Contents

As far as I know all of my code works. I also included my implementations of merge sort and heap sort in my code. I wrote two versions of radix sort, one that uses 8 bit (byte) histograms and one that uses 16 bit (word) historgrams. The radix sort on the included plot is the one that uses 8 bit histograms, I would have included one for the 16 bit version, but as I write this I don't have time to re-run all the tests to generate the graph. The byte version performs 9 loops over the data plus a loop of length 256 over the histograms, while the word version performs 5 loops over the data plus a loop of length 65536 over the histograms.The performance of the two is what you would expect based on the loop count, for arrays up to about 100,000 elments the byte version is faster, due to having smaller constant factors (i.e the loop over the histograms), beyond that the word verson is faster. For arrays of a million or more the word version is consistantly about twice as fast as the byte version, which is to be expected. Both of my radix sorts sort 64 bits, regardless of the number of bytes used by the input, it wouldn't be that hard to optimize this so I skipped the later loops if the input used fewer than 64 bits, but I didn't do that (I guess because I'm lazy?).

The performances of the algorithms are mostly what you would expect. My version of quick sort is faster than the system quicksort by some factor of n, due to the fact the system quick sort has the added overhead of a function call for each comparision. Both quick sorts are slower than radix sort, though my quick sort is just barely slower, which is somewhat suprising, though I imagine you would see more of a difference if I used the word version of radix sort, or optimized my radix sort by the number of bytes in the input, as it is it runs at O(8*n) as it is now. The counting sort is the fastest since it runs at O(n) for the 16 bit integers, also my counting sort is inplace which elimnates some of the overhead of radix sort.

In my personal testing I found, for comparision based sorts on random data, that quicksort is usually the fastest, followed by merge sort, followed by heapsort. Heapsort is typically significantly slower, likely because I use insertion sort for sorting the small sublists in quick/merge sort.

For fun I also did a comparision of bubblesort and insertion sort, I've never actually used bubble sort before, and it really is as bad as it's reputation, insertion sort is typically about an order of magnitude faster.

```
#define NIL -1
int search(int *A, int n, int v){
  int i;
  for(i=0;i<n;i++){
    if(A[i] == v){
      return i;
    }
  }
  return NIL;
}
```

Loop Invariant: For all 0<=j<i A[j] != v;
Initialization: i = 0, so {j: 0<=j<i} is empty
Maintenance: If there exists some 0<=j<i such that A[j] = v the loop would have terminated when i was equal to j, thus A[j] != v for all j<=0<i Termination: At end of the loop i = n, so for all 0<=j<n A[j] != v, thus
v is not in A, so we can return NIL

```
Let h(n) = f(n) + g(n);
We can assume max(f(n),g(n)) = f(n), without a loss of generality.
In this case N such that n  N f(n)  g(n), or equivlemently f(n)  0.5*h(n).
Since h(n) = f(n) + g(n) we can also say 2*h(n)  f(n).
These two inequalities together imply N 2*h(n)  2*f(n)  0.5*h(n) n  N, meaning
f(n) = (h(n)), or max(f(n),g(n)) = (f(n)+g(n))
```