GAME2001 Data Structures and Algorithms

Fall 2020

Week 4
Sorting

Sorting

- many different algorithms, each with their own strengths and weaknesses in various situations
- have to move through the list and order each object based on how it compares to the others
  - has to be efficient in terms of speed and memory consumption

- bubble sort, selection sort, and insertion sort are also known as elementary sorting algorithms

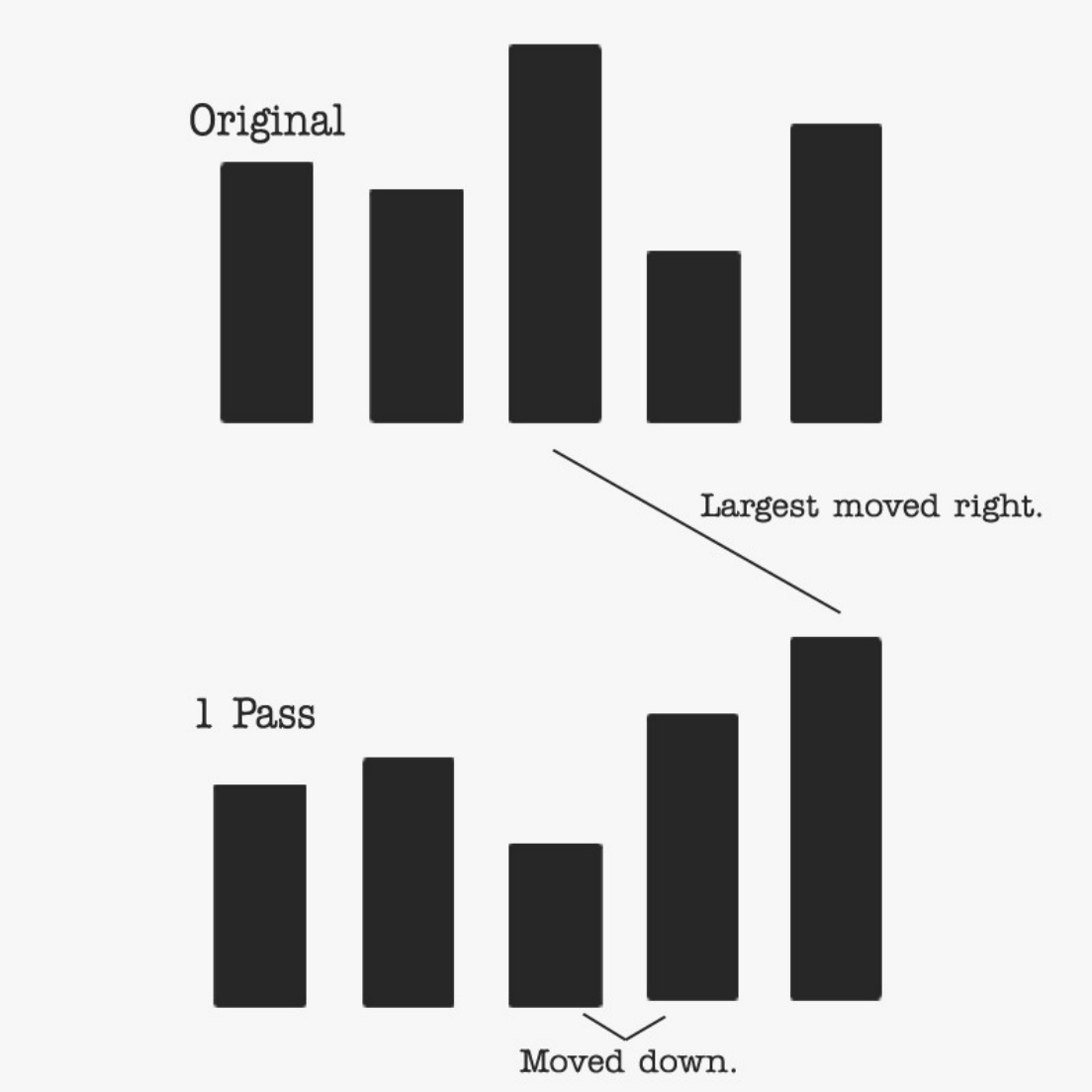
Bubble Sort

- Simple to implement and understand
  - but very slow
- It works by comparing two values in a data structure
  - If the object on the left is bigger then the two objects get swapped
    - this continues until the largest objects is at the right

