## Bubble Sort

for (int i = 0; i < data.Length; i++)

for (int j = 0; j < data.Length - 1; j++)

if (data[j] > data[j + 1])

{

tmp = data[j];

data[j] = data[j + 1];

data[j + 1] = tmp;

}

## Insertion Sort

for (int i = 0; i <= data.Length; i++) {

int j = i;

while (j > 0 && data[i] < data[j - 1])

j--;

int tmp = data[i];

for (int k = i; k > j; k--)

data[k] = data[k - 1];

data[j] = tmp;

}

## Merge Sort

A merge sort works recursively. First it divides a data set in half, and sorts each half separately. Next, the first elements from each of the two lists are compared. The lesser element is then removed from its list and added to the final result list.

int[] mergeSort (int[] data) {

if (data.Length == 1)

return data;

int middle = data.Length / 2;

int[] left = mergeSort(subArray(data, 0, middle - 1));

int[] right = mergeSort(subArray(data, middle, data.Length - 1));

int[] result = new int[data.Length];

int dPtr = 0;

int lPtr = 0;

int rPtr = 0;

while (dPtr < data.Length) {

if (lPtr == left.Length) {

result[dPtr] = right[rPtr];

rPtr++;

} else if (rPtr == right.Length) {

result[dPtr] = left[lPtr];

lPtr++;

} else if (left[lPtr] < right[rPtr]) {

result[dPtr] = left[lPtr];

lPtr++;

} else {

result[dPtr] = right[rPtr];

rPtr++;

}

dPtr++;

}

return result;

}

Each recursive call has O(n) runtime, and a total of O(log n) recursions are required, thus the runtime of this algorithm is O(n \* log n). A merge sort can also be modified for performance on lists that are nearly sorted to begin with. After sorting each half of the data, if the highest element in one list is less than the lowest element in the other half, then the merge step is unnecessary.

Apart from being fairly efficient, a merge sort has the advantage that it can be used to solve other problems, such as determining how "unsorted" a given list is.

int[] result = new int[data.Length];

for (int i = 0; i < data.Length; i++)

data[i] = h.RemoveLowest();

## Quick Sort

A quick sort, as the name implies, is intended to be an efficient sorting algorithm. The theory behind it is to sort a list in a way very similar to how a human might do it. First, divide the data into two groups of "high" values and "low" values. Then, recursively process the two halves. Finally, reassemble the now sorted list.

Array quickSort(Array data) {

if (Array.Length <= 1)

return;

middle = Array[Array.Length / 2];

Array left = new Array();

Array right = new Array();

for (int i = 0; i < Array.Length; i++)

if (i != Array.Length / 2) {

if (Array[i] <= middle)

left.Add(Array[i]);

else

right.Add(Array[i]);

}

return concatenate(quickSort(left), middle, quickSort(right));

}

The challenge of a quick sort is to determine a reasonable "midpoint" value for dividing the data into two groups. The efficiency of the algorithm is entirely dependent upon how successfully an accurate midpoint value is selected. In a best case, the runtime is O(n \* log n). In the worst case-where one of the two groups always has only a single element-the runtime drops to O(n²).

## Radix Sort

The radix sort was designed originally to sort data without having to directly compare elements to each other. A radix sort first takes the least-significant digit (or several digits, or bits), and places the values into buckets. If we took 4 bits at a time, we would need 16 buckets. We then put the list back together, and have a resulting list that is sorted by the least significant radix. We then do the same process, this time using the second-least significant radix. We lather, rinse, and repeat, until we get to the most significant radix, at which point the final result is a properly sorted list.

For example, let's look at a list of numbers and do a radix sort using a 1-bit radix. Notice that it takes us 4 steps to get our final result, and that on each step we setup exactly two buckets:

{6, 9, 1, 4, 15, 12, 5, 6, 7, 11}

{6, 4, 12, 6} {9, 1, 15, 5, 7, 11}

{4, 12, 9, 1, 5} {6, 6, 15, 7, 11}

{9, 1, 11} {4, 12, 5, 6, 6, 15, 7}

{1, 4, 5, 6, 6, 7} {9, 11, 12, 15}

Let's do the same thing, but now using a 2-bit radix. Notice that it will only take us two steps to get our result, but each step requires setting up 4 buckets:

{6, 9, 1, 4, 15, 12, 5, 6, 7, 11}

{4, 12} {9, 1, 5} {6, 6} {15, 7, 11}

{1} {4, 5, 6, 6, 7} {9, 11} {12, 15}

Given the relatively small scope of our example, we could use a 4-bit radix and sort our list in a single step with 16 buckets:

{6, 9, 1, 4, 15, 12, 5, 6, 7, 11}

{1} {} {} {4} {5} {6, 6} {7} {} {9} {} {11} {12} {} {} {15}

Notice, however, in the last example, that we have several empty buckets. This illustrates the point that, on a much larger scale, there is an obvious ceiling to how much we can increase the size of our radix before we start to push the limits of available memory. The processing time to reassemble a large number of buckets back into a single list would also become an important consideration at some point.

Because radix sort is qualitatively different than comparison sorting, it is able to perform at greater efficiency in many cases. The runtime is O(n \* k), where k is the size of the key. (32-bit integers, taken 4 bits at a time, would have k = 8.) The primary disadvantage is that some types of data may use very long keys (strings, for instance), or may not easily lend itself to a representation that can be processed from least significant to most-significant. (Negative floating-point values are the most commonly cited example.)

## STL sort & stable\_sort

#include <iostream>

#include <algorithm>

#include <vector>

using namespace std;

// We need this function to define how to sort

// the vector. We will pass this function into the

// third parameter and it will tell it to sort descendingly.

bool wayToSort(int i, int j) { return i > j; }

int main()

{

vector<int> intVec = {56, 32, -43, 23, 12, 93, 132, -154};

// Do not include the () when you call wayToSort

// It must be passed as a function pointer or function object

sort(intVec.begin(), intVec.end(), wayToSort);

for (int i : intVec)

cout << i << " ";

return 0;

}

## string-int

#include <sstream>

#include <string>

using namespace std;

string myStream = "45";

istringstream buffer(myString);

int value;

buffer >> value; // value = 45

## int-string

int a = 10;

ostringstream ssa;

ssa << a;

string str = ssa.str();

## lexicographical compare function - string

bool lfc(string a, string b){

return lexicographical\_compare(&a,&a+a.length(),&b,&b+b.length());

}

## std fill

fill(&data[0][0],&data[0][0]+sizeof(data)/sizeof(data[0][0]),'\0');