GAME2001 Data Structures and Algorithms

Fall 2020

Week 11
Advanced Sorting

Advanced Sorting

- Shellsort
- Quicksort
- Radix sort

- Much faster than simple sorts
- Do not require extra memory

Shellsort

- Great for medium size data
  - Quicksort can outperform it for larger data
- Based on insertion sort
- It minimizes the number of copies to increase performance

Shellsort

- Performs insertion sort on a few widely spread elements
- During each pass the range (gap) is reduced until all elements are in their correct position
- Gets the array close to being partially sorted
  - Causes later passes to execute faster
- When the range is 1, insertion sort is applied

Shellsort

- How do you choose a good initial range?
  - Knuth method h = h * 3 + 1
  - h starts out as 1 and keeps going until its larger than the array size, at that point the previous size is used as the gap sequence

Array size = 500 1, 4, 13, 40, 121, 364, 1093

Shellsort

- You can use the inverse algorithm to figure out the previous gaps
  - h = (h – 1) / 3

Shellsort

```
1   void Shellsort()
2   {
3      assert(m_array != NULL);
4
5      T temp;
6      int i = 0, k = 0;
7
8      // Sequence...
9      int seq = 1;
10
11     while(seq <= m_numElements / 3)
12        seq = seq * 3 + 1;
13
14     while(seq > 0)
15     {
16        for(k = seq; k < m_numElements; k++)
17        {
18           temp = m_array[k];
19           i = k;
20
21           while(i > seq - 1 && m_array[i - seq] >= temp)
22           {
23              m_array[i] = m_array[i - seq];
24              i -= seq;
25           }
26
27           m_array[i] = temp;
28        }
29
30        seq = (seq - 1) / 3;
31     }
32  }
```

```
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 };
```

Shellsort Example

```
2  #include "ShellsortArray.h"
3
4  using namespace std;
5
6  int main(int args, char *arg[])
7  {
8     cout << "Shellsort Algorithm" << endl << endl;
9
10    const int size = 10;
11    int i = 0;
12
13    UnorderedArray<int> array(size);
14
15    for(i = 0; i < size; i++)
16       array.push(rand() % 100);
17
18    cout << "Before shellsort sort:";
19
20    for(i = 0; i < size; i++)
21       cout << " " << array[i];
22
23    cout << endl;
```

```
25    array.Shellsort();
26
27    cout << " After shellsort sort:";
28
29    for(i = 0; i < size; i++)
30       cout << " " << array[i];
31
32    cout << endl << endl;
33
34    return 1;
35 }
```

```
Shellsort Algorithm

Before shellsort sort: 41 67 34 0 69 24 78 58 62 64
 After shellsort sort: 0 24 34 41 58 62 64 67 69 78
```

Partitioning

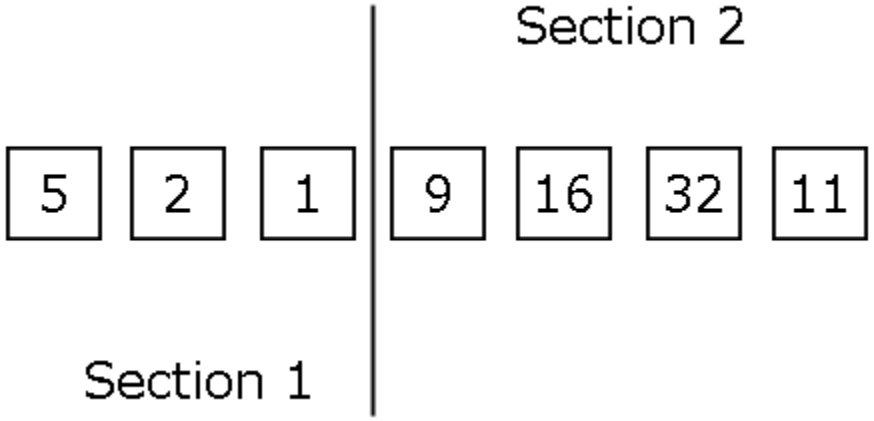
- Used as the basis for quicksort
- Sole purpose is to split data into two sections

- Partitioning Algorithm
  - Takes a data set that needs to be partitioned and a condition by which it is split up