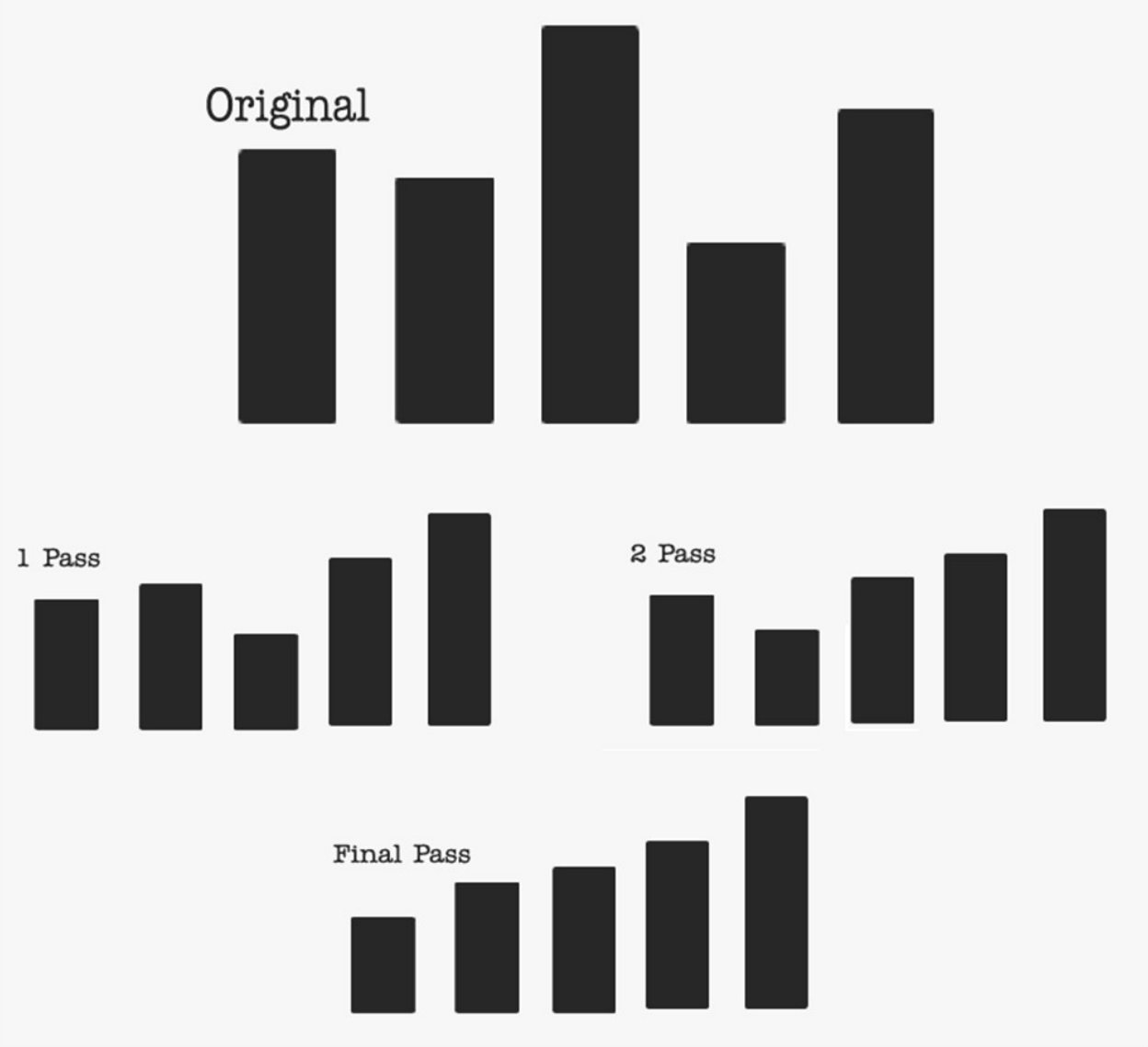
Bubble Sort

- The first pass moves the largest object from its position in the array to the far right
- The second pass moves the second largest object to one element before the largest and so on

