Sorting Algorithms

Arden Rasmussen

DATE

Contents

Abstract

This article will organize an extensive list of sorting algorithms, in order to compare their efficiency in different situations. The different sorting algorithms will be compared based on the run time for the algorithm, the number of comparisons required for the algorithm, and the number of array access required for the algorithm. These properties of each algorithm will be determined by averaging runs for a multitude of arrays of the same length, to achieve an average value.

1 Comparison Sorting Algorithms

Comparison sort algorithms are algorithms whose method of sorting the items is by comparing each item to another item from the array, depending on this comparison the items are swapped. This is some of the most common method of sorting algorithms, because it is simple to understand, and think of.

1.1 Binary Tree Sort

Best-Case Performance	$O(n \log n)$
Average Performance	$O(n \log n)$
Worst-Case Performance	$O(n^2)$
Worst-Case Space Complexity	$O(n)$

Binary tree sort is an algorithm that utilizes the binary tree structure in order to sort the items into the correct order. Figure 1.1 is an example of a binary tree structure, where each node has up to two child nodes. Using this structure a binary tree can be created to sort an unsorted array of elements. Through the process of creating a binary tree the elements are sorted as they are inserted into the tree. A simple recursive algorithm can be used to insert elements into the binary tree. Once all elements have been inserted into the binary tree to retrieve the sorted array, the elements of the binary array must be read from left to right.

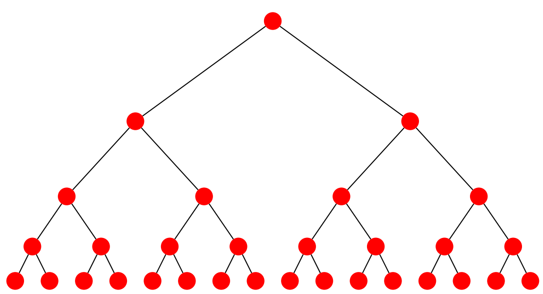

Figure 1: Example of Binary Tree structure

The process of adding one item to a binary search tree is on average $O(\log n)$ so adding n items to the tree will average to $O(n \log n)$. However, adding an item to an unbalanced tree needs $O(n)$ time, meaning that a worst case for adding n items to a tree will be $O(n^2)$. This worst case performance occurs when the algorithm is acting on an already sorted array.

Pseudocode for the Binary Tree algorithm is shown below in Algorithm 1. The core of the algorithm is a object ($root$) that contains a left and right sub-copy of itself ($leftTree$, $rightTree$). The function $BinaryTree$ takes in an array of items to be sorted, it them adds each one to the binary tree ($root$), and then when the tree is completed it can be read, which in turn returns the sorted array. The function $Insert$ takes a binary tree ($tree$) to add a item ($value$) to. First this function checks to see if the binary tree is empty, and if so, it creates the root with the item to be inserted. If the binary tree is not empty, then the function determines if the item belongs in the left sub-tree, or the right sub-tree. If the item is less than the item in the root position of the binary tree, then $Insert$ function is run with the left sub-tree as the new root, and otherwise the function is run with the right sub-tree as the new root. The $ReadTree$ function traverses the binary tree from left most to right most node, by recursively entering the left branch if it exists, then adding the current node to the array, then recursively entering the right branch if it exists.

Source code examples of Binary Tree sort can be found in A.1.

Algorithm 1 Binary Tree Pseudocode

```
function BinaryTree(a)
    root ← null
    for i = 0 to n do
        Insert(root,a[i])
    end for
    a ←ReadTree(root, a)
end function
function Insert(tree, value)
    if tree = null then
        tree.value ← value
        tree.leftTree ← null
        tree.rightTree ← null
    else if tree.value ≤ value then
        Insert(tree.leftTree, value)
    else
        Insert(tree.rightTree, value)
    end if
    return tree
end function
function ReadTree(tree, a)
    if tree ≠ null then
        ReadTree(tree.leftTree, a)
        a append tree.value
        ReadTree(tree.rightTree, a)
    end if
    return a
end function
```

1.2 Bubble Sort

Best-Case Performance	$O(n)$
Average Performance	$O(n^2)$
Worst-Case Performance	$O(n^2)$
Worst-Case Space Complexity	$O(1)$

Bubble sort is an algorithm that goes term by term, sorting pairs of two adjacent elements. Although the algorithm is extremely simplistic it is not very effective for arrays that are not mostly sorted already. The algorithm repeatedly steps through the list of items, and it compares each pair of adjacent elements, and swaps the two elements if they are not in order. This is repeated until no swaps are necessary. This algorithm is extremely simple, as it only considers two elements at a time.

