Sada Kurapati

Lost in ever growing technology…

Category Algorithm

MAY 1 2015

Recursion

This is a simple concept and yet scares many, as a simple mistake can bring down even a supercomputer by making infinite recursive calls and run out of resources. Also it is hard to think recursively as we can't easily relate to any of the real world elements. It is not too complex topic to understand like 5th dimension (http://www.ted.com/talks/brian_greene_on_string_theory?language=en) and by the way, the dimensions topic is pretty interesting too.

Why should I learn recursion?

It is one of the key concepts and following are some of the algorithms and problem spaces which we can easily solve using recursion –

- Divide and Concur – Ex: Merge Sort (https://sadakurapati.wordpress.com/2013/08/20/algorithms-merge-sorting/)
- Backtracking (https://sadakurapati.wordpress.com/2013/12/10/n-queens-backtracking-algorithm/)
- Dynamic Programming (a few)
- Graphs & Tree problems (https://sadakurapati.wordpress.com/2014/10/15/graphs-trees-shortest-path-bfs-dijkstra-bellman-ford/)
- Linked List problems
- Permutations & Combinations
- Solving games – like Maze, Sudoku etc

Hoping now that you are motivated to understand recursion lets start by discussing an example.

What is recursion?

Let us assume you bought a box of chocolates to a classroom and want to give it to everyone in the class. One intuitive and simple way is to go to each person and give a chocolate. This will surely make us work hard and more the members in classroom the more work we do. Is there a better way?
What about, you **take a chocolate** for yourself and give the box to the person next to you and say, can you **take one and pass on to the next person**? He simply can take one chocolate for himself and then do the **same thing** – giving it to next person.
Above problem can be classified as recursion – distributing chocolates. Taking a chocolate is doing some work and then simply **recurse** for remaining persons in the classroom. One of the important thing is, this will stop when everyone gets a chocolate. By the way, this is called the **base case**.
Following are the two important things to remember when we write recursive functions,

- It should contain a **base case**. That means, we should have a small enough problem where we can solve it directly and simply stop recursing further. In above example, last person simply takes the chocolate for himself and stops as everybody already got one.
- The problem **should become smaller and smaller** for each recursive call. In the above example, first we need to give it to all and next it is all-1, all-2, all-3 and so on.

Let us look at the simple programming example – print "Hello World!" 10 times.
The intuitive way is, loop through 10 times and print the statement "Hello World!" but assume that we have a constraint of not to use loop construct. What about, we print "Hello World!" and then recursively say, can you print 9 more times, print and recurse for 8 more times, print and recurse 7 more times and so on till we reach printing only 1 time. This can be written as follows,

```java
public static void printHelloWorld(int times) {
    if (times == 0) {
        return; // base case
    } else {
        System.out.println("Hello World!");
        printHelloWorld(times - 1); // recursive call
    }
}
```

The above can also be written simply as,

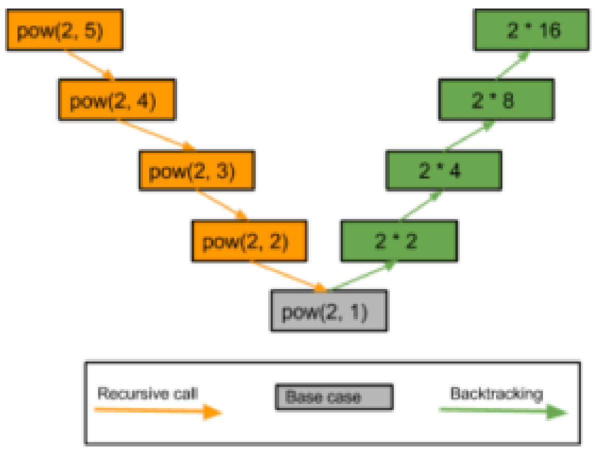
```java
// implicit base case
public static void printHelloWorld(int times) {
    if (times > 0) {
        System.out.println("Hello World!");
        printHelloWorld(times - 1); // recursive call
    }
}
```

The best way to get comfortable with this topic is to solve as many recursive problems as possible. Let us look into some of the problems below and see how can we solve them with recursion.

POW function

Given a base and exponent, how do we raise the wbase to the given exponent. Iteratively, we can solve this by simply using a loop construct and multiply base exponent times. Can we do this recursively? Let us take an example – base is 2 and exponent is 10. This can also be written as 2 * (2 raised to 9). So that means, if we know the value of base raised to (exponent – 1) times, then we can simply multiply it with base once and we get what we want. If you look at this approach carefully, raising base to (exponent – 1) times is same problem and so we can easily form this into recursive formula – **base raise to exponent = base * (base raised to exponent -1)**. That is pow(base, exponent) = base * pow (base, exponent -1). The base case occurs when exponent is 0 and in that case, we simply return 1. Following is the java code,

```java
public static double pow(double base, double power) {
    if (power <= 0) {
        return 1.0;
    } else {
        return base * pow(base, power - 1);
    }
}
```



(https://sadakurapati.files.wordpress.com/2015/05/recursion_pow.png)
Recursive call stack for POW function

A visual way to look at the above is for – raise 2 to 5,

Just FYI, if you carefully observe it, we can do little optimization and reduce the run time of the above from O(n) to O(log n). The recursive formula can also be written as,

- If exponent is even, then pow(base, exponent) = pow(base, exponent/2) * pow(base, exponent/2)
- if exponent is odd, then pow(base, exponent) = base * pow(base, exponent/2) * pow(base, exponent/2)

Optimized java code for POW function is,

```java
public static double power(double x, int n) {
    if (n == 0) {
        return 1;
    } else {
        double v = power(x, n / 2);
        if (n % 2 == 0) {
            return v * v;
        } else {
            return v * v * x;
        }
    }
}
```

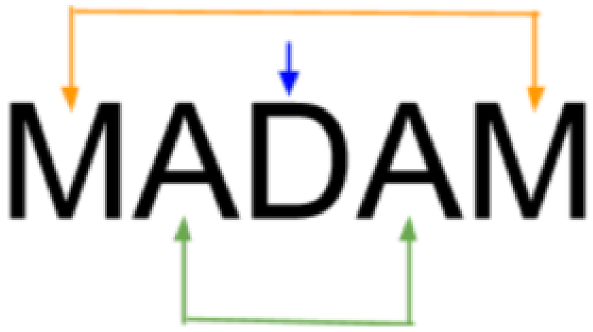
By the way, the above code does not work for negative exponents and you can take that as an exercise.
Hint – write a wrapper function to check if exponent is negative or positive and then I hope you know what to do.

Palindrome

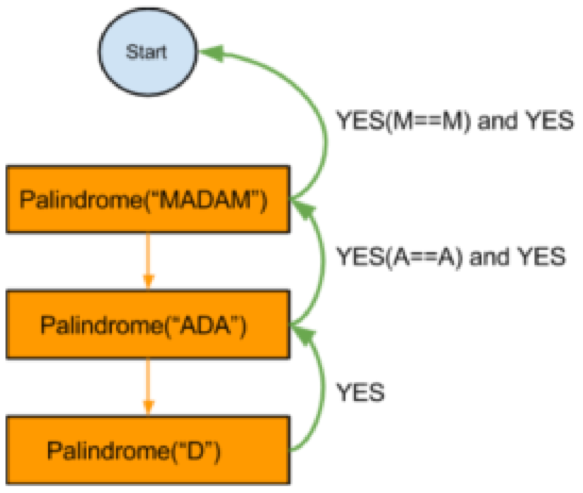
Given a string input, find if it is a palindrome or not. As per Wikipedia – "A palindrome (http://en.wikipedia.org/wiki/Palindrome) is a word, phrase, number, or other sequence of characters which reads the same backward or forward". One other way to think of this is, take first and last character and compare. If they are equal then remove them and check for inner string. If at any point the first and last character is not same, it is not a palindrome otherwise it is. The recursive formula is,

- palindrome(INPUT) = (first character == last character) and palindrome(INPUT after first and last characters removed)

The better way to understand is to visualize it.



(https://sadakurapati.files.wordpress.com/2015/05/recursion-madam-1.png)
Comparisons for "MADAM" word



(https://sadakurapati.files.wordpress.com/2015/05/recursion-madam-2.png)
Recursive call stack for "MADAM" palindrome method

Java code for implementing palindrome recursively is,

```java
public static boolean isPalindromeRecursively(String input) {
    if (input.length() <= 1) {
        // one character or zero characters string is palindrome.
        return true;
    } else {
        char firstChar = input.charAt(0);
        char lastChar = input.charAt(input.length() - 1);
        // both end chars are same and inner input is palindrome, then
        return (firstChar == lastChar) && isPalindromeRecursively(inpu
    }
}
```

Note,

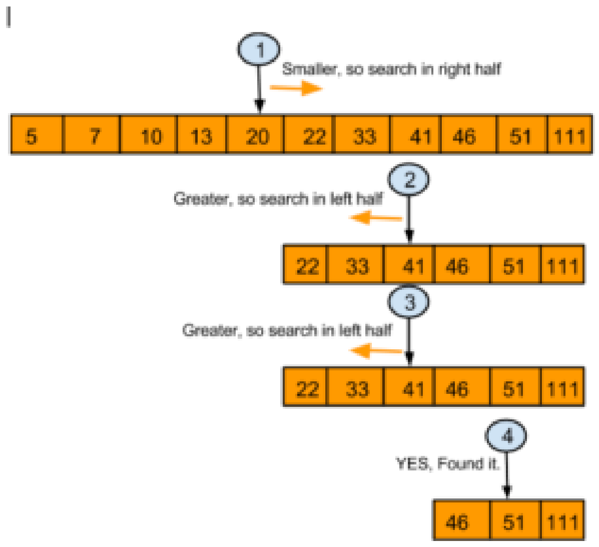
- The above code is not efficient as it creates many substrings and the better way to write it by converting the input to array and use start and end indexes.
- It might have few more constraints like ignoring special characters and spaces. For example, "A man, a plan, a canal — Panama!" might be considered as a palindrome but above code returns false.

A good exercise for you would be to take the above constraints and write/modify the code.

Binary search

This is one of the best algorithms to know and can be written very easily using recursion. The problem is, we are given a list of sorted integers (for the sake of simplicity) and a number which we need to search for. The brute force algorithm is to simply scan the elements from start to finish and see if we have the number which we are searching for. We can take the advantage of the sorted property and do much better as follows.

- check the mid element
  - if it is the one which we are looking for then we are done
  - If not,
    - if it is greater than what we are looking for, then **recursively search in the left half** as all elements on right will be greater and there is no way we can find what we are looking for as the array is sorted
    - If it is less than what we are looking for, then **search in the right half** as all elements in the left half are smaller
  - if neither of the above are true then that element does not exist in this – not found.


1
Smaller, so search in right half
5 7 10 13 20 22 33 41 46 51 111
2
Greater, so search in left half
22 33 41 46 51 111
3
Greater, so search in left half
22 33 41 46 51 111
4
YES, Found it.
46 51 111

(https://sadakurapati.files.wordpress.com/2015/05/recursion-binary-search.png)
Recursion – Binary Search comparisons

Following is the java code for binary search algorithm,

```java
public static int search(int[] input, int element, int start, int end)
    if(end < start) return - 1;
    int mid = (start + end)/2;
    if(input[mid] == element) return mid;
    if(element < input[mid]) {
        return search(input, element, start, mid -1);
    } else {
        return search(input, element, mid + 1, end);
    }
}
```

Summary

There are many different kinds of problems which we can resolve and most of them tend to following these patterns –

- Dive into half and recurse on left or right (sometimes both) half – like binary search
- Solve first/last and then recurse on remaining
- Choose one or make one decision and recurse on remaining one

- Recurse first and then process – like power function as we need the value from recursive call to calculate the current value
- Process and then recurse – like binary search and some of graph searching algorithms
- Bottom up vs Top down recursion
- Base case is important and also make sure every recursive call is made for smaller problem – otherwise we end up in infinite loop

Hope this blog gave you some insights on recursion and the best way to get comfortable with this is to practise as many problems as possible. Following are few other problems,

- N – Queens problems (https://sadakurapati.wordpress.com/2013/12/10/n-queens-backtracking-algorithm/)
- Towers of Hanoi (https://sadakurapati.wordpress.com/2013/11/05/algorithm-tower-of-hanoi-game-with-k-towers/)
- Permutations of given input
- Edit distance – String similarity
- Knapsack (https://sadakurapati.wordpress.com/2013/11/30/algorithm-knapsack/)
- Factorials
- Fibonacci numbers

By Sada Kurapati • Posted in Algorithm, Graphs, Recursion, Trees • Tagged algorithm, backtracking, binary search, java, N queens, palindrome, pow function, recursion
OCT **15** 2014

Graphs & Trees – Shortest Path (BFS, Dijkstra, Bellman Ford)

Before exploring this topic, it will help if you go through my previous blog on graph basics (https://sadakurapati.wordpress.com/2014/01/06/graph-tree-algorithms/) which covers the different types of graphs, their representations & different searching techniques.
The motivation behind the shortest paths are many, for example finding a quick route between two cities, search engine crawling, data transfer to destination over the network, finding relationships on Facebook, Linkedin and many more. Following are the few key elements which we need to keep in mind while implementing any algorithm on graphs.