- The bubble sort needs to loop through the array once for each object



- One pass over an array using bubble sort


Original
1 Pass
2 Pass
Final Pass

Bubble Sort

- gets its name from the fact that the large items bubble up at the top of the list until all items are in order
- Because of the number of passes, comparisons, and copies and the brute-force style of the algorithm, it is slow (especially user-defined objects)

Bubble Sort

```
1  template<typename T>
2  class UnorderedArray
3  {
4     public:
5        void BubbleSort()
6        {
7           assert(m_array != NULL);
8
9           for(int k = m_numElements - 1; k > 0; k--)
10          {
11             for(int i = 0; i < k; i++)
12             {
13                if(m_array[i] > m_array[i + 1])
14                {
15                   T temp = m_array[i];
16                   m_array[i] = m_array[i + 1];
17                   m_array[i + 1] = temp;
18                }
19             }
20          }
21       }
22 };
```

```cpp
#include <iostream>
#include "Arrays.h"

using namespace std;

int main(int args, char *arg[])
{
    cout << "Bubble Sort Algorithm" << endl << endl;

    UnorderedArray<int> array(5);
    array.push(80);
    array.push(64);
    array.push(99);
    array.push(76);
    array.push(5);

    cout << "Before sort:";
    for(int i = 0; i < 5; i++)
    {
        cout << " " << array[i];
    }
    cout << endl;

    array.BubbleSort();

    cout << " After sort:";
    for(int i = 0; i < 5; i++)
    {
        cout << " " << array[i];
    }
    cout << endl << endl;

    return 1;
}
```

```
Bubble Sort Algorithm

Before sort: 80 64 99 76 5
 After sort: 5 64 76 80 99
```

