Laboratory 20B
Comparing QuickSorts

CS-102

Chapter 20

Part 11

Basic Strategy of the Hoare QuickSort

1. Start by taking the element that was positioned in the middle of the array, and swapping that value with the value that was at the beginning of the array. We will call this value the pivotValue. We shall its index, pivotIndex.
2. So now you proceed to examine every number from the element in location 1 to the element in location n-1. If the element is smaller than the pivotValue, then we increment the pivotIndex by 1, and swap its value with that located in the new pivotIndex which is 1 greater than the old index.
3. Once you know how many numbers, x, are smaller than the number in location 0, you now know where our number, currently in location 0 should go. In addition, we have swapped all larger numbers to indices > pivotIndex.
4. You now can take the number stored in 0 and swap it with the number currently stored in location x.
5. Now the number stored in x is in its proper location in the array and will never be moved again. All numbers larger are located above it and those lower below it.
6. Now you repeat the process above for both those numbers from 0 up to x-1 and for those numbers stored in x+1 up to n-1.
7. By exhaustively repeating this process until every number is relocated to its proper position, you will end up sorting the entire array in ascending order.

Basic Strategy of the Efficient QuickSort

1. We’ll start by making the entire array a single partition, and we’ll begin by seeing where the element stored in the start location (location zero) will ultimately be positioned within the partition. This element will be called the pivotValue.
2. So now you proceed to check every number, starting from the element in location 1 until you find a number larger than the pivotValue. We’ll call it the largerlower which is located at location x.
3. Having found that number, we will now start at the end of the list (location n-1), and proceed to check every number, working our way down the list, until we find a number less than the pivotValue. We’ll call it the smallerhigher which is located in location y.
4. Now we swap the largerlower with the smallerhigher – swapping the contents of locations x and y.
5. Now that the swapped numbers are in their proper relationship to the pivotnumber, we return to step 2 and continue on from where we left off at location x.
6. In the same way, when we again find another number larger than the pivotnumber, we set x to that new location, and then go to the location y and start moving down looking for a number smaller than the pivotnumber. Then, again, we swap the two.
7. This process proceeds until location x exceeds location y. Then the partition is done.

```
// QuickSort – Vector Version Using & C.A.R.Hoare sorting Algorithm
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <vector>
#include <random>
#include <algorithm>
#include <chrono>
using namespace std;
void commaSeparate(int);
void quickSort(vector<int>&,int,int);
int partition(vector<int>&,int,int);
int bigRand(int, int);
const int MAX_VALUE = 100000000;
const int MIN_VALUE = 1;
int main()
{
   vector<int> a(MAX_VALUE);
   int n,i;
   int elapsed_secs;
   unsigned seed = time(0);
   srand(seed);
   n = MAX_VALUE;
   for(i=0;i<n;i++)
            a[i]=bigRand(MAX_VALUE, MIN_VALUE);
```

QuickSort : Vector Version Using & C.A.R.Hoare sorting Algorithm

Part 1 of 4

```
    cout << "The clock has now started:" << endl << endl;
    auto started = chrono::high_resolution_clock::now();  // Starts the clock
    quickSort(a,0,n-1);
    auto done = chrono::high_resolution_clock::now();     // Stops the clock
    elapsed_secs = chrono::duration_cast<chrono::milliseconds>(done-started).count();
    cout << "Elapsed time = " << elapsed_secs << " milliseconds." << endl << endl;
    cout << "The first sorted element is: " << a[0] << endl;
    cout << "The middle sorted element is: " << a[MAX_VALUE/2] << endl;
    cout << "The final sorted element is: " << a[MAX_VALUE -1];
    cin.get();
    return 0;
}
void quickSort(vector<int> &set, int start, int end)
{
   int pivotPoint;
   if (start < end)
   {
      // Get the pivot point.
      pivotPoint = partition(set, start, end);
      // Sort the first sub list.
      quickSort(set, start, pivotPoint - 1);
      // Sort the second sub list.
      quickSort(set, pivotPoint + 1, end);
   }
}
```