- The type of graph – Directed vs undirected, weights (+ve & -ve)
- Possibility of cycles – one of the most important
- Density of the graph – this will affect performance a lot
- Do we really need an optimal path – in some cases, it will take a lot of time to calculate this and we might be happy with sub optimal path itself
- And finally, the meaning/definition of shortest – number of edges or vertices, minimum weight in terms of weighted graphs etc.

Basically, if we don't have any weights and the shorted path is the minimum number of edges/vertices, then it is a straightforward problem as we can use Breadth First Search or Depth First Search algorithm to find it. The only additional information which we need to keep in track while doing BFS/DFS is the predecessor information so that we can keep track of the shortest path.

Shortest Path – by BFS

Breadth First Search can be used to find the shortest paths between any two nodes in a graph.
To understand BFS in more details, check this post (https://sadakurapati.wordpress.com/2014/01/06/graph-tree-algorithms/)
Following is the pseudo instructions for this algorithm.

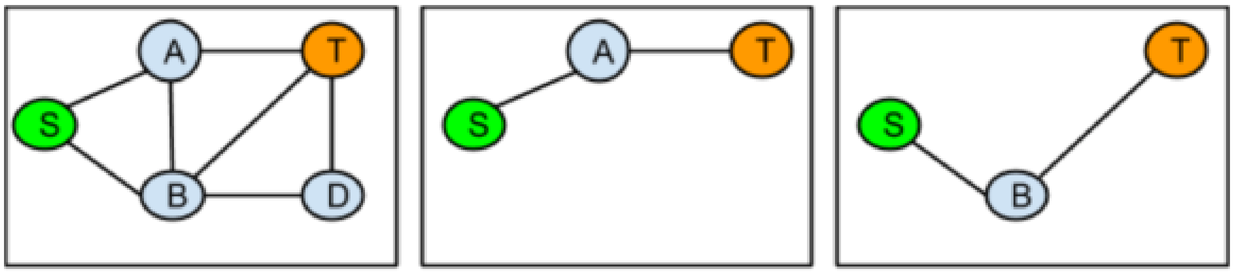
- start at the source node S
- put this in current level bucket
- while we have not ran out of graph (next level bucket is not empty) and not found our target node
    - take a node from current level bucket – call it C
    - for each node reachable from current node C – call it N
        - (we can also check to see if it is visited node and skip it to avoid the cycles)
        - mark the predecessor of N as C
        - Check if N is our target node (that is have we reached the target node)
            - If YES, then exit
    - put this N in next level bucket
    - mark next level bucket as current level

The code for this will be almost same as BFS except we keep track of predecessors so that we can build the path. For this, we can simply use a dictionary. The code for this can be found in my previous blog here (https://sadakurapati.wordpress.com/2014/01/06/graph-tree-algorithms/).

Dijkstra’s algorithm

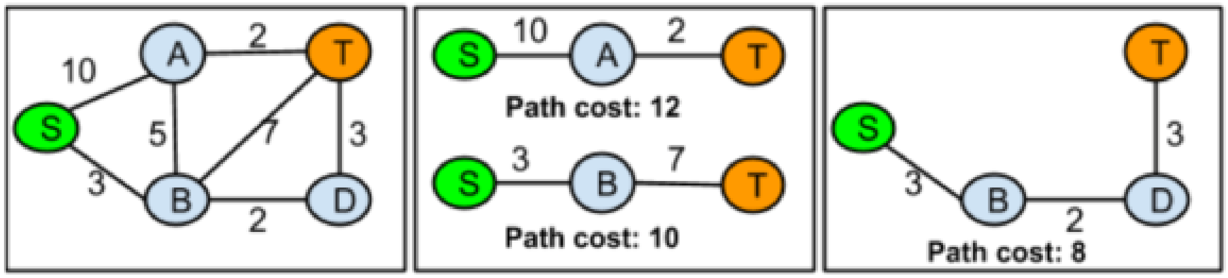
Dijkstra’s shortest path algorithm is mainly used for the graphs with +ve weights and in the case of finding single source shortest paths.
What does it mean by single source shortest path?
It means, given a source node (S), this algorithm will provide us shortest paths to all of the nodes in a graph which are reachable from S.
If we need a shortest path between two nodes, we simply stop(exit) executing this algorithm when we find the shortest path to the target node. It is sometimes very hard to find the stopping point for shortest path between two nodes, so we mostly perform full algorithm and then find the shortest path between nodes S and T. There might be also cases where we just need suboptimal solution (a reasonable shortest path) and in this case, we could save some good amount of execution time by simply stopping when we reach our expected suboptimal path. By the way, this algorithm is named after Edsger Dijkstra as he came up with this.

First let us look into an example on why we can't use BFS for weighted graph. Consider the below graph without weights and let us assume we want to find a shortest path between source node S and target node T.



Graph - Shortest paths by BFS

(https://sadakurapati.files.wordpress.com/2014/10/sp_by_bfs.png)
So in this case, we could end up with any one of the path shown above shortest paths from S to T. We don't bother which path we choose as both are the shortest paths between S and T in terms of number of nodes in a path. Now let us assign some weights.



Graph with +ve weights

(https://sadakurapati.files.wordpress.com/2014/10/gp_negative_weights.png)
The above diagram clearly tells us that both the paths by BFS (S–>A–>T & S–>B–>T) are not shortest and the actual shortest path is with the cost 8 and it is S –>B–>D–>T.
Based on this example, we can clear say that,

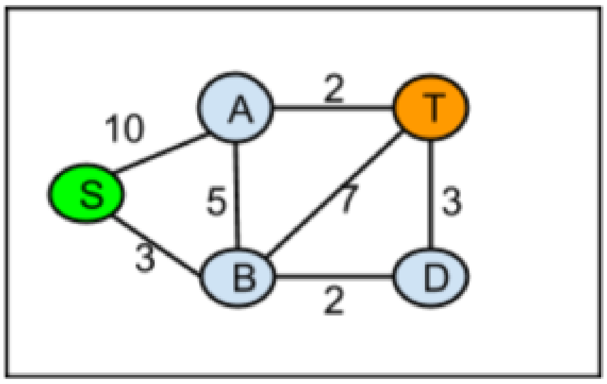
- Path with less number of nodes might not be the shortest path when there are weights
- BFS algorithm will not work in all of the cases when we have a graph with weights

Now let us look into the Dijkstra's algorithm and see how it finds out the shortest path. It basically takes each node and explores all of the paths from that node by calculating the cost and predecessors and performs this till we find the shortest paths. It will be exponential if we simply do the above approach but Dijkstra's algorithm uses the following two techniques to avoid exploration of exponential paths.

- Relaxation
  - Initialize the path cost of all of the nodes to Infinity
  - Then whenever we reach a node, we take the minimum of (older cost, path cost of predecessor + cost between predecessor and this node)
- Topological sort
  - The order of processing each node from graph hugely affects the total execution time. Will look into in detail in below sections on this.

Relaxation Technique

To explain this, let us consider the same example.



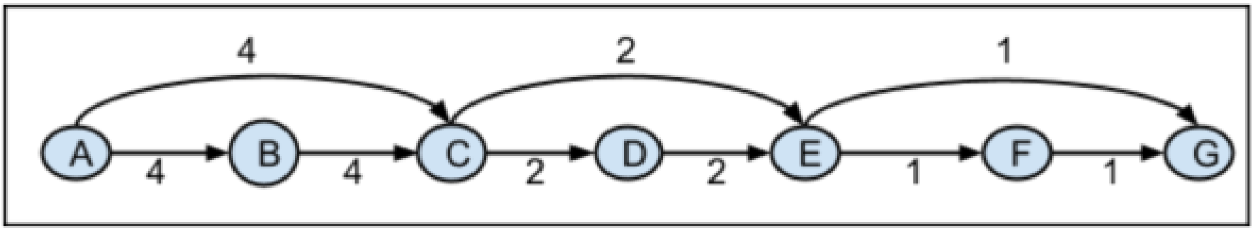
(https://sadakurapati.files.wordpress.com/2014/10/gp_weights_1.png)
There are many paths between S to T and for simplicity, let us assume that we reach node T in the below order.

- S → A → T (cost: 12)
  - In this case, we are reaching T from A, so the total cost is S→ A path cost + cost from A → T and the cost is 10 + 2 = 12
  - So this path cost is MIN (Infinite, 12) = 12. Note that we initialize the cost as Infinity. Update the predecessor of T as A.
- S → B → T (cost: 10)
  - Total cost is cost of S to B + cost of B to T. Cost is 3 + 7 = 10
  - So this path cost is MIN(12, 10) = 10. Found a new path. Update the predecessor of T as B.
- S → A → B → T (cost: 22)
  - Cost is MIN (10, 22) = 10. No change
- S → B → D → T (cost: 8)
  - Cost is MIN (10, 8) = 8. Found a new path. Update the predecessor of T as D.

So once we done with the algorithm execution, we will have the shortest path from S to T as S→ B → D → T.

Topological sort

Now you might be wondering why we need a topological sort at all. Does the relaxation technique won't sort out the problem completely? To answer this, we need to look into a well known example.



(https://sadakurapati.files.wordpress.com/2014/10/gp_topological_sort.png) In this, first let us calculate the weights from A to G by just taking the direct straight links. Then the shortest paths for the vertices B to G will be B-4, C-8, D-10, E-12, F13 & G-14. Now let us look at the relaxation. E → G, there is an edge with weight 1, so the shortest path will be 13 (A-E 12 and E to G 1 = 13). Now look at the edge from C to E. If we take that then the shortest path to E will become 10 (A → C 8, C → E 2), now the shortest path to E becomes 10, then we need to change the weights of F to 11 (from 13) and G to 11 (from 13). This is just an example where the dynamic paths can become exponential and it really hurts the performance of our shortest path algorithm.

Dijkstra uses a priority query which will help us picking up the vertices with minimum path which leads to a better topological sort.
Following is the pseudo instruction-
Dijkstra algorithm for shortest paths to all the vertices in a graph from source vertex 's'

```
◦ Initialize a dictionary S which holds the shortest paths already calculated.
◦ Add all of the vertices to a priority queue – Q.
◦ Set the distance from s to s as zero and to all other vertices as infinite.
◦ while the Queue is not empty
    ◦ take a vertex 'u' from query – extract min
    ◦ add u to S
    ◦ for each vertex v which is adjacent to u,
        ◦ relax u → v. relaxation step. Minimum (shortest path to u + weight of edge u to v,
          existing shortest path)
```

Finally, **S** contains the shortest paths to all of the nodes from graph from source vertex **s**. Basically, it is a greedy algorithm. The topological sort it uses is the order of picking up a vertex from queue with minimum path weight.

Following is the pseudo code
G – graph, s – source, Q – priority query, dist – dictionary stores vertex and shortest path
function Dijkstra (G, s)

```
◦ Q, dist = InitializeGraph(G, s)
◦ while Q is not empty
    ◦ u = Q.extract_min() // retrieves the vertex with minimum path distance.
    ◦ mark u as visited
    ◦ for each adjacent vertex v of u
        ◦ if v is not yet visited
            ◦ relax(u, v)
```

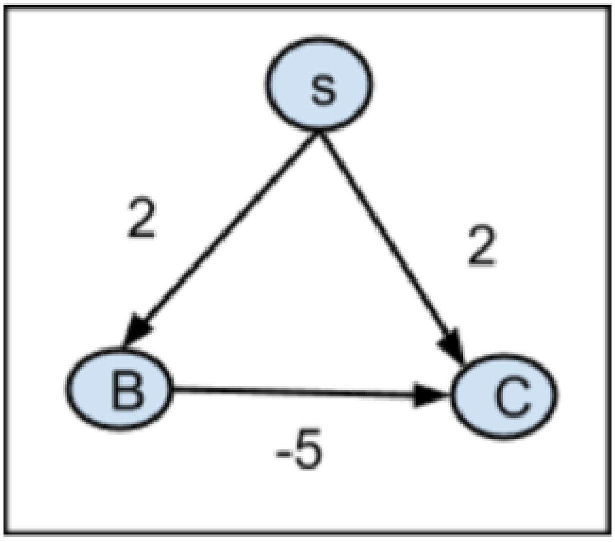
// initializes a graph and returns priority queue
function InitializeGraph(G, s)

```
- dist[s] =0
- for each v in G
    - if v not equal to s
        - dist[v] = infinity
    - Q.add(v, dist[v]) – priority queue
```

return Q, dist
function relax(u, v, Q)

```
- newWeight = dist[u] +  weight from u to v
- if newWeight < dist [v]
    - dist[v] = newWeight
    - Q.decrease_priority(v, newWeight)
```

Why does Dijkstra won't work for a graph with negative edges?
The problem is the relaxation step and the condition of running the algorithm till we don't have any edges which can be relaxed.
Let us look at the following graph and think for a few seconds why it will fail.



