Comparison of Sorting Algorithms

Arden Rasmussen

19-11-2016

Contents

Abstract

This project aims to compare and contrast the some commonly used sorting algorithms. The different sorting algorithms that will be tested will be tested on value range, and value size, in order to determine the best and worst case scenarios for each algorithm.

1 Introduction

1.1 Terms

This is a set of standard terms and variables that will be referenced to for explanations of the sorting algorithms, and their methods.

n : The number of values that will be sorted by the sorting algorithms.

r : The range for n that will be sorted.

$Values$: The stored values to be sorted, of n length, and in r range.

t : The time in seconds that an algorithm takes to sort n values.

(i) Reference to specific line in referenced code example.

2 Algorithms

Here is a description of the algorithms that were a part of this research. In this section each algorithm will be described, and the method of its sorting will be explained in words, as well as with persudo code.

2.1 Bubble Sort

Bubble sort works off of a method where the algorithm only focus on a two terms of the list at a time. Bubble sort compares two adjacent pairs of numbers, and switches them to be in the correct order. The algorithm moves through the list until it reaches the end. However, this does not mean that the list is sorted, so the algorithm returns to the beginning and repeats the loop switching items when necessary until there are no more items that need to be switched.

This method can then be optimized through several methods. One optimization method that is implemented in this example is setting a limit to the maximum on the loop. If there are no more values that need to be swapped, then the algorithm does not need to continue to check those values. So simply the algorithm can track a limit to the maximum value that needs to be checked, and instead of looping through the entire list of values, the algorithm can loop through a select number of the values from the lists.

Here is an example of the persudo code, followed by a line by line explanation of the code. A example of bubble sort in actual code can be found in **??**.

BubbleSort$Values$ n =length($Values$) $n \neq 0$ $newn = 0$ $i = 1$ to $n - 1$ $Values[i-1] > Values[i]$ Swap($Values[i-1]$, $Values[i]$) $newn = i$ $n = newn$

(1) The definition of the function with an input of $Values$ which is a list of values to be sorted.

(2) The beginning of a loop the follow until $Values$ is completely sorted.

(3) The definition of $newn$ variable which will be used to determine the maximum limit on the sorting loop.

(4) The for loop that begins at the first two items in $Values$ and ends at the $n-1$ item in $Values$.

(5) The if check to determine if the two items need to be switched.

(6) Swaps the two compared items positions.

(7) Sets $newn$ to i in order to move the upper limit up to the current maximum necessary swap position.

(10) Sets n to $newn$ essentially setting the upper limit on the for loop.

Bubble sort is considered a very poor method for sorting algorithms, and is most often less efficient than other extremely simple sorting algorithms. Bubble sort has a complexity of $O(n^2)$. This complexity means that for an increase in the n, the complexity of the algorithm, and the time requirement grow exponentially. For small n values this algorithm would work sufficiently, but as the number of values grows this algorithm becomes much less efficient, and soon becomes useless.

2.2 Bucket Sort

Bucket sort works with a method of dividing the set of values into smaller and smaller sub groups. The algorithm creates a set of "buckets" that it then sorts the values into. This method can be simplified as sorting groups at a time. For example any values that are in the first 10% will be placed in the first bucket, then values in the second 10% will be placed in the second bucket. This continues until all values have been placed into a bucket. Then the algorithm is run again with each of the bucket that were just created as the main bucket. Each of the terms that was placed in the first bucket are then again sorted into ten more buckets. This is done until all values are in their own bucket. Then the buckets are placed back together in order. This results in the sorted list of values.

Bucket sort can as well be optimized. Bucket sort is very effective at breaking up the large data set into smaller semi-sorted sets of data, but at a point bucket sort becomes inefficient. Because going through algorithm until every value has its own bucket requires much more calculations than some optimized algorithms. One such optimization is to use bucket sort, then once the number of elements in a bucket is below a specified level, then that bucket is sorted using some other sorting algorithm. Or another option is to place all the buckets back into a single semi-sorted list, and then use *Insertion Sort* to complete the sorting, because *Insertion Sort*'s runtime is based off of an elements distance from its final position.