QuickSort : Vector Version Using & C.A.R.Hoare sorting Algorithm

Part 2 of 4

```
//**********************************************************
// partition selects the value in the middle of the          *
// array set as the pivot. The list is rearranged so         *
// all the values less than the pivot are on its left        *
// and all the values greater than pivot are on its right.   *
//**********************************************************
int partition(vector<int> &set, int start, int end)
{
  int pivotValue, pivotIndex, mid;
  mid = (start + end) / 2;
  swap(set[start], set[mid]);
  pivotIndex = start;
  pivotValue = set[start];
  for (int scan = start + 1; scan <= end; scan++)
  {
    if (set[scan] < pivotValue)
    {
      pivotIndex++;
      swap(set[pivotIndex], set[scan]);
    }
  }
  swap(set[start], set[pivotIndex]);
  return pivotIndex;
}
```

QuickSort : Vector Version Using & C.A.R.Hoare sorting Algorithm

Part 3 of 4

QuickSort : Vector Version Using & C.A.R.Hoare sorting Algorithm

Part 4 of 4

```
//*********************************************
// swap simply exchanges the contents of                    *
// value1 and value2.                                        *
//*********************************************
void swap(int &value1, int &value2)
{
  int temp = value1;
  value1 = value2;
  value2 = temp;
}
int bigRand(int upper, int lower)
{
   const int twoToFifteenth = 32768;
   int rand_num, rand_num1, rand_num2, randnum;
   int top = 32768;
   rand_num1 = 1 + rand() ;//% top;  // Check this out
   rand_num2 = rand() ;//% top;
   rand_num = twoToFifteenth*rand_num1 + rand_num2;
   randnum = rand_num%(upper - lower) + lower;
   return randnum;
}
```

QuickSort : Vector Version Using Efficient Sorting Algorithm

Part 1 of 4

```
// QuickSort – Vector Version Using Efficient Sort Algorithm
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <vector>
#include <random>
#include <algorithm>
#include <chrono>
using namespace std;
void commaSeparate(int);
void quickSort(vector<int>&,int,int);
int partition(vector<int>&,int,int);
int bigRand(int, int);
const int MAX_VALUE = 100000000;
const int MIN_VALUE = 1;
int main()
{
   vector<int> a(MAX_VALUE);
   int n,i;
   int elapsed_secs;
   unsigned seed = time(0);
   srand(seed);
   n = MAX_VALUE;
   for(i=0;i<n;i++)
             a[i]=bigRand(MAX_VALUE, MIN_VALUE);
```

```
   cout << "The clock has now started:" << endl << endl;
   auto started = chrono::high_resolution_clock::now();  // Starts the clock
   quickSort(a,0,n-1);
   auto done = chrono::high_resolution_clock::now();     // Stops the clock
   elapsed_secs = chrono::duration_cast<chrono::milliseconds>(done-started).count();
   cout << "Elapsed time = " << elapsed_secs << " milliseconds." << endl << endl;
   cout << "The first sorted element is: " << a[0] << endl;
   cout << "The middle sorted element is: " << a[MAX_VALUE/2] << endl;
   cout << "The final sorted element is: " << a[MAX_VALUE -1];
   cin.get();
   return 0;
}

void quickSort(vector<int> &a,int start,int end)
{
   int j;
   if(start<end)
   {
      j=partition(a,start,end);
      quickSort(a,start,j-1);
      quickSort(a,j+1,end);
   }
}
```

Program 20-11 QuickSort : Vector Version Using Efficient sorting Algorithm

Part 2 of 4

QuickSort : Vector Version Using Efficient sorting Algorithm

Part 3 of 4

```
int partition(vector<int> &a,int start,int end)
{
   int v,i,j,temp;
   v=a[start];              // v = a[start] is the pivotpoint
   i=start;
   j=end+1;
   do
   {
      do
         i++;
      while(a[i]<v&&i<=end);  // while a[i] is less than pivotpoint
      do
         j--;
      while(v<a[j]);         // while a[j] is greater than pivotpoint
      if(i<j)
      {
         temp=a[i];         // since a[i] is greater than pivotpoint
         a[i]=a[j];          // and a[j] is less than the pivotpoint
         a[j]=temp;         // swapping the two will correct them both
      }
   }while(i<j);              // When i==j, we're done.
   a[start]=a[j];             // Swap the contents of a[j] with pivotpoint
   a[j]=v;
   return j;                  // j is now the final location for pivotpoint
}
```