- Pivot value: value used in the comparison
- Pivot index: the position in the container where the first section ends and the second begins

Partitioning

Pivot value = 10



Partitioning

- not a sorting algorithm
- it is one step closer to being sorted
- O (N)

Partitioning

```
int Partition(int lIndex, int rIndex, T pivot)
{
   int currentLeft = lIndex;
   int currentRight = rIndex;

   while(1)
   {
      while(currentLeft < rIndex && m_array[currentLeft] < pivot)
      {
         // Searching left for val bigger than pivot.
         // This will break when it finds it or moves to the end.
         currentLeft++;
      }

      while(currentRight > lIndex && m_array[currentRight] > pivot)
      {
         // Same as left side.
         currentRight--;
      }

      if(currentLeft >= currentRight)
      {
         // Done with partition (no more to search).
         break;
      }

      // Swap elements if we get here.
      SwapElements(currentLeft, currentRight);
   }

   // Returns position of the pivot.
   return currentLeft;
}
```

```
void SwapElements(int index1, int index2)
{
   assert(index1 >= 0 && index1 < m_numElements);
   assert(index2 >= 0 && index2 < m_numElements);
   assert(m_array != NULL);

   T temp = m_array[index1];
   m_array[index1] = m_array[index2];
   m_array[index2] = temp;
}

int Partition(T pivot)
{
   return Partition(0, m_numElements - 1, pivot);
}
```

Partitioning Example

```
#include <iostream>
#include "PartitioningArray.h"

using namespace std;

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

   const int size = 10;
   int i = 0;
   int pivotValue = 60;

   UnorderedArray<int> array(size);

   // Insert elements and print basic stats.
   for(i = 0; i < size; i++)
      array.push(rand() % 100);

   cout << "Array size - " << size << " pivot value - "
        << pivotValue << "." << endl << endl;

   // Display elements.
   cout << "Before partitioning:";

   for(i = 0; i < size; i++)
      cout << " " << array[i];

   cout << endl << endl;

   // Partition then display results.
   int pivot = array.Partition(pivotValue);

   cout << "After partitioning (pivot index - "
        << pivot << "):";

   for(i = 0; i < size; i++)
      cout << " " << array[i];

   cout << endl << endl;

   return 1;
}
```

```
Partitioning Algorithm
Array size - 10 pivot value - 60.
Before partitioning: 41 67 34 0 69 24 78 58 62 64
After partitioning (pivot index - 5): 41 58 34 0 24 69 78 67 62 64
```

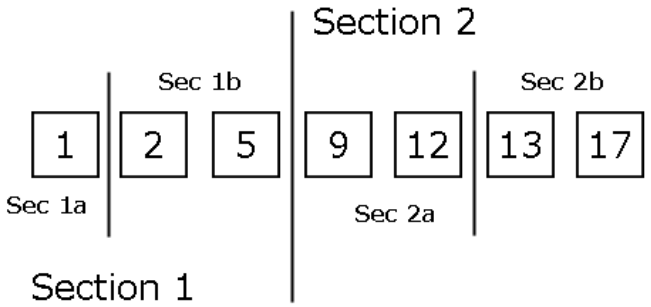
Quicksort

- Very popular for large data sets
- Based on the partitioning algorithm
- Uses recursion and partitioning to partition data sections until the lowest level is reached
  - Data is sorted

Quicksort

1) Pick a pivot value and partition the array into two sections
2) Recursively partition the two sections further until there is only one element left