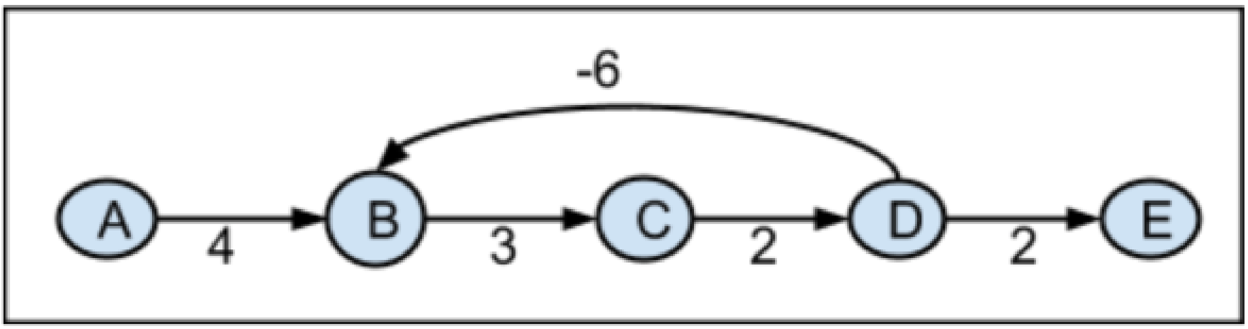
(https://sadakurapati.files.wordpress.com/2014/10/gp_negative_weights_1.png)
Here, when we start at s (source),

- we set s to s as 0 and all other (s → b & s → c) as infinite
- Then we pull c, we relax that edge (min (infinity, 2)), mark c as visited
- then we pull b, we relax that edge (min (infinity, 2)), mark b as visited.
- No more edges, the algorithm is done and it says the shortest path from s to c is 2.

But in fact, the shortest path from s to c is s → b → C is -3. The simplest reason is, Dijkstra algorithm assumes that the weights will only increase. In this case, when it visited(pulled out from Q) c, it marked it as visited and never going to visit again as it assumes that shortest path to c from anywhere else will be more than 2.

Bellman-Ford

Now let us look into Bellman Ford algorithm and see how it solves the shortest path problem for a graph with negative weights. This algorithm closely follows the Dijkstra algorithm and uses relaxation technique. The only difference is, it executes this |V| – 1 times for each edge where |V| is the number of vertices in the graph.
There is also another aspect which it determines is the negative cycle paths. Let us look at the following example.



(https://sadakurapati.files.wordpress.com/2014/10/belman-ford.png)
In this example, can you determine the shortest path distance from B → D? No, because every time we cycle between B → C → D → B, it reduces by 1. So we can cycles through this infinite times and there is no way we can identify the shortest path from B → D in this graph. So in this case, Bellman-Form algorithm throws error saying it can't find the short path.
Following is the pseudo code for Bellman-Ford
V[] – vertices, E[] – edges, s – source vertex
function Bellman-Ford(V[], E[], s)

- // Initialize the graph
- dist[] = ***InitializeGraph***(V[], s)
- for 1 to size(V) -1   // This is the diffrence from Dijkstra
  - for each edge e (u, v) in E
    - ***relax***(u, v, dist[])
- if ***haveNegativeCycles***(E[], dist[])
  - throw new Error("Graph have negative cycles")

return dist

// initializes a graph
function InitializeGraph(V[], s)

- for each vertex v in V[]
  - dist[v] = infinity
- dist[s] = 0

return dist[]

// relaxation
function relax(u, v, dist[])

- newWeight = dist[u] + weight from u to v
- if newWeight < dist [v]
  - dist[v] = newWeight

function haveNegativeCycles(E[], dist[])

```
for each edge e (u, v) in E
    if dist[u] + weight from u → v < dist[v]
        return true  // yes, if can find a edge to relax, then it contains negative cycles
return false // no negative cycles
```

So basically it follows the Dynamic Programming (DP) approach. When we relax all of the edges once, the graph (dist[]) will have shortest paths with 1 edge, after two cycles, we will have shortest paths with 2 edge paths and so on. After we perform this |V| – 1 times, we will have shortest paths from the source to all of the vertices and we simply throw the error when it contains negative cycles.

Thats all about shortest paths. These are just basics and we do have more complex ones with heuristics and NP-Hard problems like Traveling Salesman problem, but it will be very hard to describe them in a blog post.

Thank you for your time and feel free to provide your feedback through comments.

By Sada Kurapati • Posted in Algorithm, Graphs, Trees • Tagged algorithm, Belman-ford, BFS, Dijkstra, graphs, shortest path
JAN **6** 2014

Graph & Tree algorithms

Graphs and Trees are at the heart of a computer and very useful in representing and resolving many real world problems like calculating the best paths between cities (Garmin, TomTom) and other famous ones like Facebook search and Google search engine crawling. If you know any programming language, you might have heard of garbage collection and for that the typical graph search called Breadth First Search is used. In this post, I will try to cover the following topics.

- Graphs – directed, undirected and special case of directed which are called Directed Acyclic Graph(DAG)
- Trees – Binary trees and also most useful version of it, the Binary Search Trees (BST)
- Standard traversal techniques – Breadth First Search (BFS), Depth First Search (DFS).

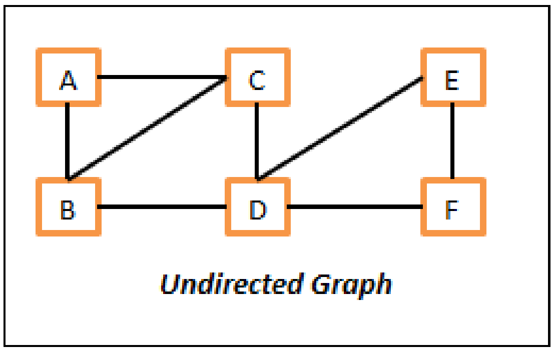
I know it will be a bit lengthy but I assure you it will be worth going through. I will try to cover the following topics in another post.

- Overview of shortest path algorithms – BFS, Dijkstra and Bellman Ford
- BST Search, Pre-order, In-order and Post-order.

Graphs

A graph consists of vertices (nodes) and edges. Think vertex as a point or place and the edge is the line/path connecting two vertices. In a typical graph and tree nodes there will be some kind of information is stored and the vertices will be associated with some cost. There are mainly two types of graphs and they are **undirected** and **directed** graphs.

In undirected graphs, all of the edges are undirected which means we can travel in either direction. So if we have an edge between vertex A and B, then we can travel from A to B and also from B to A. Following is the sample undirected graph.

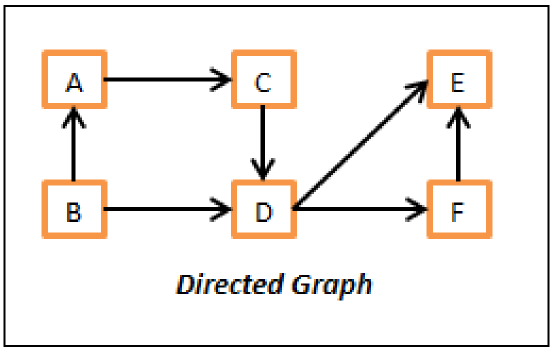


(https://sadakurapati.files.wordpress.com/2014/01/undirected-graph.png)
A Undirected Graph

Here in this undirected graph,

- The vertices are – A, B, C, D, E and F.
- The edges are – {A, B}, {A, C}, {B, C}, {B, D}, {C, D}, {D, E}, {D, F} and {E, F}. Here edge {A, B} means, we can move from A to B and also from B to A.

In directed graphs, all of the edges are directed which means we can only travel in the direction of an edge. So for example, if we have an edge between vertices A and B, then we can only go from A to B and not from B to A. If we need to go from B to A, then there should be another edge which should point B to A. Following is the sample directed graph.



(https://sadakurapati.files.wordpress.com/2014/01/directed-graph.png)
Directed Graph

Here in this directed graph,

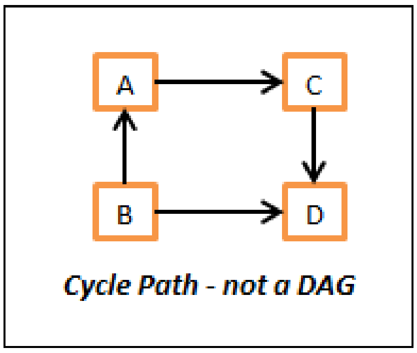
- The vertices are – A, B, C, D, E and F.

- The edges are – (A, C), (B, A), (B, D), (C, D), (D, E), (D, F) and (F, E). Here edge (A, C) means, we can only move from A to C and **not** from C to A.

The graphs can be mixed with both directed and undirected edges but let us keep this aspect away in this post to understand the core concepts better. Just an FYI the undirected edges are represented by curly braces '{}' where as directed edges are represented by '()'.

Directed Acyclic Graphs – DAG

This is a variation of standard directed graph without any cycle paths. It means there is no path where we can start from a vertex and comeback to the same starting vertex by following any path. By the way, **cycles in the graph** are very important and we **need to be very careful about** it. Otherwise, we could end up writing infinite algorithms which will never be completed.



(https://sadakurapati.files.wordpress.com/2014/01/cycle-in-graph.png)
Cycle in Graph

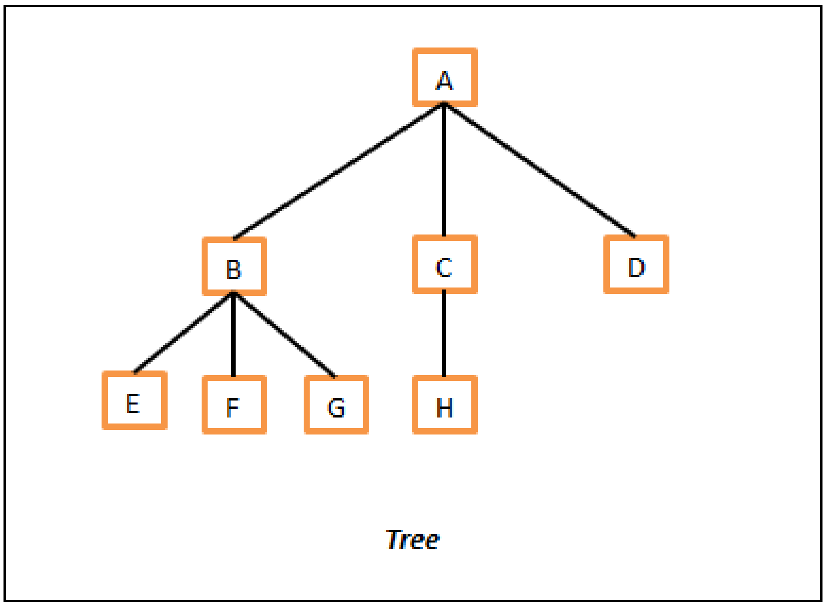
If we carefully look at the above listed directed graph, there is a cycle path and because of that, this graph is not a DAG.

Trees

This is another variation or we may call it subset of a graph. So we can say, all trees are graphs but not all graphs are trees. It is almost similar to the real world tree. It will have one top level vertex which is called root, and every node will have either zero or more child vertices. The best examples of tree data structure are family relationship and a company organization chart. Few important things to note down here,

- There will not be any cycles in this.
- Every node will have only one parent except the root which will not have any.

Following is a sample tree structure.

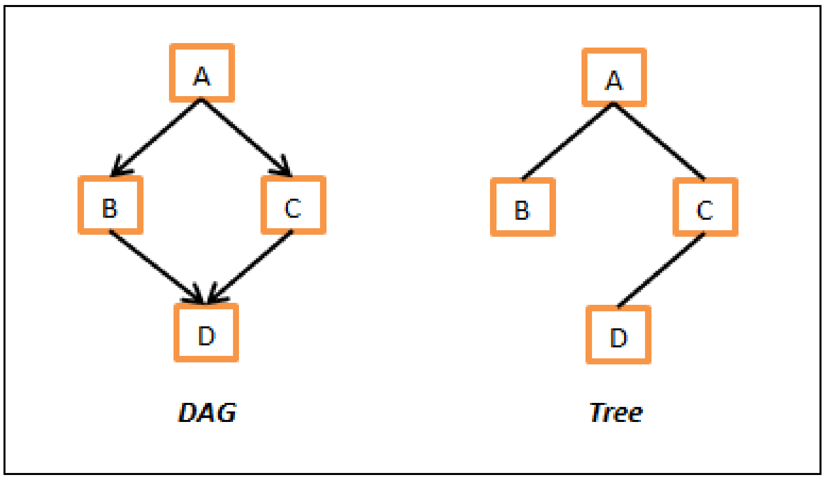


(https://sadakurapati.files.wordpress.com/2014/01/a-tree.png)
A Tree

As a standard the tree is represented from top to bottom. In this, the top node A is called as root of this tree as there are no parents to it. Then all of the child vertices (B, C and D) of A are represented on next level. All of the nodes which don't have any child nodes (E, F, G, H and D) are called leaf nodes.