Here is bucket code in persudo code, accompanied with a line by line explication. Real examples of code can be found in **??**.

BucketSort$Values$, n $Arrays$ $<-$ new array of n empty lists $i = 0$ to (length($Values$)-1) insert $Values[i]$ into $Arrays$[floor($Values[i]$/(rangeof($Values$)/n))] clear $Values$ $i = 0$ to $n - 1$ append BucketSort$Arrays[i]$, n to $Values$ $Values$

(1) The definition of the function with an input of $Values$ which is a list of values to be sorted, and n which is the number of buckets to sort into.

(2) Creates a new list of n empty list or "buckets".

(3) Beginning of a for loop that goes through the length of $Values$ and sorts them into the $Arrays$.

(4) The method used for determining what $Array$ to place the item into is done by dividing the value of the item by the value range of the array, then rounding down. This will result in returning the correct $Array$ for the item to be sorted into.

(6) Clears $Values$ of any items to prepare it for rewriting with sorted results.

(7) Beginning of a for loop that goes through every list in $Arrays$

(8) Recursively calls bucket sort with each new list of items that was placed in each bucket. Eventually there will only be a single item in a bucket and it cannot be sorted. The returned results will be appended together into a complete and sorted list of values.

(10) This sorted list of values is then returned for any functions higher up in the chain of being called in the recursive function.

Bucket sort is a moderately effective sorting algorithm, but could be greatly improved with some simple optimizations. On its own Bucket sort has a complexity of $O(n)$, as there is a need for a bucket for every element that is being sorted. This means that it is fairly efficient, but still could be greatly improved by the implementation of a limit for the minimum size of bucket before initiation a different sorting algorithm.

2.3 Counting Sort

Counting Sort uses a method where the algorithm counts the number of elements with the same value. Counting sort goes through all the elements in the list, and creates counters for each value in the element list range. The algorithm then goes through the list and adds to each respective counter for each element. Once all the counters are created, then the final list of elements is created by adding each counter to the list. This means that this algorithm is extremely memory efficient with data sets that have a small range, as the algorithm only needs to store a list of range number of integers. However when the data set has a wide range, then this algorithm becomes much less efficient, as there will be many unused counters, and will require storing many more values. This algorithm is very efficient for time, but for memory the algorithms complexity quickly grows, causing it to be unmanageable for data sets with a wide range of values.

This algorithm can be optimized by reducing the number of for loops that are necessary, and limiting the counters to a lower and upper bound. Because if the values in a list begin until sever hundred, there is no reason to store counters for everything up until then.

Here is bucket code in persudo code, accompanied with a line by line explication. Real examples of code can be found in **??**.

CountingSort$Values$ $Counters$ $<-$ an array of integer values min $<-$ integer for smallest value in $Values$ max $<-$ integer for largest value in $Values$ $i = 0$ to length($Values$) - 1 $Values[i] > max$ $max = Values[i]$ $Values[i] < min$ $min = Values[i]$ $i = min$ to max append 0 to $Counters$ $i = 0$ to length($Values$) - 1 $Counters[values[i] - min] + 1$ Clear $Values$ $i = 0$ to length($Counters$) - 1 $j = 0$ to $Counters[i]$ append $(i + min)$ to $Values$

(1) The definition of the function with an input of $Values$ which is a list of values to be sorted.

(5) For loop used to determine the minimum and maximum values in $Values$.

(13) For loop to create counters for each value from min to max

(16) For loop that runs through all elements in $Values$ and adds to each respective counter for each value.

(19) Clears $Values$ of any items to prepare it for rewriting with sorted results.

(20) Beginning of a for loop that goes through every counter in $Counters$

(21) For loop that adds the counted number of each element back into $Values$ based off of $Counters$

Counting sort is extremely efficient in terms of time, as there is no actual comparison between the different elements in the list. This means that counting sort runs fairly quickly, but the memory necessary grows with the range of elements. For a list of elements that has a small range, counting sort will be extremely effective, but for a list of elements with a large range of elements, counting sort will require much more memory to run. The complexity of counting sort is $O(n + k)$, where k is the range of values. This is true for the best and worst case scenarios for counting sort, making it an extremely predictable sorting algorithm.

