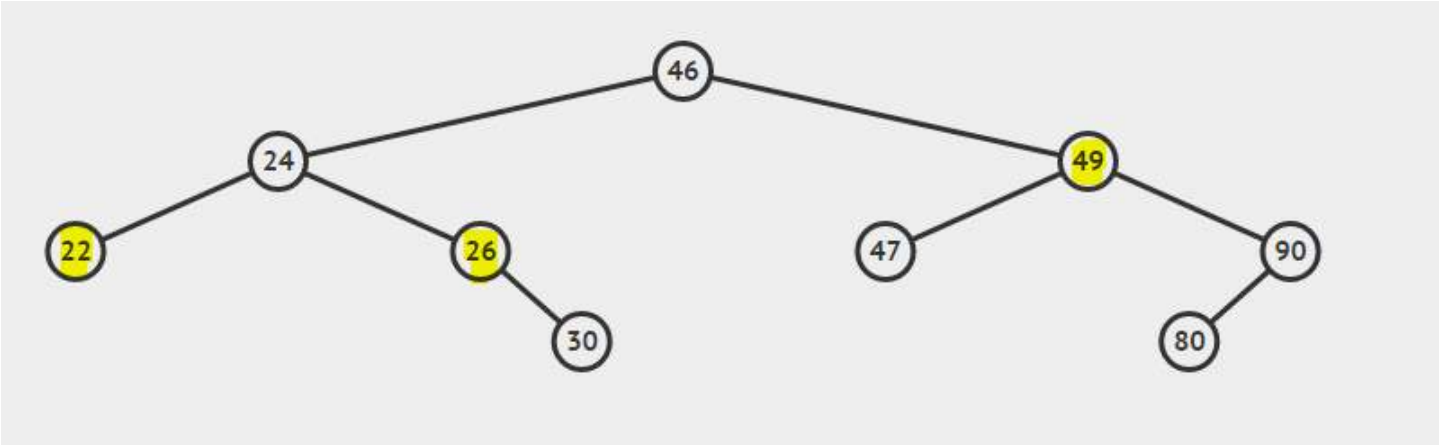

```
def pre_order_print(root):
    if not root:
        return
    print root.data
    pre_order_print(root.l_child)
    pre_order_print(root.r_child)
```



Section 5.2: Binary Search Tree - Deletion(C++)

Before starting with deletion I just want to put some lights on what is a Binary search tree(BST), Each node in a BST can have maximum of two nodes(left and right child).The left sub-tree of a node has a key less than or equal to its parent node's key. The right sub-tree of a node has a key greater than to its parent node's key.

Deleting a node in a tree while maintaining its **Binary search tree property.**



There are three cases to be considered while deleting a node.

- Case 1: Node to be deleted is the leaf node.(Node with value 22).
- Case 2: Node to be deleted has one child.(Node with value 26).
- Case 3: Node to be deleted has both children.(Node with value 49).

Explanation of cases:

1. When the node to be deleted is a leaf node then simply delete the node and pass `nullptr` to its parent node.
2. When a node to be deleted is having only one child then copy the child value to the node value and delete the child **(Converted to case 1)**
3. When a node to be delete is having two childs then the minimum from its right sub tree can be copied to the node and then the minimum value can be deleted from the node's right subtree **(Converted to Case 2)**

Note: The minimum in the right sub tree can have a maximum of one child and that too right child if it's having the left child that means it's not the minimum value or it's not following BST property.