DAG vs Tree

As you can see by now both the DAG and tree will not have any cycle paths but there is a key difference that is, in DAG there can be nodes with multiple parents but where as in tree, every node will have either a single parent or none. Following diagrams depicts this difference.

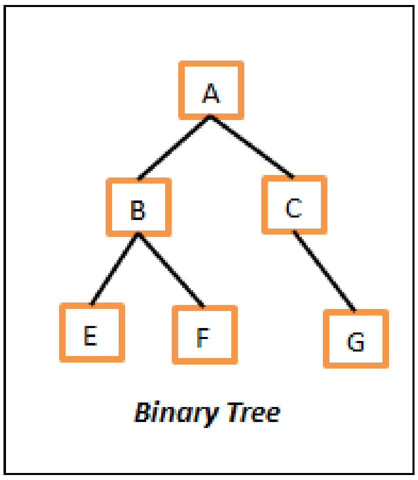


(https://sadakurapati.files.wordpress.com/2014/01/dag-vs-tree.png)
DAG vs Tree

In the above DAG example, D has two parents B and C and this is not possible in the tree. In tree example above, D has just one parent C.

Binary Tree

This is a version of the tree structure and in this every node can have strictly **at most two child nodes**. That means a node either can have two child nodes, just one or simply no child nodes in which case we call them leaf nodes. Following is a sample binary tree.



(https://sadakurapati.files.wordpress.com/2014/01/binary-tree.png)
Binary Tree

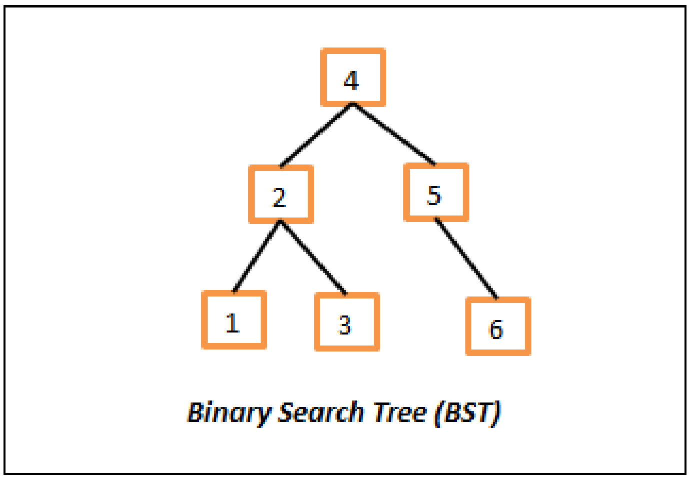
Binary Search Tree (BST)

This is a most useful version of the Binary Tree in which every node (including root) holds the following two properties.

- The **left child** node (if exists) should contain the value which is **less than or equal** to the parent.
- The **right child** node (if exists) should contain the value which is **greater than or equal** to the parent.

The above ordering property is really useful which we can understand clearly when we go through some of the search algorithms (especially the Binary search).

For simplicity, we will use the numbers in the nodes to show this relationship but the nodes can contain any information as long they are comparable. That means we should be able to compare them in some way and tell, whether those are equal or whether which one is greater. Following is a sample BST,



(https://sadakurapati.files.wordpress.com/2014/01/binary-search-tree.png)
Binary Search Tree

Data Structure representation

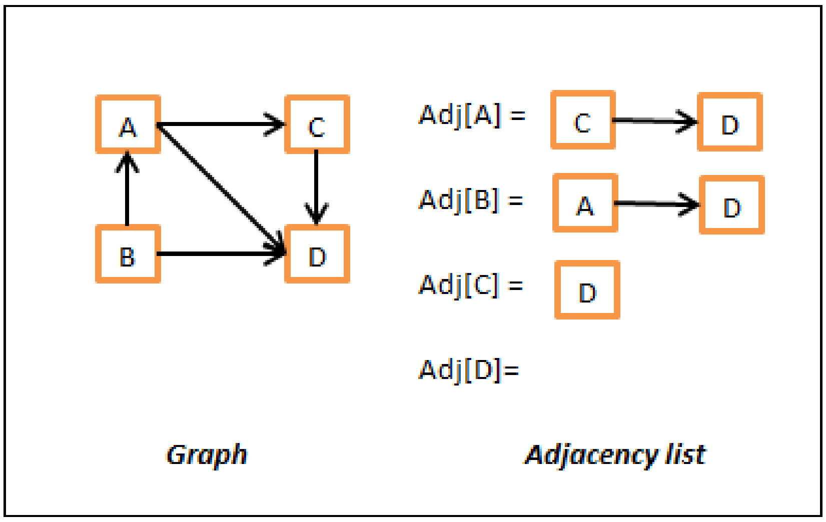
Hopefully all of the above details provide the insights on graphs and tree data structures. Before we get on to the actual traversals and searching algorithms, it is essential and useful to understand how they are represented or stored them in programming languages.

Following are the two mostly used representations.

Adjacency Lists

In this, the node is represented by an object and the edge is represented by the object references. Every object will have some data and a list of adjacent node references or a method which can return the list of adjacent nodes. If there are no neighbors, then it will be null or empty. So in nutshell, we have some mechanism to quickly find out neighbors for a given node.

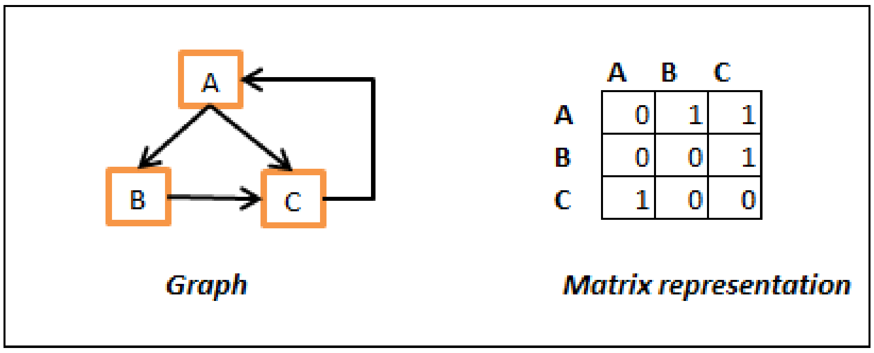
There is also another way to represent using adjacency lists which having a map which contains the key as the node and the value is list containing all of its neighbors. In this case, we can simply get all of its neighbors of a node N by simply looking for a key N in the map. That is we could say the neighbors of node N are simply Adj[N] where Adj is the adjacency map. Following is a sample representation of this representation.



(https://sadakurapati.files.wordpress.com/2014/01/graph-adjacency-list-representation.png)
Graph Adjacency list representation

Matrix representation

In this, we will use a double dimensional array where rows and columns represent the nodes and will use some kind of special value or actual edge cost as the value if they are connected. Following is sample example of representation.



	A	B	C
A	0	1	1
B	0	0	1
C	1	0	0

(https://sadakurapati.files.wordpress.com/2014/01/graph-matrix-representation.png)
Graph matrix representation

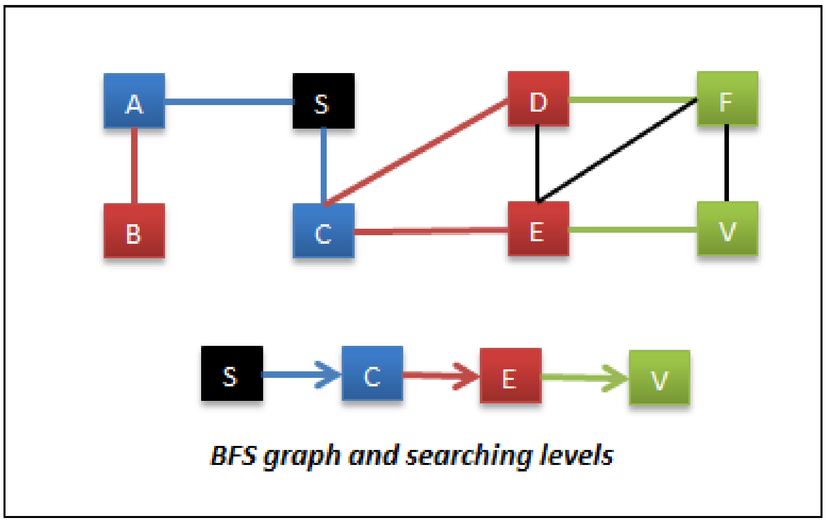
In the above, we have used 1 to represent the edge and 0 for no edge and we are looking from row to column. That is Array[B][C] is 1 as there is edge from B to C but Array[C][B] is zero as there is no edge between C to B. We could also use some value to represent the actual cost and use some sentinel (infinity or -1) value to represent no connection between the nodes. Yes, it is not a regulatory object oriented design but it have its own advantages like accessing random node and get the edge cost of a neighbor in constant time.

Searching Algorithms

There are two main search techniques which work fine on both the graphs and all versions of trees. It is just that we need to be careful about the cycles in graphs.

Breadth First Search (BFS)

In this, we will search or examine all of the sibling nodes first before going to its child nodes. In other words, we start from the source node (root node in trees) and visit all of the nodes reachable (neighboring nodes) from the source. Then we take each neighbor node and perform the same operation recursively until we reach destination node or till we run out of graph. The following color coded diagrams depicts this searching.



(https://sadakurapati.files.wordpress.com/2014/01/breadth-first-search.png)
Breadth First Search

In this, the starting node is S and the target node is V. In this,

- Visit all of the nodes reachable from S. So we visit nodes A & C. These are the nodes reachable in 1 move or nodes at level 1.
- Then visit the nodes reachable from A & C. So we visit nodes B, D & E. These are the nodes reachable in 2 moves or we can say nodes at level 2.
- Then visit the nodes reachable from B, D & E. So we visit nodes F & V. These are the nodes reachable in 3 moves or we can say at level 3.

Advantages of this approach are,

- The path from which we first time reach a node can be considered as shortest path if there are no weights/costs to the edges or they might be constant. This is called single source shortest paths.
- This can be used in finding friends in social network sites like Facebook, Google+ and also linked in. In linked in, you might have seen the 1st, 2nd or 3rd degree as super scripts to the friend's names – it means that they are reachable in 1 move, 2 moves and 3 moves from you and now you might guess how they calculate that J

BFS implementation

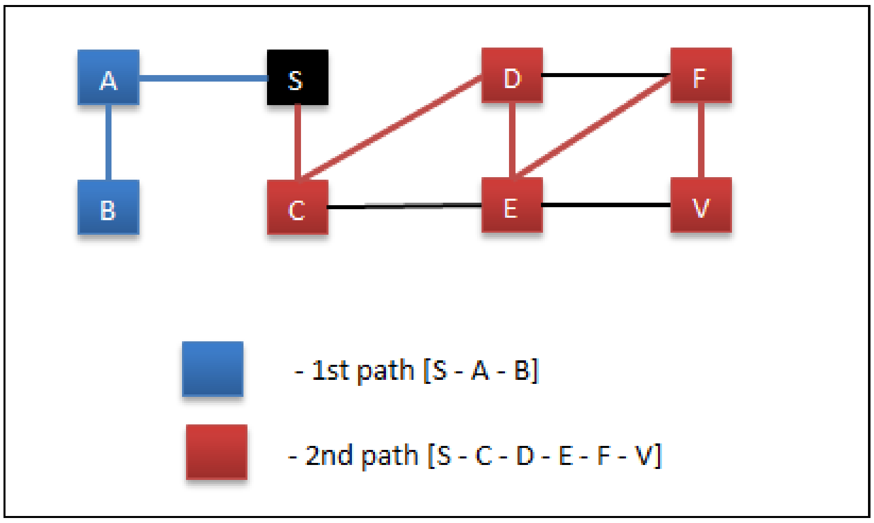
We can implement this in multiple ways. One of the most popular ways is by using Queue data structure and other by using lists to maintain the front and next tier nodes. Following is the java implementation of this algorithm.

```java
public static void graphBFSByQueue(Node<String> source) {
  //if empty graph, then return.
  if (null == source) {
    return;
  }
  Queue<Node<String>> queue = new LinkedList<Node<String>>();
  //add source to queue.
  queue.add(source);
  visitNode(source);
  source.visited = true;
  while (!queue.isEmpty()) {
    Node<String> currentNode = queue.poll();
    //check if we reached out target node
    if (currentNode.equals(targetNode)) {
      return; // we have found our target node V.
    }
    //Add all of unvisited neighbors to the queue. We add only unvisi
    for (Node<String> neighbor : currentNode.neighbors) {
      if (!neighbor.visited) {
        visitNode(neighbor);
        neighbor.visited = true; //mark it as visited
        queue.add(neighbor);
      }
    }
  }
}

public static void graphBFSByLevelList(Node<String> source) {
  //if empty graph, then return.
  if (null == source) {
    return;
  }
  Set<Node<String>> frontier = new HashSet<Node<String>>();
  //this will be useful to identify what we visited so far and also i
  //if we dont need level, we could just use a Set or List
  HashMap<Node<String>, Integer> level = new HashMap<Node<String>, In
  int moves = 0;
  //add source to frontier.
  frontier.add(source);
  visitNode(source);
  level.put(source, moves);
  while (!frontier.isEmpty()) {
    Set<Node<String>> next = new HashSet<Node<String>>();
    for (Node<String> parent : frontier) {
      for (Node<String> neighbor : parent.neighbors) {
        if (!level.containsKey(neighbor)) {
          visitNode(neighbor);
          level.put(neighbor, moves);
          next.add(neighbor);
        }
        //check if we reached out target node
        if (neighbor.equals(targetNode)) {
          return; // we have found our target node V.
        }
      }//inner for
    }//outer for
    moves++;
    frontier = next;
  }//while
}

public static void treeBFSByQueue(Node<String> root) {
  //if empty graph, then return.
  if (null == root) {
    return;
  }
  Queue<Node<String>> queue = new LinkedList<Node<String>>();
  //add root to queue.
  queue.add(root);
  while (!queue.isEmpty()) {
    Node<String> currentNode = queue.poll();
    visitNode(currentNode);
    //check if we reached out target node
    if (currentNode.equals(targetTreeNode)) {
      return; // we have found our target node V.
    }
    //Add all of unvisited neighbors to the queue. We add only unvisi
    for (Node<String> neighbor : currentNode.neighbors) {
      queue.add(neighbor);
    }
  }
}

public static void treeBFSByLevelList(Node<String> root) {
  //if empty graph, then return.
  if (null == root) {
    return;
  }
  List<Node<String>> frontier = new ArrayList<Node<String>>();
  //add root to frontier.
  frontier.add(root);
  while (!frontier.isEmpty()) {
    List<Node<String>> next = new ArrayList<Node<String>>();
    for (Node<String> parent : frontier) {
      visitNode(parent);
      //check if we reached out target node
      if (parent.equals(targetTreeNode)) {
        return; // we have found our target node V.
      }

      for (Node<String> neighbor : parent.neighbors) {
        next.add(neighbor);
      }//inner for
    }//outer for
    frontier = next;
  }//while
```

```
}
```