Bubble sorts complexity is a worst case of $O(n^2)$, because each element must be slowly moved into place one step at a time. This means that the algorithm takes increasingly longer to run as the number of elements to sort grows. However, bubble sort does have an advantage over more complicated sorting algorithms. This is when the list is already in sorted order. Because the algorithm contains a simple check to see if the list is sorted, when the input is already sorted bubble sort can run with a best case of $O(n)$, because it just needs to loop through the elements once to determine if it is sorted. Other more complex sorting algorithms such as Quick sort (**??**) are unable to do this, and so cannot run with this efficiency.

The cayuse for bubble sorts slow worst-case run time is because of what are commonly called "turtles". Turtles are small elements that begin near the back of the list. Because these items can only move towards that begin once per loop, then it the smallest item is at the end of the list, it will take $n-1$ passes in order to move it all the way to the beginning of the list. In opposition to turtles, there are "rabbits". Rabbits are large elements, because the largest element will always be swapped back, it means that no matter where in the list it begins, it will be in place after the first pass.

Pseudocode for the Bubble Sort algorithm is shown below in Algorithm 2. The function gets a value (*length*) for the length of the array of elements. While this value is not equal to zero, the algorithm loops through the array swapping any elements that need to be swapped. In each loops, the algorithm creates a new length value (*newLength*). This new length is set to the position in the list where the maximum swap occurs, because due to the nature of the algorithm, once there is no swapping in the upper bounds, it indicates that the upper part of the array has been sorted, and there is no need to loop through those elements unnecessarily.

There are many variation to bubble sort that attempt to improve the run time for the algorithm, such as Cocktail Sort (**??**), or Comb Sort (**??**). Source code examples of Bubble sort can be found in A.2.

Algorithm 2 Bubble Sort Pseudocode

```
function Bubble(a)
    length ←length(a)
    while length ≠ 0 do
        newLength ← 0
        for i = 1 to length − 1 do
            if a[i − 1] > a[i] then
                Swap(a[i − 1],a[i])
                newLength ← i
            end if
        end for
        length ← newLength
    end while
    return a
end function
```

Appendices

A Algorithms

A.1 Binary Tree Sort

A.1.1 C++

```
#ifndef BINARY_TREE_HPP
#define BINARY_TREE_HPP
namespace sort{
  struct BinaryTree{
    int value;
    BinaryTree* left;
    BinaryTree* right;
  };
  void BinaryTreeSort();
  void InsertNode(BinaryTree* &tree, int new_value);
  void ReadTree(BinaryTree* node);
}
#endif
```

../algorithms/binary_tree/binary_tree.hpp

```
#include "binary_tree.hpp"
#include <time.h>
#include <vector>
#include "../algo_core.hpp"

void sort::BinaryTreeSort() {
  clock_t start = clock();
  BinaryTree* root = NULL;
  for (int i = 0; i < data.size(); i++) {
    result.vec_access++;
    InsertNode(root, data[i]);
  }
  data.clear();
  ReadTree(root);
  result.time_elapsed = (double)(clock() - start) / CLOCKS_PER_SEC;
}

void sort::InsertNode(BinaryTree*& tree, int new_value) {
  if (tree == NULL) {
    tree = new BinaryTree;
    tree->value = new_value;
    tree->right = NULL;
    tree->left = NULL;
  } else if (tree != NULL) {
    result.comparisons++;
    if (new_value <= tree->value) {
      InsertNode(tree->left, new_value);
    } else {
      InsertNode(tree->right, new_value);
    }
  }
```

```
}

void sort::ReadTree(BinaryTree* node) {
  if (node != NULL) {
    ReadTree(node->left);
    result.vec_access++;
    data.push_back(node->value);
    ReadTree(node->right);
  }
}
```

../algorithms/binary_tree/binary_tree.cpp

A.2 Bubble Sort

A.2.1 C++

```cpp
#ifndef BUBBLE
#define BUBBLE
namespace sort{
  void BubbleSort();
}
#endif
```

../algorithms/bubble/bubble.hpp

```cpp
#include "bubble.hpp"
#include <time.h>
#include <vector>
#include "../algo_core.hpp"

void sort::BubbleSort() {
  clock_t start = clock();
  int length = data.size();
  while (length != 0) {
    for (int i = 1; i < length; i++) {
      result.comparisons++;
      result.vec_access += 2;
      if (data[i - 1] > data[i]) {
        result.vec_access += 2;
        iter_swap(data.begin() + i - 1, data.begin() + i);
      }
    }
    length--;
  }
  result.time_elapsed = (double)(clock() - start) / CLOCKS_PER_SEC;
}
```

../algorithms/bubble/bubble.cpp