- The key is to choose a good pivot value

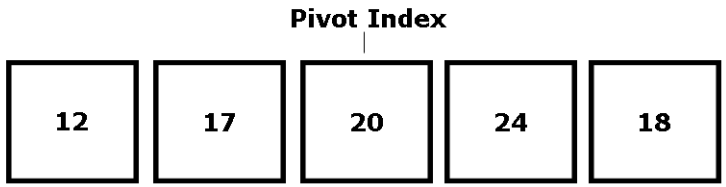


Quicksort

```
1  template<typename T>
2  class UnorderedArray
3  {
4     public:
5        void SwapElements(int index1, int index2)
6        {
7           assert(index1 >= 0 && index1 < m_numElements);
8           assert(index2 >= 0 && index2 < m_numElements);
9           assert(m_array != NULL);
10
11          T temp = m_array[index1];
12          m_array[index1] = m_array[index2];
13          m_array[index2] = temp;
14       }
15
16       void Quicksort()
17       {
18          QuickSort(0, m_numElements - 1);
19       }
20
21    private:
22       void QuickSort(int lVal, int rVal)
23       {
24          if((rVal - lVal) <= 0)
25             return;
26
27          int pivotIndex = Partition(lVal, rVal,
28                                     m_array[rVal]);
29
30          QuickSort(lVal, pivotIndex - 1);
31          QuickSort(pivotIndex + 1, rVal);
32       }
```

Quicksort

```
34      int Partition(int lIndex, int rIndex, T pivot)
35      {
36          int currentLeft = lIndex;
37          int currentRight = rIndex - 1;
38
39          while(1)
40          {
41              while(m_array[currentLeft] < pivot)
42              {
43                  // Searching left for val bigger than pivot.
44                  // This will end when it finds it.
45                  currentLeft++;
46              }
47
48              while(currentRight > 0 && m_array[currentRight] > pivot)
49              {
50                  // Same as left side.
51                  currentRight--;
52              }
53
54              if(currentLeft >= currentRight)
55              {
56                  // Done with partition (no more to search).
57                  break;
58              }
59
60              // Swap elements if we get here.
61              SwapElements(currentLeft, currentRight);
62          }
63
64          SwapElements(currentLeft, rIndex);
65
66          // Returns position of the pivot.
67          return currentLeft;
68      }
69  };
```



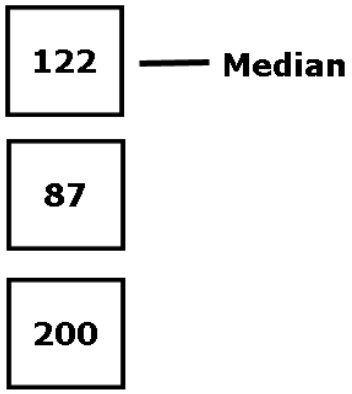
Quicksort Example

```
#include <iostream>
#include "QuicksortArray.h"

using namespace std;

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

   const int size = 10;
   int i = 0;
   UnorderedArray<int> array(size);

   // Insert elements and print basic stats.
   for(i = 0; i < size; i++)
      array.push(10 + rand() % 90);

   // Display elements.
   cout << "Before Quicksort:";

   for(i = 0; i < size; i++)
      cout << " " << array[i];

   cout << endl << endl;

   // Sort then display results.
   array.Quicksort();

   cout << " After Quicksort:";

   for(i = 0; i < size; i++)
      cout << " " << array[i];

   cout << endl << endl;

   return 1;
}
```

```
Quicksort Algorithm
Before Quicksort: 51 27 44 50 99 74 58 28 62 84
 After Quicksort: 27 28 44 50 51 58 62 74 84 99
```

Quicksort: Median-of-three

- A method for quickly picking a good pivot value
- The first, middle and last element are examined and the median is chosen as the pivot value



Quicksort: Median-of-three

- Problem:
  - Cant use it with less than three items
  - We can fall back to another simpler searching algorithm in this case

Quicksort: Median-of-three

```
#define QUICKSORT_CUTOFF    4

template<typename T>
class UnorderedArray
{
   public:
      void InsertionSort()
      {
         InsertionSort(0, m_numElements - 1);
      }

      void SwapElements(int index1, int index2)
      {
         assert(index1 >= 0 && index1 < m_numElements);
         assert(index2 >= 0 && index2 < m_numElements);
         assert(m_array != NULL);

         T temp = m_array[index1];
         m_array[index1] = m_array[index2];
         m_array[index2] = temp;
      }

      void Quicksort()
      {
         QuickSort(0, m_numElements - 1);
      }
```

```
   private:
      void QuickSort(int lVal, int rVal)
      {
         if((rVal - lVal + 1) < QUICKSORT_CUTOFF)
         {
            InsertionSort(lVal, rVal);
            return;
         }

         int center = (lVal + rVal) / 2;

         if(m_array[lVal] > m_array[center])
            SwapElements(lVal, center);

         if(m_array[lVal] > m_array[rVal])
            SwapElements(lVal, rVal);

         if(m_array[center] > m_array[rVal])
            SwapElements(center, rVal);

         int pivotIndex = Partition(lVal, rVal, center);

         QuickSort(lVal, pivotIndex - 1);
         QuickSort(pivotIndex, rVal);
      }
```