Depth First Search (DFS)

In this, we will search a node and all of its child nodes before proceeding to its siblings. This is similar to the maze searching. We search a full depth of the path and if we don't find target path, we backtrack one level at time and search all other available sub paths.



(https://sadakurapati.files.wordpress.com/2014/01/depth-first-search.png)
Depth First Search

In this,

- Start at node S.
  - Visit A
  - Visit B
  - Backtrack to A as B don't have any child nodes
  - Backtrack to S as A don't have any unvisited child nodes
  - Backtrack to node S
    - Visit C
    - Visit D
    - Visit E
    - Visit F
    - Visit V – reached searching node so exit.

The path to node V from S is "S – C – D – E – F – V" and the length of it is 5. In the same graph, using BFS the path we found was "S – C – E – V" and the length of it is 3. So based on this observation, we could say DFS finds a path but it may or may not be a shorted path from source to target node.

Few of advantages of this approach are,

- The fastest way to find a path – it may not be a shortest one.

- If the probability of target node exists in the bottom levels. In an organization chart, if we plan to search a less experienced person (hopefully he will be in bottom layers), the DFS will find him faster compared to BFS because it explores the child nodes lot earlier compared to BFS.

DFS implementation

We can implement this in multiple ways. One is using the recursion and other is using Stack data structure. Following is the java implementation of this algorithm.

```java
public static void graphDFSByRecersion(Node<String> currentNode) {
  if (null == currentNode) {
    return; // back track
  }
  visitNode(currentNode);
  currentNode.visited = true;
  //check if we reached out target node
  if (currentNode.equals(targetNode)) {
    return; // we have found our target node V.
  }
  //recursively visit all of unvisited neighbors
  for (Node<String> neighbor : currentNode.neighbors) {
    if (!neighbor.visited) {
      graphDFSByRecersion(neighbor);
    }
  }
}

public static void graphDFSByStack(Node<String> source) {
  //if empty graph, return
  if (null == source) {
    return; //
  }
  Stack<Node<String>> stack = new Stack<Node<String>>();
  //add source to stack
  stack.push(source);
  while (!stack.isEmpty()) {
    Node<String> currentNode = stack.pop();
    visitNode(currentNode);
    currentNode.visited = true;
    //check if we reached out target node
    if (currentNode.equals(targetTreeNode)) {
      return; // we have found our target node V.
    }
    //add all of unvisited nodes to stack
    for (Node<String> neighbor : currentNode.neighbors) {
      if (!neighbor.visited) {
        stack.push(neighbor);
      }
    }
  }
}

public static void treeDFSByRecersion(Node<String> currentNode) {
  if (null == currentNode) {
    return; // back track
  }
  visitNode(currentNode);
  //check if we reached out target node
  if (currentNode.equals(targetNode)) {
    return; // we have found our target node V.
  }
  //recursively visit all of unvisited neighbors
  for (Node<String> neighbor : currentNode.neighbors) {
            graphDFSByRecersion(neighbor);
  }
}

public static void treeDFSByStack(Node<String> source) {
  //if empty graph, return
  if (null == source) {
    return; //
  }
  Stack<Node<String>> stack = new Stack<Node<String>>();
  //add source to stack
  stack.push(source);
  while (!stack.isEmpty()) {
    Node<String> currentNode = stack.pop();
    visitNode(currentNode);
    //check if we reached out target node
    if (currentNode.equals(targetTreeNode)) {
      return; // we have found our target node V.
    }
    //add all of unvisited nodes to stack
    for (Node<String> neighbor : currentNode.neighbors) {
        stack.push(neighbor);
    }
  }
}
```

Following is the full sample code in Java and you can download and play with the code.

```
 click here to see full Solution.java
```

By Sada Kurapati • Posted in Algorithm, Graphs, Trees • Tagged algorithm, algorithms, BFS, Breadth first search, data structure, data structures, Depth First Search, DFS, java, programming
OCT **25** 2013

Quicksort – a practical and efficient sorting algorithm

Today let us look into another sorting algorithm. This is one of the popular and most efficient for majority of the practical usages. It has worst case time complexity of O(n^2) but performs lot better in average case with time complexity of O(n log n). This performs better than MergeSort (https://sadakurapati.wordpress.com/2013/08/20/algorithms-merge-sorting/) and HeapSort (https://sadakurapati.wordpress.com/2013/09/11/heap-ds-heapsort/)which also have same asymptotic time complexity O(n log n) on average case but the constant factors hidden in the asymptotic time complexity for quick sort are pretty small.

Overview

Like merge sorting, this follows the same Divide and Conquer paradigm. Following are the details on three steps of divide and conquer paradigm and how Quicksort performs the sorting of an array A[i…j] (i & j are indexes and i < j).

- **Divide**: divide (partition) the array into two sub arrays A[i…k-1] and A[k+1…j] (j < k < j), such that all of the elements in A[i…k-1] are less than are equal to A[k] and all of the elements are in A[k+1..j] are greater than A[k]. A[i..k-1] <= A[k] < A[k+1…j]
- **Conquer**: Sort the sub arrays A[i..k-1] and A[k+1…j] by calling quick sort recursively
- **Combine**: As the sub arrays are sorted, no need to perform any work in this step. The array A[i…j] is sorted

So on high level,

1. Pick an element – typically this is called pivot.
2. Move all elements smaller than equal to picked element to left
3. Move all elements greater then picked element to right
4. Perform steps 1 to 3 recursively on left and right sub arrays till we have array with one element were the divide bottoms out.

For now let us look at the coding aspects of the standard single pivot quick sort.

Algorithm

Following is the pseudo code for the quicksort algorithm.

```
QUICKSORT(A, p, r)
  If p < r
    q = PARTITION (A, p, r)
    QUICKSORT(A, p, q-1)
    QUICKSORT(A, q+1, r)
```

In Java,

```
public static void quickSort(int[] a, int p, int r) {
```

```java
    if (p < r) { // there are at least two elements in the array
      int q = partition(a, p, r);
      quickSort(a, p, q - 1);
      quickSort(a, q + 1, r);
    }
  }
```

To sort the array, the initial call would be QUICKSORT(A, 0, A.length-1) – for zero index based arrays. So if you look at the above algorithm it is clear that the meat of the logic goes into the partition mechanism.

Following is the partition logic. – Note that there are bunch of variations and we are discussing here the standard partition algorithm.

```
PARTITION(A, p, r)
  pivot = A[r]  - picking the last element as the pivot.
  i  = p -1
  for j = p to r-1
    if A[j] <= pivot
    i = i+1
    swap A[i] and A[j]
  swap A[i+1] with A[r] – this will put the pivot in right place
return i+1
```

In Java,

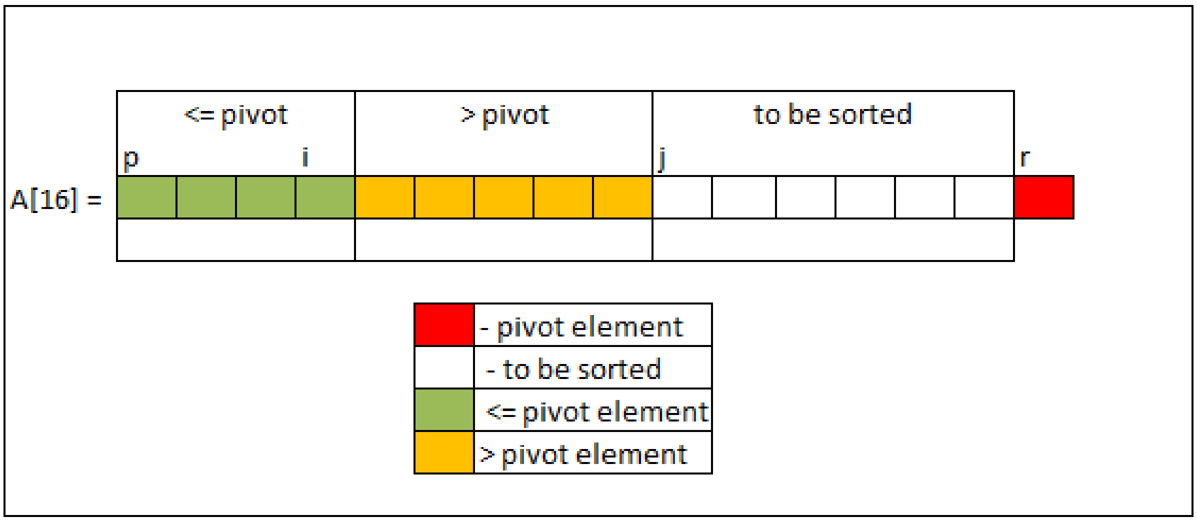
```java
    //performs the partition using last element
    public static int partition(int[] a, int p, int r) {
      //choosing last element in the array as pivot
      int pivot = a[r];
      int i = p - 1;
      int temp;
      for (int j = p; j < r; j++) {
        if (a[j] <= pivot) {
          i++;
          temp = a[i];
          a[i] = a[j];
          a[j] = temp;
        }
      }
      //now place pivot in right place
      temp = a[i + 1];
      a[i + 1] = a[r];
      a[r] = temp;
      return (i + 1);
    }
  }
```

Loop Invariant

A loop invariant is something which is true in following three stages.