QuickSort : Vector Version Using Efficient sorting Algorithm

Part 4 of 4

```
int bigRand(int upper, int lower)
{
   const int twoToFifteenth = 32768;
   int rand_num, rand_num1, rand_num2, randnum;
   int top = 32768;
   rand_num1 = 1 + rand() ;//% top;  // Check this out
   rand_num2 = rand() ;//% top;
   rand_num = twoToFifteenth*rand_num1 + rand_num2;
   randnum = rand_num%(upper - lower) + lower;
   return randnum;
}
```

QuickSort : Dynamic Memory Allocation Using C.A.R.Hoare's 1960 sorting Algorithm

Part 1 of 4

```
// QuickSort using Dynamic Memory Allocation & C.A.R.Hoare sorting Algorithm
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <chrono>
using namespace std;
void quickSort(int[],int,int);
int partition(int[],int,int);
int bigRand(int, int);
const int MAX_VALUE = 100000000;
const int MIN_VALUE = 1;
int main()
{
   int n,i;
   int *a = nullptr;
   a = new int[MAX_VALUE];
   int elapsed_msecs;
   unsigned seed = time(0);
   srand(seed);
   for(i=0;i<MAX_VALUE;i++)
         a[i]=bigRand(MAX_VALUE, MIN_VALUE);
```

```
    cout << "The clock has now started:" << endl << endl;
    auto started = chrono::high_resolution_clock::now();       // Starts the clock
    quickSort(a,0,MAX_VALUE-1);
    auto done = chrono::high_resolution_clock::now();          // Stops the clock
    elapsed_msecs = chrono::duration_cast<chrono::milliseconds>(done-started).count();
    cout << "Elapsed time = " << elapsed_msecs << " milliseconds." << endl << endl;
    cout << "The first sorted element is: " << a[0] << endl;
    cout << "The middle sorted element is: " << a[MAX_VALUE/2] << endl;
    cout << "The final sorted element is: " << a[MAX_VALUE -1];
    cin.get();
    delete [] a;
    a = nullptr;
    return 0;
}
int bigRand(int upper, int lower)
{
    const int twoToFifteenth = 32768;
    int rand_num, rand_num1, rand_num2, randnum;
    int top = 32768;
    rand_num1 = 1 + rand() ;//% top;  // Check this out
    rand_num2 = rand() ;//% top;
    rand_num = twoToFifteenth*rand_num1 + rand_num2;
    randnum = rand_num%(upper - lower) + lower;
    return randnum;
}
```

QuickSort : Dynamic Memory Allocation Using C.A.R.Hoare sorting Algorithm

Part 2 of 4

```
void quickSort(int set[], int start, int end)
{
   int pivotPoint;

   if (start < end)
   {
      // Get the pivot point.
      pivotPoint = partition(set, start, end);
      // Sort the first sub list.
      quickSort(set, start, pivotPoint - 1);
      // Sort the second sub list.
      quickSort(set, pivotPoint + 1, end);
   }
}
//**********************************************************
// partition selects the value in the middle of the              *
// array set as the pivot. The list is rearranged so             *
// all the values less than the pivot are on its left            *
// and all the values greater than pivot are on its right.       *
//**********************************************************
```

QuickSort : Dynamic Memory Allocation Using C.A.R.Hoare sorting Algorithm

Part 3 of 4

```
int partition(int set[], int start, int end)
{
  int pivotValue, pivotIndex, mid;
  mid = (start + end) / 2;
  swap(set[start], set[mid]);
  pivotIndex = start;
  pivotValue = set[start];
  for (int scan = start + 1; scan <= end; scan++)
  {
    if (set[scan] < pivotValue)
    {
      pivotIndex++;
      swap(set[pivotIndex], set[scan]);
    }
  }
  swap(set[start], set[pivotIndex]);
  return pivotIndex;
}
void swap(int &value1, int &value2)
{
  int temp = value1;
  value1 = value2;
  value2 = temp;
}
```