Quicksort: Median-of-three

```
54      int Partition(int lIndex, int rIndex, int pivot)
55      {
56          while(1)
57          {
58              while(m_array[++lIndex] < m_array[pivot]);
59              while(m_array[--rIndex] > m_array[pivot]);
60
61              if(lIndex >= rIndex)
62                  break;
63
64              SwapElements(lIndex, rIndex);
65          }
66
67          // Returns position of the pivot.
68          return lIndex;
69      }
```

```
71      void InsertionSort(int lVal, int rVal)
72      {
73          assert(m_array != NULL);
74
75          T temp;
76          int i = 0;
77
78          for(int k = lVal + 1; k <= rVal; k++)
79          {
80              temp = m_array[k];
81              i = k;
82
83              while(i > lVal && m_array[i - 1] >= temp)
84              {
85                  m_array[i] = m_array[i - 1];
86                  i--;
87              }
88
89              m_array[i] = temp;
90          }
91      }
92  };
```

Quicksort: Median-of-three Example

```
1  #include <iostream>
2  #include "Quicksort3Array.h"
3
4  using namespace std;
5
6  int main(int args, char *arg[])
7  {
8     cout << "Median-Of-Three Quicksort Algorithm"
9          << endl << endl;
10
11    const int size = 10;
12    int i = 0;
13    UnorderedArray<int> array(size);
14
15    // Insert elements and print basic stats.
16    for(i = 0; i < size; i++)
17       array.push(10 + rand() % 90);
18
19    // Display elements.
20    cout << "Before Quicksort:";
21
22    for(i = 0; i < size; i++)
23       cout << " " << array[i];
24
25    cout << endl << endl;
```

```
27    // Sort then display results.
28    array.Quicksort();
29
30    cout << " After Quicksort:";
31
32    for(i = 0; i < size; i++)
33       cout << " " << array[i];
34
35    cout << endl << endl;
36
37    return 1;
38 }
```

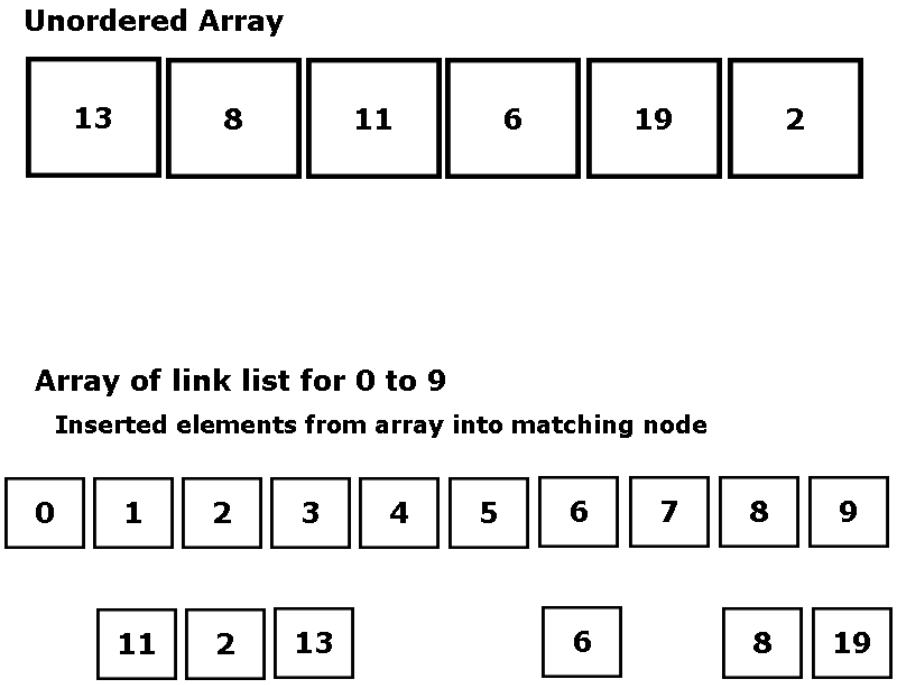
```
Median-Of-Three Quicksort Algorithm
Before Quicksort: 51 27 44 50 99 74 58 28 62 84
 After Quicksort: 27 28 44 50 51 58 62 74 84 99
```