1. Before loop execution begins – Initialization
2. While loop is executing – Maintenance
3. After the loop exits – Termination

Following is pictorial representation of quick sort loop invariant



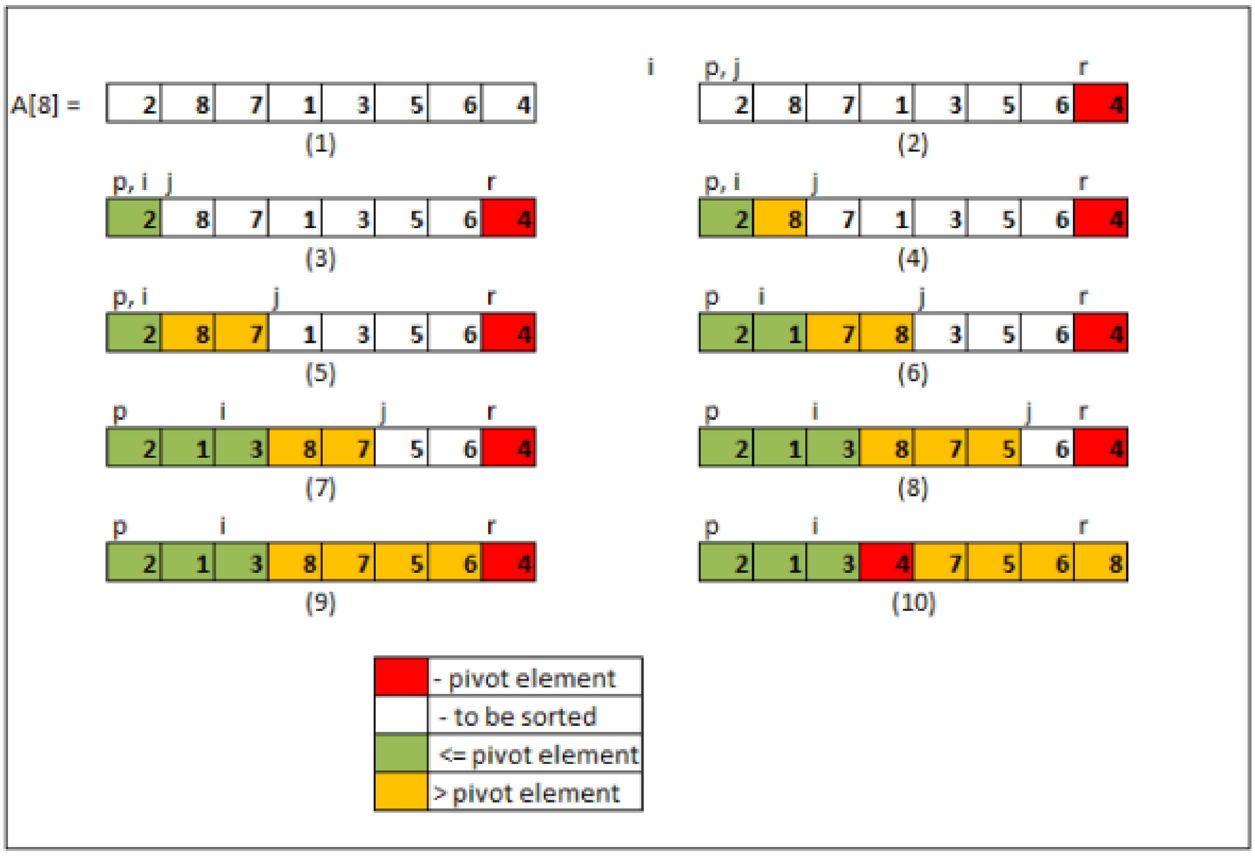
(https://sadakurapati.files.wordpress.com/2013/10/qsort_1.png)

So the loop invariant for quick sort is

- All elements in A[p…i] are less than or equal to pivot
- All elements in A[i+1…j-1] are greater than pivot
- A[r] is the pivot element

Example

Let us look at an example and some pictorial representation to understand the sorting and loop invariant clearly. The following diagram just depicts the first pass of partitioning.



(https://sadakurapati.files.wordpress.com/2013/10/qsort_2.png)

So once the above partition is done, then quick sort is called recursively on the left and right sub arrays to pivot element. Then the same partition is performed recursively till we have sub arrays with one element.

Variations

By the way, there are bunch of quick sort flavors based on the mechanism used to pick an element (pivot) to partition the array. Following are few of them,

- Pick single element as pivot – First, last, middle or some randomly located element (A[i], A[j] or A[k] where k=(i+j)/2 or A[p] – p is random index)
- Pick two or multiple elements as pivots – pick first and last or any two elements as pivot. There are different variations in this category – 3-way partition, Dual Pivot quick sort (by Vladimir Yaroslavskiy, Jon Bentley, and Joshua Bloch), etc.

Summary

Although the worst case time complexity of this sorting algorithm is quadratic, it performs very efficiently in average case.

- The worst case time complexity is O(n^2) and average case is O (n log n)
- This is very efficient and more often the best practical algorithm
- Bunch of the flavors are available and can easily customize or choose best suitable partition algorithm based on probable data sets
- The standard algorithm is not stable. If stability is required, we might need to use extra space

```
click here to see full Solution.java
```

By Sada Kurapati • Posted in Algorithm, Sorting • Tagged algorithm, algorithms, coding, data structure, divide and conquer, java, java collection, java enterprise edition, programming, quicksort, sorting, swap counting, time complexity
OCT **11** 2013

Insertion Sort – Efficient way of counting swaps

Problem Statement

Insertion Sort is a simple sorting technique which was covered in my previous blog post Algorithms – Insertion Sort (https://sadakurapati.wordpress.com/2013/08/17/algorithms-insertion-sorting/) . On a high level, the sorting will pick an element (iterated the process from 2nd element to last element), shifts all elements which are greater than the picked up element in the left part of the array by one position to the right and inserts it at correct position.

In this post, let's look at a special aspect of Insertion sorting which is counting how many shifts (or swaps) it takes for an insertion sort to finish sorting an array. So our challenge is how can we calculate or find out how many swaps are required for a given array efficiently.

Note that performance is the key element here.

Solutions

To find this, first we need to have a good understanding of Insertion and merge sorting algorithms. Following posts provide a very good details about these algorithm techniques.

- Insertion sorting (https://sadakurapati.wordpress.com/2013/08/17/algorithms-insertion-

sorting/)
- Merge sorting (https://sadakurapati.wordpress.com/2013/08/20/algorithms-merge-sorting/)

Approach 1 – Count swaps while performing Insertion sort

Perform the insertion sort and count every element swaps. Following is the code for this in Java.

```java
public static int getElementSwaps(int[] ar, int size) {
  //base case
  if (size <= 1) {
    return 0;
  }
  int shifts = 0;
  //starting at 2nd element as first element is already sorted.
  //Loop Invariant - left part of the array is already sorted.
  for (int i = 1; i < size; i++) {
    int moveMe = ar[i];
    int j = i;
    while (j > 0 && moveMe < ar[j - 1]) {
      //Move element
      ar[j] = ar[j - 1];
      --j;
      //increase the count as element swap is happend
      ++shifts;
    }
    ar[j] = moveMe;
  }
  return shifts;
}
```

How much time will it take? O(N^2).

The performance of this is acceptable for small input sizes and will start degrading for larger input sizes and becomes quickly unacceptable. Try to test this algorithm for input sizes with greater than a million elements.

How to make it better? Let's look at the problem statement again. We are interested in finding the swaps and not sorting the array (Hint: moving elements in array will take some time). So let us look at another option where we can avoid moving elements and just count the swaps.

Approach 2 – Finding bigger elements

Look at the insertion sorting algorithm closely and can you figure out when exactly an element swap is happening? The elements are swapped whenever a bigger element is found on left side of sorted array while trying to insert picked up element in correct position. So we can use this technique and count the number of swaps required without moving the elements in the array.

Following is the Java code for this technique.

```java
public static int getBiggerElementsCount(int[] ar, int size) {
  //base case
  if (size <= 1) {
    return 0;
  }
  int biggerElements = 0;
  //starting at 2nd element as first element is already sorted
  //and no swaps are required.
  for (int i = 1; i < size; i++) {
    //find number of elements less than current
    int moveMe = ar[i];
    for (int j = i - 1; j >= 0; j--) {
      if (moveMe < ar[j]) {
        //found small element on my (ar[i]) left
        ++biggerElements;
      }
    }
  }
  return biggerElements;
}
```

What is the time complexity of this? Asymptotically, it is still O(N^2) but it will perform much better compared to approach 1 as we tried out best to improve the constant factors in this asymptotic time complexity. Is there a better algorithm for this problem?

Approach 3 – Inversions and Merge Sort

An inversion is nothing but a situation where we need to invert (swap) the elements to make the array sorted. So in technical terminology, in an array A[1 .. n] there is a pair (i, j) such that **i < j** (<=n) and **A[i] > A[j]**. So to figure out how many swaps are required to sort using insertion sorting algorithm, we need to find our number of inversions required to make the array sorted.

One of the best approaches for this is to modify the merge sort algorithm so that we can count the inversions in that array while performing the sorting. For more details on merge sort, please check merge sort post (https://sadakurapati.wordpress.com/2013/08/20/algorithms-merge-sorting/)

If you can recall, the merge sort divides the array recursively till it reaches array with single element (base case, array with 1 element is sorted) and then merges left and right sorted arrays using a merge routine. So how do we count the inversions in merge routine? For each element in left array, we would like to know how many elements in right are smaller and this will give us the required inversions. To this, we need to look into the two cases where this can happen in merge routine.

- Whenever we put an element from left array, we need to increment the inversions with the count of elements [j] which are already added to result/merged array
- Once one of the array is completely merged/exhausted(after merge loop), if there are any elements left in left array, then we need to increment the count of inversions with (number of elements already merged from right array * elements left out in left array)

Let us take an example here to understand this.

left[] = {5, 8, 10} and right[] = {2, 4, 6}

Merge routine steps

- **Inversions = 0**;
- Check first element from left (5) and right (2), insert 2 into result array – result[] = {2}
- Check first element from left (5) and right (4), insert 4 into result array – result[] = {2, 4}
- Check first element from left (5) and right (6), insert 5 into result array – result[] = {2, 4, 5}
  - Here, we are inserting left element, so we need to increment the inversions with number of elements from right array already placed in result. So the **inversions += 2** (as element 2 and 4 are already placed in result array)
  - Check first element from left (8) and right (6), insert 6 into result array – result[] = {2, 4, 5, 6}
  - Place remaining elements from left to result array – result[] = {2, 4, 5, 6, 8, 10}
    - Inversions += (number elements left in left array * elements already placed in result from right array)
    - **Inversions+= 2 * 3**.

So the total number of inversions or swaps required for sorting input[] = {5, 8, 10, 2, 4, 6} array are **8**.

As we are programmers, you can execute the following code and validate this swaps count.

```java
public static void getInversions(int[] nums, int left, int right) {
  if (left < right) {
    //Split in half
    int mid = (left + right) / 2;
    //Sort recursively.
    getInversions(nums, left, mid);
    getInversions(nums, mid + 1, right);
    //Merge the two sorted sub arrays.
    merge(nums, left, mid, right);
  }
}

private static void merge(int[] nums, int left, int mid, int right) {
  int leftLength = mid - left + 1;
  int rightLength = right - mid;
  int[] lAr = new int[leftLength];
  //Just for simplicity, we are creating this right array.
  //We could use actual nums array with mid and right indexes.
  //a place to improve memory foot print.
  int[] rAr = new int[rightLength];
  for (int i = 0; i < leftLength; i++) {
    lAr[i] = nums[left + i];
  }
  for (int i = 0; i < rightLength; i++) {
    rAr[i] = nums[mid + 1 + i];
  }
  int i = 0, j = 0, k = left;
  while (i < leftLength && j < rightLength) {
    if (lAr[i] <= rAr[j]) {
      nums[k] = lAr[i];
      inversions += j;
      i++;
    } else {
      nums[k] = rAr[j];
      j++;
    }
    k++;
  }
  //remaining iversions, using long cast as multiplication will be out
  //Integer range for large inputs
  inversions += (long) j * (leftLength - i);
  if (i >= leftLength) {
    //copy remaining elements from right
    for (; j < rightLength; j++, k++) {
      nums[k] = rAr[j];
    }
  } else {
    //copy remaining elements from left
    for (; i < leftLength; i++, k++) {
      nums[k] = lAr[i];
    }
  }
}
```

By the way, there is another elegant way to do this same using data structure Binary Indexed Tree. I am not familiar with that DS as of now and will circle back to this once I learn BIT data structure.

```
click here to see full Solution.java
```

By Sada Kurapati • Posted in Algorithm, Sorting • Tagged algorithm, algorithms, coding, data structures, insertion sorting, java, programming, sorting, swap counting, time complexity
SEP 11 2013

Heap DS & Heapsort

Let us understand one of the important data structure Heap which is very useful in building heap sort and also Priority Queues.
The heap in simple words is the visualization of an array in binary tree structure. How do we do that? Let us look at the following array and binary tree representation of it. By the way, binary tree is nothing but a tree where every parent node in the tree have maximum of two child nodes.