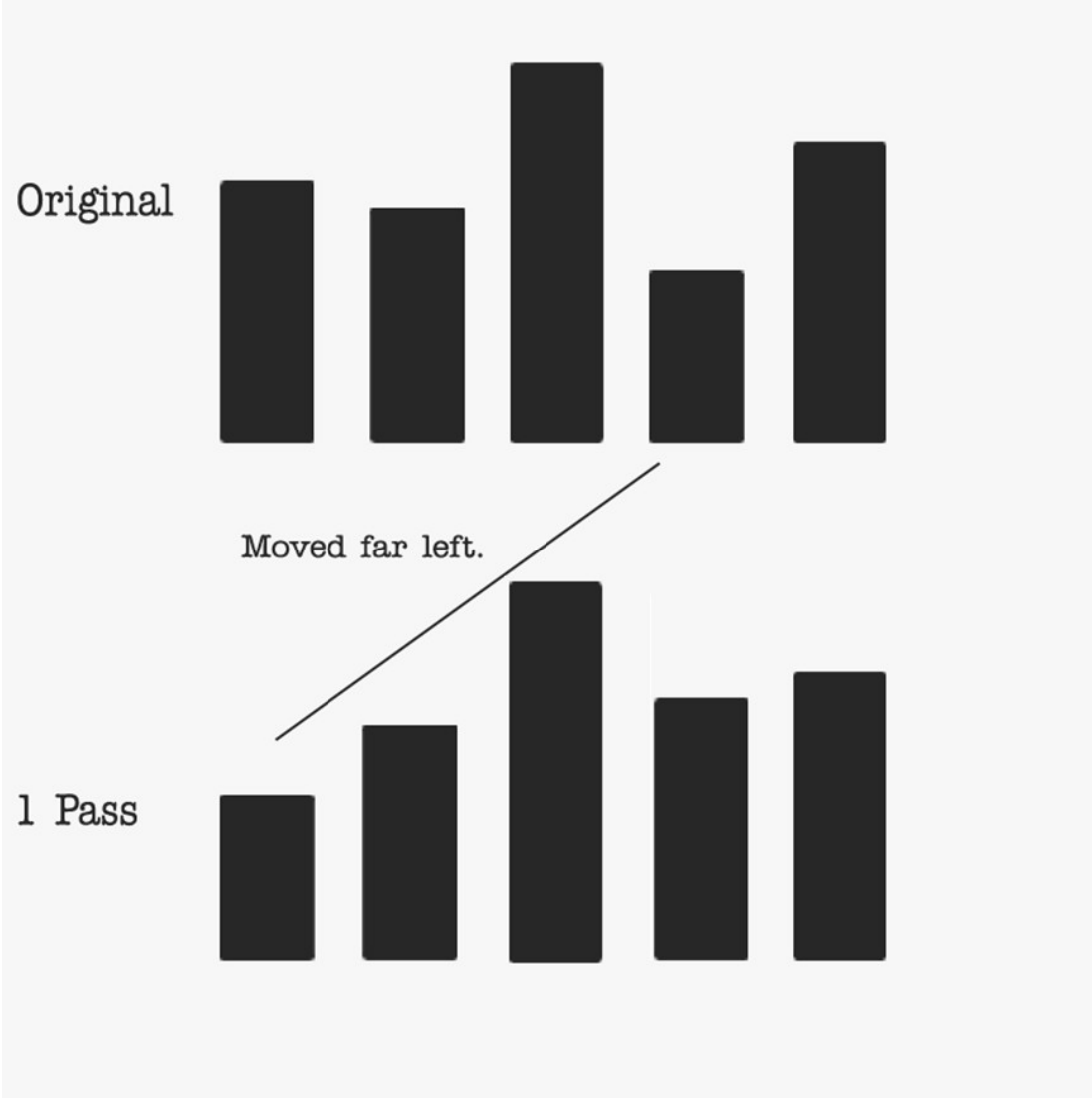
Selection Sort

- similar to bubble sort
- operates left to right instead of right to left
  - comparisons in $O(N^2)$
  - better as N grown larger
- not as many swaps, on average, but there are still many comparisons and passes

Selection Sort

- loop through the list and keep track of the smallest item
- once a pass over the list is complete, the smallest item in the list is swapped with the first item
- only one swap per pass, but the number of comparisons is the same

- will sort items from the left to the right, while the bubble sort will sort items from the right to the left