2.4 Cube Sort

2.5 Heap Sort

2.6 Insertion Sort

Insertion sort is a very common sorting algorithm, as it is very simple, and still effective. The method of insertion sort is to select the first value in the lsit, and to then check it against every value after it in the list. If the selected value is greater than the other value in the comparison, then the two values are swiched, otherwise the loop stops.

2.7 Merge Sort

2.8 Quick Sort

2.9 Radix Sort

2.10 Selection Sort

2.11 Shell Sort

2.12 Tim Sort

2.13 Tree Sort

Appendices

A Algorithms

A.1 Bubble

A.1.1 C++

```cpp
#include "../../induco.h"
#include "../../sort_headers.h"
#include <cmath>
#include <ctime>
#include <iostream>
#include <vector>

double sort::BubbleSort(bool display) {
  if (display == true) {
    std::cout << "Bubble Sort:\n";
  }
  double timeelapsed;
  induco::Timer(true);
  int n = values.size();
  while (n != 0) {
    int newn = 0;
    for (int i = 1; i < n; i++) {
      if (values[i - 1] > values[i]) {
        iter_swap(values.begin() + i - 1, values.begin() + i);
        newn = i;
      }
    }
    n = newn;
    if (display == true) {
      induco::DrawLoadingBar((values.size() - n) / (float)values.
    size(), 50);
    }
  }
  timeelapsed = induco::Timer();
  if (display == true) {
    induco::DrawLoadingBar(1, 50);
    std::cout << "\nSorted " << values.size() << "\n";
    std::cout << induco::DisplayTime(timeelapsed, true) << "\n";
  }
  return (timeelapsed);
}
```

algorithms/bubble/bubble_sort.cpp

A.2 Bucket

A.2.1 C++

```cpp
#include "../../induco.h"
#include "../../sort_headers.h"
#include <cmath>
#include <ctime>
#include <iostream>
#include <vector>

namespace sort {
int Btotalrecursioncount = 0;
}

double sort::BucketSort(int bucketcount, bool display) {
  if (display == true) {
    std::cout << "Bucket Sort:\n";
  }
  Btotalrecursioncount = 0;
  double timeelapsed;
  induco::Timer(true);
  if (display == true) {
    induco::DrawLoadingBar(0, 50);
  }
  BRecursion(bucketcount, &values, display, true);
  timeelapsed = induco::Timer();
  if (display == true) {
    induco::DrawLoadingBar(1, 50);
    std::cout << "\nSorted " << values.size() << "\n";
    std::cout << induco::DisplayTime(timeelapsed, true) << "\n";
  }
  return (timeelapsed);
}

void sort::BRecursion(int bucketcount, std::vector<int> *bucket,
    bool display,
                      bool top) {
  std::vector<std::vector<int>> buckets;
  int max = 0, min = 0, bucketrange;
  for (int i = 0; i < bucketcount; i++) {
    std::vector<int> newbucket;
    buckets.push_back(newbucket);
  }
  max = bucket->at(0);
  min = bucket->at(0);
  for (int i = 1; i < bucket->size(); i++) {
    int value = bucket->at(i);
    if (value > max) {
      max = value;
    }
    if (value < min) {
      min = value;
    }
  }
  bucketrange = ceil((max - min) / (double)bucketcount);
  for (int i = 0; i < bucket->size(); i++) {
```

```
53      int value = bucket->at(i);
54      if (value == max) {
55        buckets[buckets.size() - 1].push_back(value);
56      } else {
57        buckets[BBucket(value - min, bucketrange)].push_back(value);
58      }
59    }
60    bucket->clear();
61    for (int i = 0; i < buckets.size(); i++) {
62      if (buckets[i].size() <= 1) {
63        bucket->insert(bucket->end(), buckets[i].begin(), buckets[i].
      end());
64      } else if (BDupes(buckets[i]) == true) {
65        Btotalrecursioncount += buckets[i].size();
66        bucket->insert(bucket->end(), buckets[i].begin(), buckets[i].
      end());
67      } else {
68        BRecursion(bucketcount, &buckets[i], display);
69        bucket->insert(bucket->end(), buckets[i].begin(), buckets[i].
      end());
70      }
71      if (display == true && top == true) {
72        induco::DrawLoadingBar(bucket->size() / (double)
      scrambledvalues.size(),
73                               50);
74      }
75    }
76  }
77
78  int sort::BBucket(int value, int bucketsize) {
79    return ((floor(value / (double)bucketsize)));
80  }
81
82  bool sort::BDupes(std::vector<int> bucket) {
83    if (bucket.size() <= 1) {
84      return (true);
85    }
86    int value = bucket[0];
87    for (int i = 1; i < bucket.size(); i++) {
88      if (bucket[i] != value) {
89        return (false);
90      }
91    }
92    return (true);
93  }
```