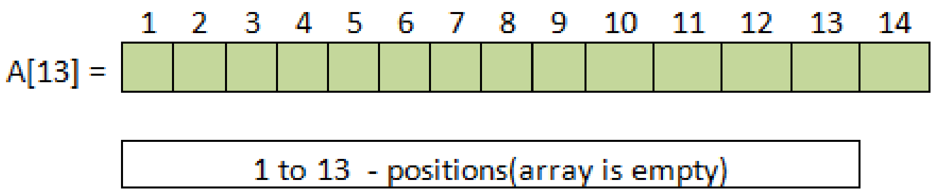


Figure – H.1, the array of elements

(https://sadakurapati.files.wordpress.com/2013/09/hs1.png)

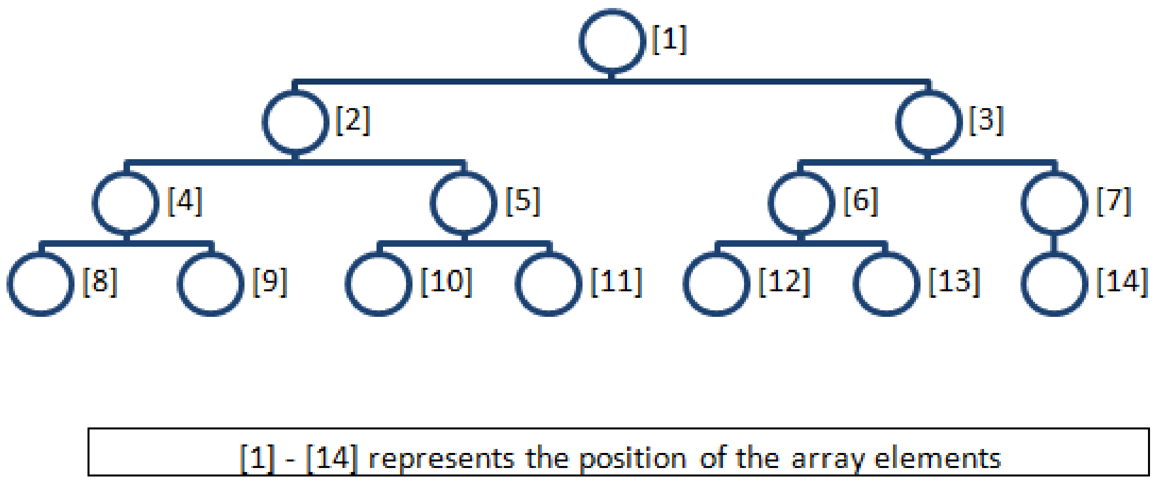


Figure – H.2, visualization of array with binary tree structure

(https://sadakurapati.files.wordpress.com/2013/09/hs2.png)

For simplicity, I took the empty array so that we don't confuse with the array elements and the positions/indexes. The **Figure – H.1** represents an array with 14 elements and **Figure – H.2** is the visualization of the same array with binary tree structure. Few things to observe here,

1. The first element of array is the root element.
2. The positioning (index) is sequential from **root to child** nodes from **left to right**.
3. For every i^{th} node,
   - Parent is positioned at floor of i/2.
   - Left child is positioned at 2*i
   - Right child is positioned at 2*I +1, that is (left child position) +1
4. Note that the parent, left and right child positions can be quickly calculated by single instruction in most of the current machines.
   - Parent – floor of n/2, which is simply left shifting the bits by 1 (i <<1)
   - Left child – 2 * i, which is simply right shifting the bits by 1 (i>>1)

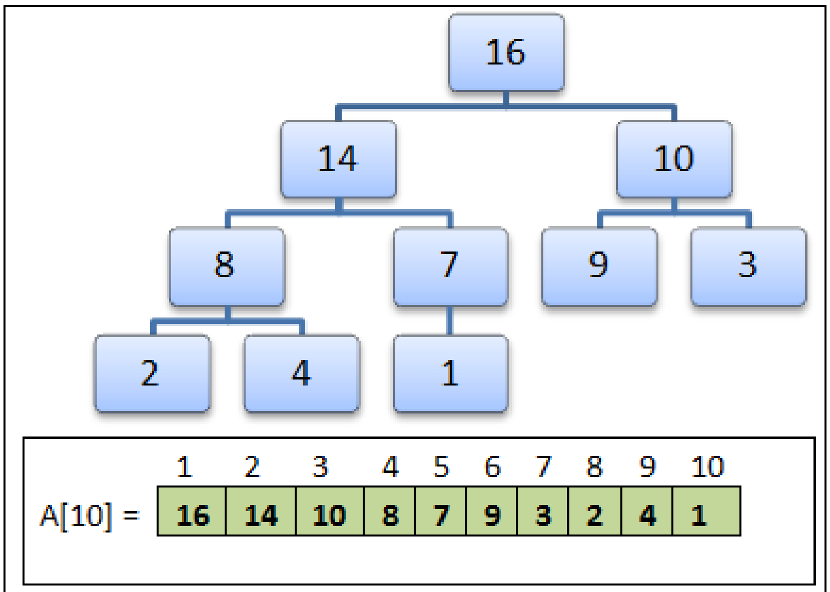
Right child – yes you guessed it right it is right shifting the bits by 1 and adding 1 or simply left(i)+1.

Now you might be wondering why we need to visualize this way. There are lot of advantages and the important one is we can perform the sorting in O(n log n).

There are two types of heaps,

- **Max** heap – keys of parent nodes are always **greater than or equal** to those of the children and the highest key is in the root node
- **Min** heap – keys of parent nodes are always **less than or equal** to those of the children and the **lowest key** is in the root node

For clarity, check the two types of heaps (Max & Min) and their tree structure visualization below.



	1	2	3	4	5	6	7	8	9	10
A[10] =	**16**	**14**	**10**	**8**	**7**	**9**	**3**	**2**	**4**	**1**

Figure H.3 - Max heap

(https://sadakurapati.files.wordpress.com/2013/09/hs3.png)

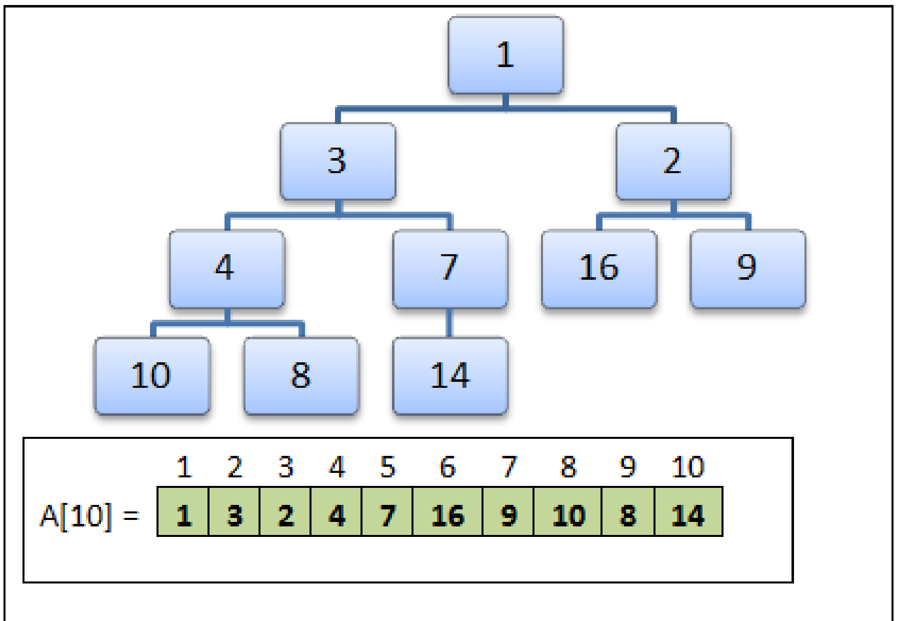


Figure H.4 - Min heap

(https://sadakurapati.files.wordpress.com/2013/09/hs4.png)

To understand, just check the parent (n/2), left child (2* i) and right child (2* i +1) positions in each of the array and tree structure in above examples

Now, let us get into the heap sorting. For simplicity, from now on we only look at the Max heaps and max heap sort (elements in ascending order). To understand this, we need to look at two concepts,

1. **maxHeapify** – This will perform the operations/swaps between elements and arrange the elements to hold the max heap property (parent element is greater than equal to child nodes)
   - Compare the elements at parent (give position i), left child and right child.
   - If parent is maximum, then we are done.
   - If any of the child node is maximum, then swap parent with that child node.
   - The above step might create a max heap property violation at swapped child node, so we simply call maxHeapify at this child node.
2. **buildHeap** – This will iterate throw all of the parent nodes and create a max heap.
   - For each node from n/2 to 1, call maxHeapify. Why n/2 to 1? Because all of the nodes from n/2+1 to n are child nodes and we no need to check for the maxHeap property.

Java implementation for **maxHeapify**:

```java
private static void maxHeapify(int nums[], int i, int heapSize) {
  int left = leftAt(i);
  int right = rightAt(i);
  int maxElementAt = i;
  if (left <= heapSize && nums[left] > nums[maxElementAt]) {
    maxElementAt = left;
  }
  if (right <= heapSize && nums[right] > nums[maxElementAt]) {
    maxElementAt = right;
  }
```

```
    if (maxElementAt != i) {
      swap(nums, i, maxElementAt);
      maxHeapify(nums, maxElementAt, heapSize);
    }
  }
```

Java implementation for **buildHeap**:

```
  private static void buildMaxHeap(int nums[]) {
    int parentAt = parentAt(nums.length);
    for (int i = parentAt; i >= 0; i--) {
      maxHeapify(nums, i, nums.length - 1);
    }
  }
```

Now, performing the heap sort is simple,

1. buildHeap – this will bring the maximum element at top(1st element in array)
2. Swap element at 1st position with nth position – this will move maximum element to its right position.
3. The above step might violate the max heap property, so call maxHeapify for 1st element.
4. And perform steps 2 to 3 recursively and we are done and all of the elements are now sorted in ascending order.

Java implementation for **heapSort**:

```
  public static void sort(int nums[]) {
    buildMaxHeap(nums);
    int heapSize = nums.length - 1;
    while (heapSize > 0) {
      swap(nums, heapSize, 0);
      --heapSize;
      maxHeapify(nums, 0, heapSize);
    }
  }
```

Summary:

- The time complexity of this heap sort is O(n log n)
- It takes constant O(1) auxiliary extra memory.
- And it is in place.

The full java code for this heap sorting is attached below.

```
click here to see full HeapSort.java
```

By Sada Kurapati • Posted in Algorithm, Sorting • Tagged algorithm, algorithms, data structure, data structures, heap, heap sort, java, sorting
AUG **20** 2013

Algorithms – Merge sorting

Today, let us learn another algorithm which has better time complexity [O(n log n)] than Insert sort [$O(n^2)$] and also follows completely different sorting technique. Let us introduce ourselves to one of the great algorithm paradigm, Divide and Conquer.

'Merge sort' closely follows this algorithm technique. Before we examine the logic behind this sort, let us look into the detailed aspects of this new technique. In this Divide and Conquer technique, we basically perform following steps.

- **Divide** the problem into number of small and similar problems.
- **Conquer** all of the sub problems by solving them recursively. If we get to the state we reach the base case or the case where we can solve this sub problem easily, then solve them directly.
- **Combine** the solution of these sub problems and prepare the solution to the actual/parent problem.

So how does 'merge sort' use this paradigm and sort the element?

Divide: Divide the given array of elements into two sub arrays at index n/2 (mid) – n is the number of elements in the given problem. You might be wondering what happens when n is odd number? In this case, we take mid as floor of n/2. For example, if we get array with 5 elements, then we divide them into [1 – 2] and [3 – 5] element arrays as FLOOR [n/2] = 2.

Conquer: Solve these sub arrays recursively.

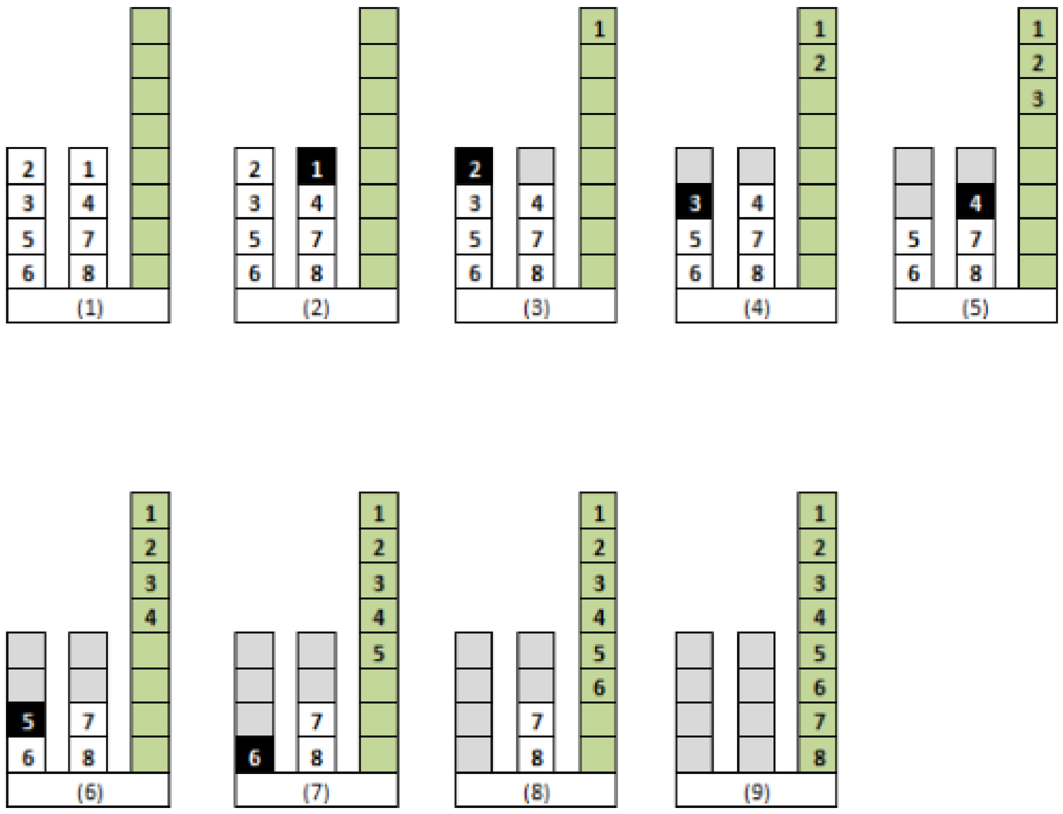
Combine: Merge the sorted sub arrays to get the combined sorted array which we are looking for.