Original
Moved far left.
1 Pass

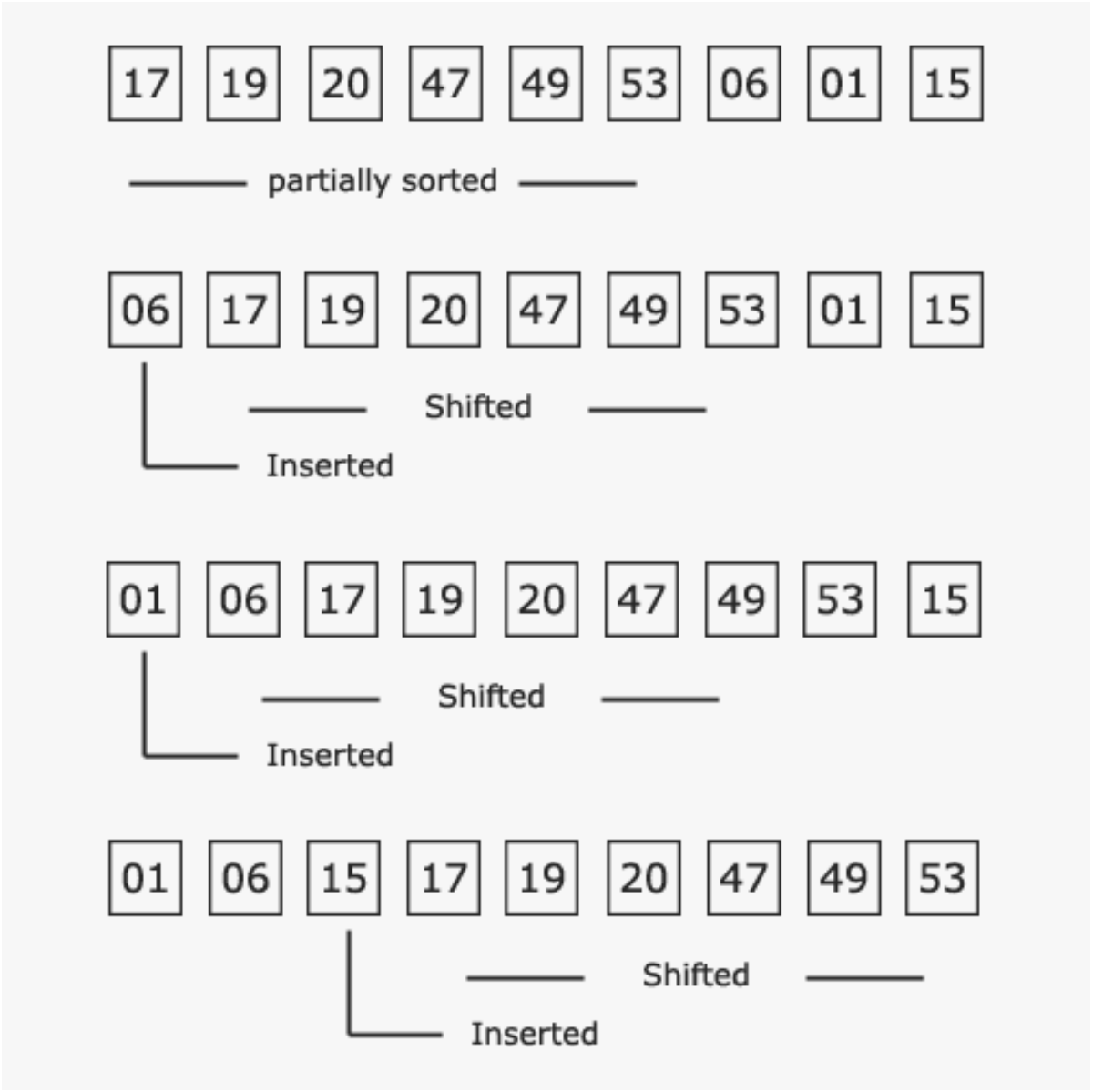
```
1  template<typename T>
2  class UnorderedArray
3  {
4     public:
5        void SelectionSort()
6        {
7           assert(m_array != NULL);
8
9           T temp;
10          int min = 0;
11
12          for(int k = 0; k < m_numElements - 1; k++)
13          {
14             min = k;
15
16             for(int i = k + 1; i < m_numElements; i++)
17             {
18                if(m_array[i] < m_array[min])
19                   min = i;
20             }
21
22             if(m_array[k] > m_array[min])
23             {
24                temp = m_array[k];
25                m_array[k] = m_array[min];
26                m_array[min] = temp;
27             }
28          }
29       }
30 };
```

```cpp
#include <iostream>
#include "Arrays.h"

using namespace std;

int main(int args, char *arg[])
{
    cout << "Selection Sort Algorithm" << endl << endl;

    UnorderedArray<int> array(5);
    array.push(80);
    array.push(64);
    array.push(99);
    array.push(76);
    array.push(5);

    cout << "Before sort:";
    for(int i = 0; i < 5; i++)
    {
        cout << " " << array[i];
    }
    cout << endl;

    array.SelectionSort();

    cout << " After sort:";
    for(int i = 0; i < 5; i++)
    {
        cout << " " << array[i];
    }
    cout << endl << endl;

    return 1;
}
```

```
Selection Sort Algorithm

Before sort: 80 64 99 76 5
 After sort: 5 64 76 80 99
```

Insertion Sort

- works by comparing an item to the sorted part of the data structure
- initially the sorted part consists of just the first element
- the item gets placed in the correct position in the sorted part


17 19 20 47 49 53 06 01 15
partially sorted
06 17 19 20 47 49 53 01 15
Shifted
Inserted
01 06 17 19 20 47 49 53 15
Shifted
Inserted
01 06 15 17 19 20 47 49 53
Shifted
Inserted

Insertion Sort

- Slightly faster than selection sort
- Faster than bubble sort
- Worst case
  - When data is sorted in reverse order
  - Same as bubble sort