Radix sort

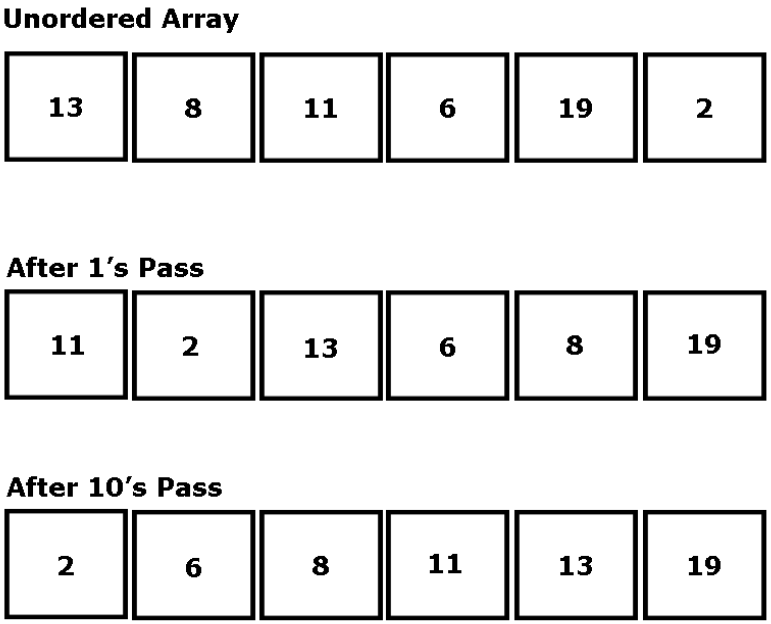
- Quickly sorts the elements in a container without performing comparisons of internal elements
- Creates a number of internal containers that it uses to perform its job

Radix sort

- sort numbers that use the decimal system



Radix sort


Unordered Array
13
8
11
6
19
2
After 1's Pass
11
2
13
6
8
19
After 10's Pass
2
6
8
11
13
19

```cpp
#include <iostream>
#include <deque>

using namespace std;

#define BASE            10
#define MAX_POSITIONS   2

void RadixSort(int *array, int size)
{
   // Base index, radix index, counter.
   int b = 0, r = 0, i = 0;
   // Container conter, base factor.
   int index = 0, factor = 0;

   // List of containers for the sort.
   deque<int> qList[BASE];

   // Place in containers then take them off for every base.
   for(b = 1, factor = 1; b <= MAX_POSITIONS; factor *= BASE, b++)
   {
      for(r = 0; r < size; r++)
      {
         index = (array[r] / factor) % BASE;
         qList[index].push_back(array[r]);
      }
      for(r = 0, i = 0; r < BASE; r++)
      {
         while(qList[r].empty() != true)
         {
            array[i++] = qList[r].front();
            qList[r].pop_front();
         }
      }
   }
}

int main(int args, char **argc)
{
   cout << "Radix Sort Example"
        << endl << endl;

   const int size = 10;
   int array[size];
   int i = 0;

   // Populate array.
   for(i = 0; i < size; i++)
      array[i] = 10 + rand() % 89;

   // Display array contents.
   cout << "Array contents before sort: ";

   for(i = 0; i < size; i++)
      cout << " " << array[i];

   cout << endl;

   // Radix sorting.
   RadixSort(array, size);

   // Display array contents.
   cout << " Array contents after sort: ";

   for(i = 0; i < size; i++)
      cout << " " << array[i];

   cout << endl << endl;

   return 1;
}
```

```
Radix Sort Example

Array contents before sort:  51 54 25 77 44 70 96 87 94 88
 Array contents after sort:  25 44 51 54 70 77 87 88 94 96
```