As part of the merge sort, we divide the array of element till we get to the point where the divided sub arrays have only 1 element. As we know the array with 1 element is already sorted (no need to perform anything in "Conquer" step), then we simply merge those **in a way** so that the final merged result is in sorted order. So in this merge sort, the important and critical element is the method to merge two sorted sub arrays.

How do we merge two sorted arrays?

1. Compare the top/first element in two sorted arrays
2. Pop out that small element (assuming we are sorting non decreasing/ascending order) and put it in another array and repeat the Step 1 and 2 till any of the sorted sub array elements are exhausted.
3. Then copy the remaining elements from non-exhausted sub array.

To explain this further in detail, let us consider an example below.



?	- ordered sub arrays to be merged - input
■	- the small element of both input sub arrays - S1 & S2
▨	- element is removed, empty position (cell)
🟩	- array with result of merging S1 & S2 sub arrays

(https://sadakurapati.files.wordpress.com/2013/08/merge-sort_bars.png)

Step (1): Shows the two ordered sub arrays and the empty result (merged) array.

Step (2): find the minimum from top of the element of two sub arrays. **1**

Step (3): pop that minimum element [1] out and insert it into the result array and then find the minimum of the two sub arrays. **2**

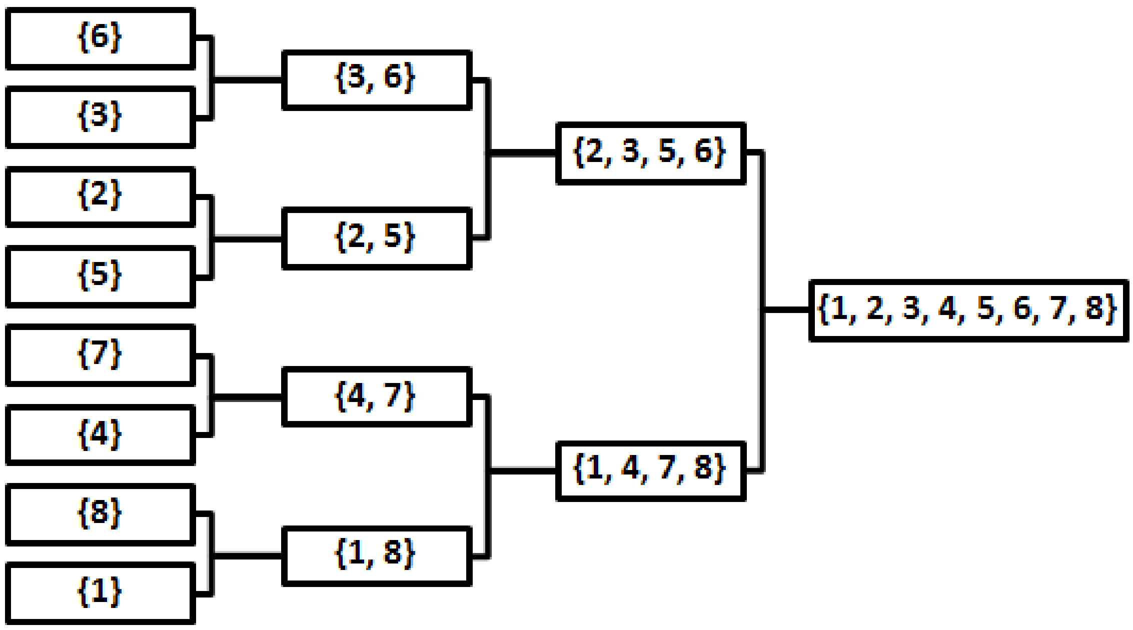
and so on……

Step (7): Find the minimum **6** and then move it to result array.

Step (8): One of the ordered sub arrays is done (all elements are moved and it is empty), so now copy the elements from non-exhausted array to the result array in the same order.

Step (9): The merging is done and the elements in the result array are all in sorted order.

The best way to understand this merge (sort) is taking an example and representing all of the steps in hierarchy as it is done recursively. Following diagram explains the merge sort for the input array **A1 = {6, 3, 2, 5, 7, 4, 8, 1}**.



(https://sadakurapati.files.wordpress.com/2013/08/merge-sort_hirarchy.png)

This is the depiction of all of the steps in performing a merge from left to right.

So once we have merge logic, then doing merge sort is very simple.

- Check if the left index is less than the right index of the array. If not, then this is one element array so no need to perform anything as it is already sorted
- If the left index is not less than the right index, then divide the array into half (mid = Floor[n/2]).
- Call merge sort for A[left…mid]
- Call merge sort for A[mid+1…right]
- Merge both the above sorted arrays – A[left..mid] and A[mid+1…right] using an auxiliary array.

Following is the java code for both the merge sort and merge methods.

MergeSort:

```java
//I sort the array using merge sort technique.
public static void sort(int[] nums, int left, int right) {
  if (left < right) {
    //Split in half
    int mid = (left + right) / 2;
    //Sort recursively.
    sort(nums, left, mid);
    sort(nums, mid + 1, right);

    //Merge the two sorted sub arrays.
    merge(nums, left, mid, right);
  }
}
```

Following is the logic for merge method:

```
  private static void merge(int[] nums, int left, int mid, int right) {
    // nums[left…mid] and A[mid+1…right] are the two sorted sub arrays t
    int size = right - left + 1;
    int temp[] = new int[size];
    //Copy the array into temp so that we can replace merged result into
    //This is an extra space and time overhead in this merge sorting.
    for (int i = 0; i < size; i++) {
      temp[i] = nums[left + i];
    }
    //Changed positions in temp array.
    mid = mid - left;
    right = right - left;
    int k = left;
    int i = 0; // new left is (left - left) which is 0;
    int j = mid + 1;
    while (i <= mid && j <= right) {
      if (temp[i] <= temp[j]) {
        nums[k] = temp[i];
        i++;
      } else {
        nums[k] = temp[j];
        j++;
      }
      k++;
    }

    if (i > mid) {
      //copy remaining elements from right
      for (; j <= right; j++, k++) {
        nums[k] = temp[j];
      }
    } else {
      //copy remaining elements from left
      for (; i <= mid; i++, k++) {
        nums[k] = temp[i];
      }
    }
  }
```

We call the merge sort as follows.

```
  public static void main(String args[]) {
    int input[];
    input = new int[]{6, 3, 2, 5, 7, 4, 8, 1};
    sort(input, 0, input.length - 1);
  }
```

The problem is divided into half and this process takes O(log n) time. The merge method takes O(n) linear time as we are simply merging the two sorted arrays. So the total asymptotic time complexity of this algorithm is O(n log n) which is better than Insertion sort which we discussed in earlier post.

Summary:

- The time complexity of this merge sort is O(n log n)
- It takes O(n) auxiliary extra memory.

- Note that it will take the same O(n log n) time even in the case of sorted inputs whereas Insertion performs faster in this case.
- This is a stable algorithm which means it will maintain the index order for the elements with same value.

The full java code for this merge sorting is attached below.

```
click here to see full MergeSort.java
```

By Sada Kurapati • Posted in Algorithm, Sorting, Trees • Tagged algorithms, data structures, divide and conquer, java, java enterprise edition, merge sort, sorting
AUG **17** 2013

Algorithms – Insertion sorting

Initially I always wondered why does every algorithm book, tutorial or class start with this insertion sort, but later on after reading through few more algorithms, I realized that this is one of the simplest and efficient (for small inputs) algorithm of all sorting algorithms exist today.

Almost all of us know what exactly means by sorting, we typically get an input of n numbers (assumed numbers for simplicity) and then we arrange them in the order of ascending (or descending) of their value. Now you might be wondering what happens if both the values are same. There are two options in this case, one we give the priority to the index of those elements in the input array and second we don't care. By the way this property is called Stable attribute of the algorithms.

So basically, if we maintain the order of index from the input when the elements are same, then those algorithms are called Stable. You might be wondering how does it matter when they are same. To explain this, we need to look into the aspect of satellite data. For example, think that we have a list of applications which we are trying to arrange using the applicant names (in alphabetical order). So when we come across two applicants with the same name, we would like to give the priority to the student who applied first and if we do so, then this is called Stable. So here, the name is the key on which we are sorting and all of the information from rest of the application like address, education, etc., is called the satellite data.

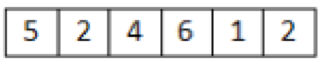
We have gone through enough on the basics of sorting algorithms, so let us jump into the insertion sort technique. After going through multiple books and tutorials on this, I think the simplest explanation to this is the way many people sort hand of playing cards.

Let us assume that we have a pile of playing cards on the desk with face down and we don't know about the rank (number) of those. To sort them, what we do is take the first card from the pile of cards and place that in the left hand at right position. Assume that we have 1, 4, 6, 8, 10 in the left hand. **Take a new card** from pile of face down cards, and if we get a 5, then we **move all of the cards greater than** 5 (6, 8, 10) to the right and then **insert that card** exactly between 4 and 6 at 3rd position.

So on high level, we perform following steps:

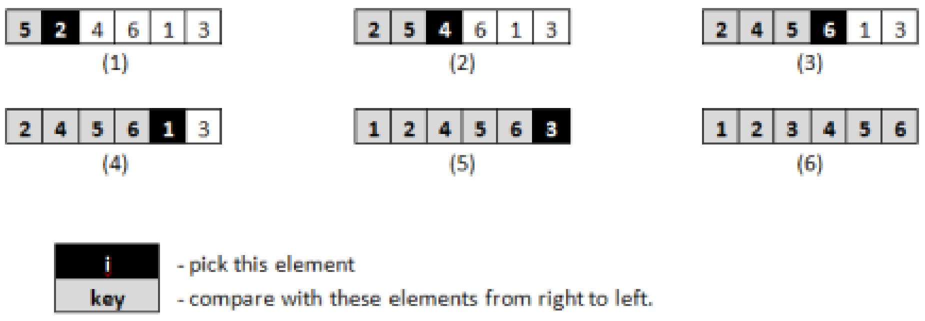
1. Take an element from unsorted array (pile of cards)
2. Compare with the elements in already sorted array on left side (in left hand) and move all of the elements which are greater than the picked element (card in right hand).
3. Then insert the picked up element in right position.

To illustrate this technique, let us go through the following example of elements and try to sort them.



(https://sadakurapati.files.wordpress.com/2013/08/insertion_sort_input.png)

First we partition the array into two at the element which we pick up. Just assume that we put some sentinel card between the left (hand) and right array (pile of cards) at the position of the picked up element.



(https://sadakurapati.files.wordpress.com/2013/08/insertion_sort.png)

The above diagram clearly explains how the numbers are sorted.

Iteration 1:

- Pick up the element at index 2 because the array with one element is already started. So in our example the number at 2nd index is 2.
- Compare with 5. As 5 >2, move 5 to right (index 2).
- Insert 2 at index 1.

Iteration 2:

- Pick up next element 4
- Compare with 5.As 5 > 4. Move the 5 to right, then compare 4 with 2 and stop here 2 < 4.
- Insert 4 at index 2.

Hope now you can continue the rest of the iterations.

Following is the java implementation of this insertion sorting technique.

insertionSort method

```
    //I sort the given numbers in ascending order using Insertion sort a
    public static void sort(int nums[]) {
```

```
    //Start with the 2nd position as the partition with 1 elements is 
    for (int i = 1; i < nums.length; i++) {
      //Take the card in hand
      int key = nums[i];
      //Move the elements to right till we find the correct position f
      //in already sorted array (loop invariant) nums[1.. j-1]
      int j = i - 1;
      //We can also right this in single line for (; j >= 0 && nums[j]
      for (; j >= 0 && nums[j] > key; j--) {
        nums[j + 1] = nums[j];
      }
      /**
       * Insert the key[card in hand] at the position. You might be wo
       * why we are inserting at nums[j+1] rather than nums[j]. Think 
       * be the j value when we wanted to insert smallest element. it 
       * and hope now you got me why we are inserting it at nums[j+1].
       */
      nums[j + 1] = key;
    }
  }
```

In the above code, the first loop

```
for (int i = 1; i &lt; nums.length; i++) {
```

will iterate through all of the elements in the array (pile of face down cards)starting at the position. In java, the arrays are zero based indexed so this loop is iterating from 1 to < length of the array. The variable key will hold the card which we picked up. The second loop

```
for (; j &gt;= 0 &amp;&amp; nums[j] &gt; key; j--) {
```

will compare all of the elements on the left side stating at its nearest neighbor element. That is it iterates from [i-1] to [0] and move all of the elements which are greater than the key nums[I] one position to the right.

Then the statement

```
nums[j + 1] = key;
```

will insert the key.

Summary:

- The time complexity of this insertion sort is O(n^2)
- It take only O(1) extra space and it is in place sorting method.
- The best case time complexity of this algorithm is linear – O(n). This is because lesser swaps and the 2nd loop execution is minimal.
- This is a stable algorithm which means it will maintain the index order for the elements with same value.

Following is the full source code for insertion sort in java language.

```
click here to see full InsertionSort.java
```

By Sada Kurapati • Posted in Algorithm, Sorting • Tagged algorithms, insertion sort, java, sorting