Insertion Sort

```
1  template<typename T>
2  class UnorderedArray
3  {
4     public:
5        void InsertionSort()
6        {
7           assert(m_array != NULL);
8
9           T temp;
10          int i = 0;
11
12          for(int k = 1; k < m_numElements; k++)
13          {
14             // item to be placed in the right slot
15             temp = m_array[k];
16             i = k;
17
18             while(i > 0 && m_array[i - 1] >= temp)
19             {
20                // switch the bigger value up
21                m_array[i] = m_array[i - 1];
22                i--;
23             }
24             // place item in correct slot
25             m_array[i] = temp;
26          }
27       }
28 };
```

```cpp
#include <iostream>
#include "Arrays.h"

using namespace std;

int main(int args, char *arg[])
{
    cout << "Insertion Sort Algorithm" << endl << endl;

    UnorderedArray<int> array(5);
    array.push(80);
    array.push(64);
    array.push(99);
    array.push(76);
    array.push(5);

    cout << "Before sort:";
    for(int i = 0; i < 5; i++)
    {
        cout << " " << array[i];
    }
    cout << endl;

    array.InsertionSort();

    cout << " After sort:";
    for(int i = 0; i < 5; i++)
    {
        cout << " " << array[i];
    }
    cout << endl << endl;

    return 1;
}
```

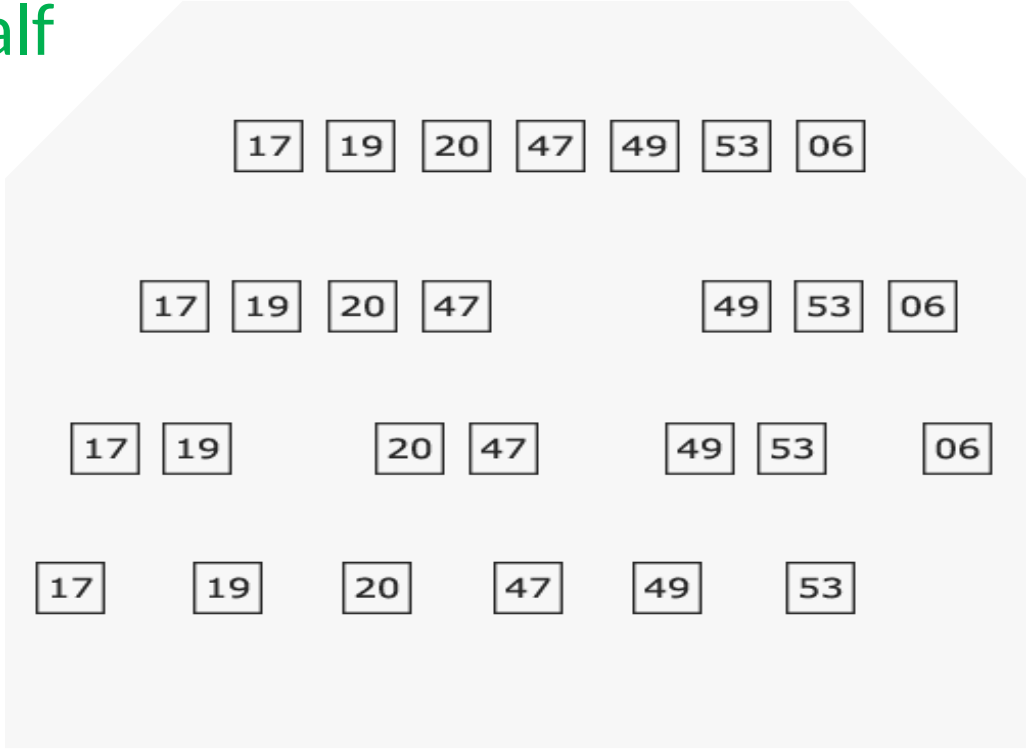
```
Insertion Sort Algorithm

Before sort: 80 64 99 76 5
 After sort: 5 64 76 80 99
```

Merge Sort

- Most efficient sorting algorithm discussed so far
- O(N * log N)
  - a list of 25,000 elements
    - $O(N^2)$ would have an N of 625,000,000
    - O(N * log N) would have an N of only 109,949