algorithms/bucket/bucket_sort.cpp

A.3 Counting

A.3.1 C++

```cpp
#include "../../induco.h"
#include "../../sort_headers.h"
#include <cmath>
#include <ctime>
#include <iostream>
#include <vector>

double sort::CountingSort(bool display) {
  if (display == true) {
    std::cout << "Counting Sort:\n";
  }
  double timeelapsed;
  induco::Timer(true);
  std::vector<int> counters;
  int max = values[0], min = values[0];
  for (int i = 1; i < values.size(); i++) {
    if (values[i] > max) {
      max = values[i];
    }
    if (values[i] < min) {
      min = values[i];
    }
  }
  for (int i = min; i <= max; i++) {
    counters.push_back(0);
  }
  for (int i = 0; i < values.size(); i++) {
    counters[values[i] - min]++;
    if (display == true) {
      induco::DrawLoadingBar(0.5 * (i / (double)values.size()), 50)
    ;
    }
  }
  values.clear();
  for (int i = 0; i <= max; i++) {
    if (display == true) {
      induco::DrawLoadingBar((0.5 * (i / (double)max)) + 0.5, 50);
    }
    for (int j = 0; j < counters[i]; j++) {
      values.push_back(i + min);
    }
  }
  timeelapsed = induco::Timer();
  if (display == true) {
    induco::DrawLoadingBar(1, 50);
    std::cout << "\nSorted " << values.size() << "\n";
    std::cout << induco::DisplayTime(timeelapsed, true) << "\n";
  }
  return (timeelapsed);
}
```

algorithms/counting/counting_sort.cpp

A.4 Insertion

A.4.1 C++

```cpp
#include "../../induco.h"
#include "../../sort_headers.h"
#include <cmath>
#include <ctime>
#include <iostream>
#include <vector>

double sort::InsertionSort(bool display) {
  if (display == true) {
    std::cout << "Insertion Sort:\n";
  }
  double timeelapsed;
  induco::Timer(true);
  for (int i = 1; i < values.size(); i++) {
    int j = i;
    while (j > 0 && values[j - 1] > values[j]) {
      iter_swap(values.begin() + j, values.begin() + j - 1);
      j--;
    }
    if (display == true) {
      induco::DrawLoadingBar(i / (double)values.size(), 50);
    }
  }
  timeelapsed = induco::Timer();
  if (display == true) {
    induco::DrawLoadingBar(1, 50);
    std::cout << "\nSorted " << values.size() << "\n";
    std::cout << induco::DisplayTime(timeelapsed, true) << "\n";
  }
  return (timeelapsed);
}
```