QuickSort : Dynamic Memory Allocation Using the C.A.R.Hoare sorting Algorithm

Part 4 of 4

```
// QuickSort using Dynamic Memory Allocation & Using Efficient Sort
Algorithm
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <chrono>
using namespace std;
void quickSort(int[],int,int);
int partition(int[],int,int);
int bigRand(int, int);
const int MAX_NUM =   100000000;
const int MAX_VALUE = 100000000;
const int MIN_VALUE = 1;
int main()
{
   int n,i;
   int *a = nullptr;
   a = new int[MAX_VALUE];
   int elapsed_msecs;
   unsigned seed = time(0);
   srand(seed);
   for(i=0;i<MAX_VALUE;i++)
         a[i]=bigRand(MAX_VALUE, MIN_VALUE);
   cout << "The clock has now started:" << endl << endl;
```

QuickSort : Dynamic Memory Allocation Version Using Efficient Sorting Algorithm

Part 1 of 4

```
 auto started = chrono::high_resolution_clock::now();          // Starts the clock
   quickSort(a,0,MAX_NUM-1);
   auto done = chrono::high_resolution_clock::now();          // Stops the clock
   elapsed_msecs = chrono::duration_cast<chrono::milliseconds>(done-started).count();
   cout << "Elapsed time = " << elapsed_msecs << " milliseconds." << endl << endl;
   cout << "The first sorted element is: " << a[0] << endl;
   cout << "The middle sorted element is: " << a[MAX_VALUE/2] << endl;
   cout << "The final sorted element is: " << a[MAX_VALUE -1];
   cin.get();
   delete [] a;
   a = nullptr;
   return 0;
}
int bigRand(int upper, int lower)
{
   const int twoToFifteenth = 32768;
   int rand_num, rand_num1, rand_num2, randnum;
   int top = 32768;
   rand_num1 = 1 + rand() ;//% top;  // Check this out
   rand_num2 = rand() ;//% top;
   rand_num = twoToFifteenth*rand_num1 + rand_num2;
   randnum = rand_num%(upper - lower) + lower;
   return randnum;
}
```

QuickSort : Dynamic Memory Allocation Version Using Efficient Sorting Algorithm

Part 2 of 4

QuickSort : Dynamic Memory Allocation Version Using Efficient Sorting Algorithm

Part 3 of 4

```
void quickSort(int a[],int start,int end)
{
   int j;
   if(start<end)
   {
      j=partition(a,start,end);
      quickSort(a,start,j-1);
      quickSort(a,j+1,end);
   }
}
```

QuickSort : Dynamic Memory Allocation Version Using Efficient Sorting Algorithm

Part 4 of 4

```
int partition(int a[],int start,int end)
{
   int v,i,j,temp;
   v=a[start];                    // v = a[start] is the pivotpoint
   i=start;
   j=end+1;
   do
   {
      do
         i++;
      while(a[i]<v&&i<=end);  // while a[i] is less than pivotpoint
      do
         j--;
      while(v<a[j]);                // while a[j] is greater than pivotpoint
      if(i<j)
      {
         temp=a[i];                 // since a[i] is greater than pivotpoint
         a[i]=a[j];                 // and a[j] is less than the pivotpoint
         a[j]=temp;                 // swapping the two will correct them both
      }
   }while(i<j);                     // When i==j, we're done.
   a[start]=a[j];                   // Swap the contents of a[j] with pivotpoint
   a[j]=v;
   return(j);                       // j is now the final location for pivotpoint
}
```

Laboratory 20B

- Load in each of the four sorting programs given to you.
- Run each of the programs 5 times recording the length of time it takes to sort 100 million pseudo-random numbers.
- Next compute the average of each five runs to get a reasonable estimate of the expected time each of the four methods take.
- Given the four programs listed in the following bullet point, answer questions 1-5:

Laboratory 20B

1. Which program takes the longest amount of time to do the sort?
2. Which program yields a sort in the shortest period of time?
3. What is the average percent improvement in execution time in using the efficient algorithm over using the original Hoare algorithm when using vectors?
4. What is the average percent improvement in execution time in using the efficient algorithm over using the original Hoare algorithm when using Dynamic Memory Allocation?
5. What is the average improvement in speed when using the Dynamic Memory Allocation array version compared to the vector version?