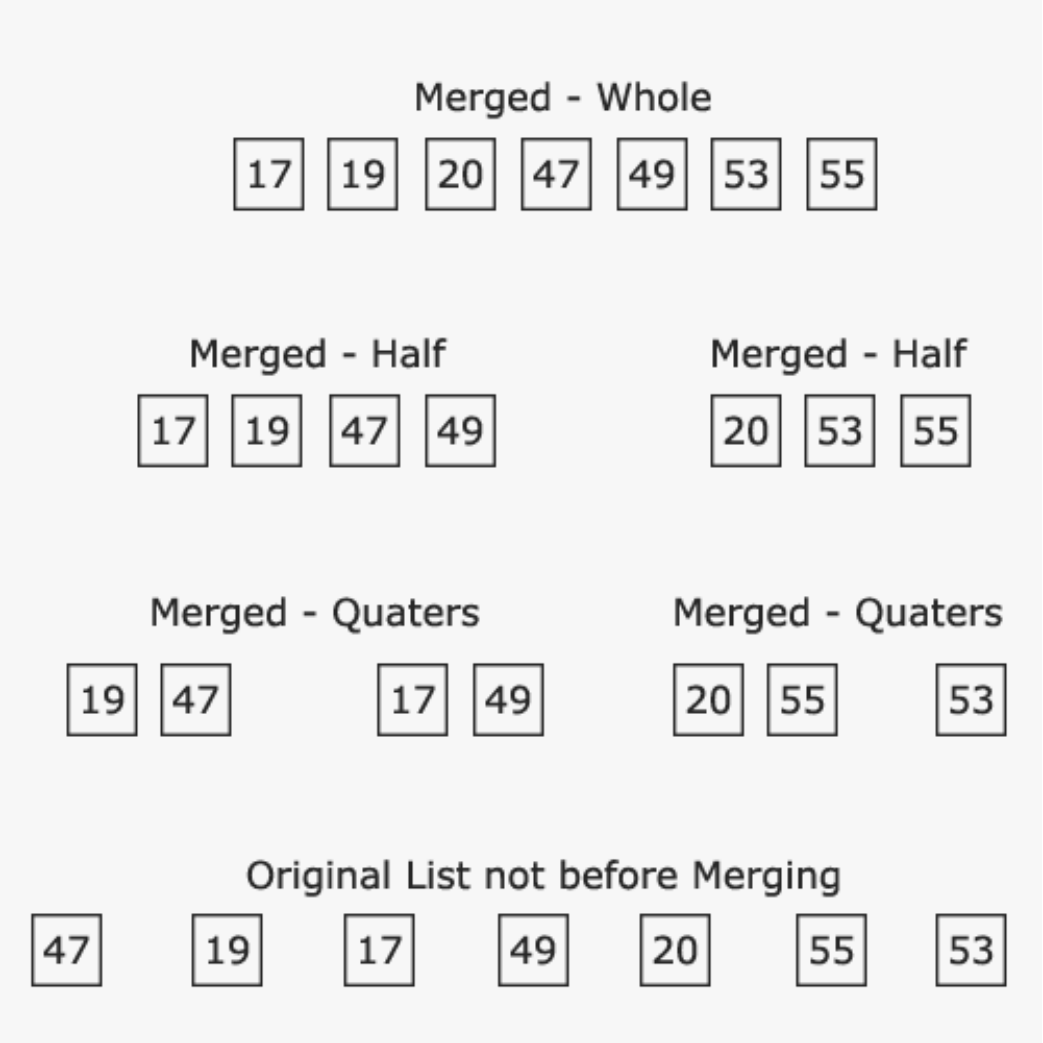
Merge Sort

- recursively splits the list into two halves.
- continues until only one element remains on each side of the original half



Merge Sort

- Once one element remains on both sides they are merged together and sorted
  - elements from both sides are combined into one list
- Once those two elements are merged, the recursive function returns



- Merging until we have a sorted list

Merge Sort

- has the disadvantage of needing another array that is equal in size to the array that is being sorted