algorithms/insertion/insertion_sort.cpp

A.5 Merge

A.5.1 C++

```cpp
#include "../../induco.h"
#include "../../sort_headers.h"
#include <cmath>
#include <ctime>
#include <iostream>
#include <vector>

namespace sort {
std::vector<int> Mworkvector;
}

double sort::MergeSort(bool display) {
  if (display == true) {
    std::cout << "Merge Sort:\n";
  }
  double timeelapsed;
  induco::Timer(true);
  Mworkvector = values;
  for (int i = 1; i < values.size(); i = 2 * i) {
    for (int j = 0; j < values.size(); j = j + 2 * i) {
      if (display == true) {
        induco::DrawLoadingBar(j / (double)values.size(), 50);
      }
      int vara, varb;
      if (j + i >= values.size()) {
        vara = values.size();
      } else {
        vara = j + i;
      }
      if (j + 2 * i >= values.size()) {
        varb = values.size();
      } else {
        varb = j + 2 * i;
      }
      MMerge(j, vara, varb);
    }
    values = Mworkvector;
  }
  timeelapsed = induco::Timer();
  if (display == true) {
    induco::DrawLoadingBar(1, 50);
    std::cout << "\nSorted " << values.size() << "\n";
    std::cout << induco::DisplayTime(timeelapsed, true) << "\n";
  }
  return (timeelapsed);
}

void sort::MMerge(int ileft, int iright, int iend) {
  int i = ileft;
  int j = iright;
  for (int k = ileft; k < iend; k++) {
    if (i < iright && (j >= iend || values[i] <= values[j])) {
      Mworkvector[k] = values[i];
```

```
54          i++;
55        } else {
56          Mworkvector[k] = values[j];
57          j++;
58        }
59      }
60  }
```

algorithms/merge/merge_sort.cpp

A.6 Quick

A.6.1 C++

```cpp
#include "../../induco.h"
#include "../../sort_headers.h"
#include <cmath>
#include <ctime>
#include <iostream>
#include <vector>

namespace sort {
int Qtotalrecursioncount = 0;
}

double sort::QuickSort(bool display) {
  if (display == true) {
    std::cout << "Quick Sort:\n";
  }
  Qtotalrecursioncount = 0;
  double timeelapsed;
  induco::Timer(true);
  QRecursive(0, values.size(), display);
  timeelapsed = induco::Timer();
  if (display == true) {
    induco::DrawLoadingBar(1, 50);
    std::cout << "\nSorted " << values.size() << "\n";
    std::cout << induco::DisplayTime(timeelapsed, true) << "\n";
  }
  return (timeelapsed);
}

void sort::QRecursive(int low, int high, bool display) {
  if (low < high) {
    int partition = QPartition(low, high);
    QRecursive(low, partition - 1, display);
    QRecursive(partition + 1, high, display);
  }
  if (display == true) {
    Qtotalrecursioncount++;
    induco::DrawLoadingBar(Qtotalrecursioncount / (double)(2 *
    values.size()),
                           50);
  }
}

int sort::QPartition(int low, int high) {
  int piviot = values[high];
  int i = low;
  for (int j = low; j < high; j++) {
    if (values[j] <= piviot) {
      iter_swap(values.begin() + i, values.begin() + j);
      i++;
    }
  }
  iter_swap(values.begin() + i, values.begin() + high);
  return (i);
```

```
53 }
```

algorithms/quick/quick_sort.cpp

A.7 Selection

A.7.1 C++

```cpp
#include "../../induco.h"
#include "../../sort_headers.h"
#include <cmath>
#include <ctime>
#include <iostream>
#include <vector>

double sort::SelectionSort(bool display) {
  if (display == true) {
    std::cout << "Seleciton Sort:\n";
  }
  double timeelapsed;
  induco::Timer(true);
  for (int i = 0; i < values.size() - 1; i++) {
    int smallestpointer = i;
    for (int j = i + 1; j < values.size(); j++) {
      if (values[j] < values[smallestpointer]) {
        smallestpointer = j;
      }
    }
    if (smallestpointer != i) {
      iter_swap(values.begin() + i, values.begin() +
    smallestpointer);
    }
    if (display == true) {
      induco::DrawLoadingBar(i / (double)values.size(), 50);
    }
  }
  timeelapsed = induco::Timer();
  if (display == true) {
    induco::DrawLoadingBar(1, 50);
    std::cout << "\nSorted " << values.size() << "\n";
    std::cout << induco::DisplayTime(timeelapsed, true) << "\n";
  }
  return (timeelapsed);
}
```

algorithms/selection/selection_sort.cpp