- twice as much memory as bubble, insertion and selection sort

Merge Sort - Recap

- First split the list into halves
  - which are further split in half recursively until no more splits can be done (one element left on each side)
- Once at the bottom of the recursive heap, we can start returning from the recursive calls
  - for each return we merge the elements at that level into a temporary array, which is sorted and copied back into the original array
- By the time we make it to the top of the recursive heap, the entire list has been sorted

```
template<typename T>
class UnorderedArray
{
   public:
      void MergeSort()
      {
         assert(m_array != NULL);

         T *tempArray = new T[m_numElements];
         assert(tempArray != NULL);

         MergeSort(tempArray, 0, m_numElements - 1);
         delete[] tempArray;
      }

   private:
      void MergeSort(T *tempArray, int lowerBound, int upperBound)
      {
         if(lowerBound == upperBound)
            return;

         int mid = (lowerBound + upperBound) >> 1;

         MergeSort(tempArray, lowerBound, mid);
         MergeSort(tempArray, mid + 1, upperBound);

         Merge(tempArray, lowerBound, mid + 1, upperBound);
      }

      void Merge(T *tempArray, int low, int mid, int upper)
      {
         int tempLow = low, tempMid = mid - 1;
         int index = 0;

         while(low <= tempMid && mid <= upper)
         {
            if(m_array[low] < m_array[mid])
            {
               tempArray[index++] = m_array[low++];
            }
            else
            {
               tempArray[index++] = m_array[mid++];
            }
         }

         while(low <= tempMid)
         {
            tempArray[index++] = m_array[low++];
         }

         while(mid <= upper)
         {
            tempArray[index++] = m_array[mid++];
         }

         for(int i = 0; i < upper - tempLow + 1; i++)
         {
            m_array[tempLow + i] = tempArray[i];
         }
      }
};
```

```cpp
#include <iostream>
#include "Arrays.h"

using namespace std;

int main(int args, char *arg[])
{
    cout << "Merge Sort Algorithm" << endl << endl;

    UnorderedArray<int> array(5);
    array.push(80);
    array.push(64);
    array.push(99);
    array.push(76);
    array.push(5);

    cout << "Before sort:";
    for(int i = 0; i < 5; i++)
    {
        cout << " " << array[i];
    }
    cout << endl;

    array.MergeSort();

    cout << " After sort:";
    for(int i = 0; i < 5; i++)
    {
        cout << " " << array[i];
    }
    cout << endl << endl;

    return 1;
}
```

```
Merge Sort Algorithm

Before sort: 80 64 99 76 5
 After sort: 5 64 76 80 99
```

STL Sorting

- sort()
  - sort objects into ascending or descending order based on a binary predicate
  - takes as parameters a random access iterator to the first element, a random access iterator to the last element and the optional binary predicate
  - algorithm is not stable

STL Sorting

- stable_sort()
  - same as the sort() function, with the exception that the stable_sort() function is stable

STL Sorting

- partial_sort()
  - partially sort based on a binary predicate
  - entire data structure or only on a few of its internal elements
  - takes as parameters
    - a random access iterator to the first element in the range to be sorted,
    - a random access iterator to the last element in the range to be sorted,
    - a random access iterator to the end of the list (which might not be the end of the sorting range),
    - and optionally a binary predicate

```cpp
#include<iostream>
#include<vector>
#include<algorithm>
#include<functional>
#include<iterator>

using namespace std;

inline bool CompareNoCase(char lVal, char rVal)
{
   return tolower(lVal) < tolower(rVal);
}

int main(int args, char *arg[])
{
   cout << "STL Sorting Algorithm" << endl;

   char str1[] = "lekiamhjdqn";
   char str2[] = "peuyxknasdb";
   vector<int> int1;

   int1.push_back(58);
   int1.push_back(23);
   int1.push_back(1);
   int1.push_back(53);
   int1.push_back(33);
   int1.push_back(84);
   int1.push_back(12);

   cout << "Original str1 data: " << str1 << "." << endl;
   sort(str1, str1 + (sizeof(str1) - 1), CompareNoCase);
   cout << "  Sorted str1 data: " << str1 << "." << endl;

   cout << endl;

   cout << "Original str2 data: " << str2 << "." << endl;
   stable_sort(str2, str2 + (sizeof(str2) - 1), CompareNoCase);
   cout << "  Sorted str2 data: " << str2 << "." << endl;

   cout << endl;

   ostream_iterator<int> output(cout, " ");

   cout << "Original int1 data: ";
   copy(int1.begin(), int1.end(), output);
   cout << endl;

   partial_sort(int1.begin(), int1.begin() + int1.size(),
                int1.end(), less<int>());

   cout << "  Sorted int1 data: ";
   copy(int1.begin(), int1.end(), output);
   cout << endl << endl;

   return 1;
}
```

```
STL Sorting Algorithm
Original str1 data: lekiamhjdqn.
  Sorted str1 data: adehijklmnq.

Original str2 data: peuyxknasdb.
  Sorted str2 data: abdeknpsuxy.

Original int1 data: 58 23 1 53 33 84 12
  Sorted int1 data: 1 12 23 33 53 58 84
```