Name: Chad Huntebrinker

Answer the following Questions:

**Q1: [5 Points]** Write a function that sorts a linked list. (Use any sort algorithm, make sure to state which one you’re going to use). Provide the results of running your project. (Replace position with iterators).

|  |
| --- |
| Use this space to answer this question  //Chad Huntebrinker  //CS 303  //Using selection sort  #include <iostream>  #include <list>  #include <vector>  using namespace std;  int main() {  list<int> userList;  list<int> tempList;  //Fill linked list with values  userList.push\_back(6);  userList.push\_back(3);  userList.push\_back(19);  userList.push\_back(8);  userList.push\_back(14);  userList.push\_back(32);  userList.push\_back(28);  userList.push\_back(42);  userList.push\_back(12);  userList.push\_back(22);    //Setting up the iterators  list<int>::iterator begin1;  list<int>::iterator begin2;  list<int>::iterator temp;  list<int>::iterator minIndex;  list<int>::iterator end = userList.end();  //Implementing the selection sort  //The first loop is used to go through the whole list  for (begin1 = userList.begin(); begin1 != end; ++begin1) {  temp = minIndex = begin1;  //This second loop goes through the list to find the smallest item.  for (begin2 = ++temp; begin2 != end; ++begin2) {  if (\*begin2 < \*minIndex) {  minIndex = begin2;  }  }  //If the smallest item doesn't match up with the begining element found in the first loop, then  //we swap them.  if (minIndex != begin1) {  swap(\*begin1, \*minIndex);  }  }  //Output the list  for (begin1 = userList.begin(); begin1 != end; ++begin1) {  cout << \*begin1 << " ";  }  cout << endl << endl;  return 0;  }  Output: 3 6 8 12 14 19 22 28 32 42 |

**Q2: [5 Points]** for the following array:

30 5 40 11 20 9 15 2 60 25 80 3 73 35 4 75 20 6

Sort the array using the Sell sort algorithm, start with a gap equal to 8. Show the details of the process and No Code is required.

|  |
| --- |
| Use this space to answer this question  1)  After separating to different arrays: 30 , 5 , 40 , 11 , 20 , 9 , 15 , 2 , 60 , 25 , 80 , 3 , 73 , 35 , 4 , 75 , 20 , 6  30, 60, 20 -> 20, 30, 60 (Sort each mini array and then insert them back in  5, 25, 6 -> 5, 6, 25 at their spots)  40, 80 -> 40, 80  11, 3 -> 3, 11  20, 73 -> 20, 73  9, 35 -> 9, 35  15, 4 -> 4, 15  2, 75 -> 2, 75  After Sorting: 20 , 5 , 40 , 3 , 20 , 9 , 4 , 2 , 30 , 6 , 80 , 11 , 73 , 35 , 15, 75 , 60 , 25  2) Sort with a gap of 4  Before sorting: 20 , 5 , 40 , 3 , 20 , 9 , 4 , 2 , 30 , 6 , 80 , 11 , 73 , 35 , 15 , 75 , 60 , 25  After sorting: 20 , 5 , 4 , 2 , 20 , 6 , 15 , 3 , 30 , 9 , 40 , 11 , 60 , 25 , 80 , 75 , 73 , 35  3) Sort with a gap of 2  Before sorting: 20 , 5 , 4 , 2 , 20 , 6 , 15 , 3 , 30 , 9 , 40 , 11 , 60 , 25 , 80 , 75 , 73 , 35  After sorting: 4 , 2 , 15 , 3 , 20 , 5 , 20 , 6 , 30 , 9 , 40 , 11 , 60 , 25 , 73 , 35 , 80 , 75  4) Final sort  Before sorting: 4 , 2 , 15 , 3 , 20 , 5 , 20 , 6 , 30 , 9 , 40 , 11 , 60 , 25 , 73 , 35 , 80 , 75  After final sort: 2 , 3 , 4 , 5 , 6 , 9 , 11 , 15 , 20 , 20 , 25 , 30 , 35 , 40 , 60 , 73 , 75 , 80 |

**Q3: [5 Points]** Trace the execution of the call mystery(6) for the following recursive function. What does this function do?

int mystery (int n) {

if (n == 0)

return 0;

else

return n \* n + mystery (n – 1);

}

Use a stack (table) to trace the execution and show the details of the recursion process

|  |
| --- |
| Use this space to answer this question  Start with: 6  Stack after 1: [6 \* 6 + mystery(6-1)]  /\*After first iteration, the stack has 36 + mystery(n-1) because it didn’t meet the base case of n == 0\*/  Stack after 2: [36 + mystery(6-1) | 5 \* 5 + mystery(5-1)]  /\*Still hasn’t met the base case, so the function does another recursive call only with 4 as n\*/  Stack after 3: [36 + mystery(6-1) | 25 + mystery(5-1) | 4 \* 4 + mystery(4-1)]  /\*Still hasn’t met the base case, so the function does another recursive call only with 3 as n\*/  Stack after 4: [36 + mystery(6-1) | 25 + mystery(5-1) | 16 + mystery(4-1) | 3 \* 3 + mystery(3-1)]  /\*Once again, it still hasn’t met the base case, so the function does another recursive call only with 2 as n\*/  Stack after 5: [36 + mystery(6-1) | 25 + mystery(5-1) | 16 + mystery(4-1) | 9 + mystery(3-1) | 2 \* 2 + mystery(2-1)]  /\*I know what you’re thinking, “Surely it must have reached the base case by now?” And that is where you would be wrong because it still hasn’t met the base case, so the function does another recursive call only with 1 as n\*/  Stack after 6: [36 + mystery(6-1) | 25 + mystery(5-1) | 16 + mystery(4-1) | 9 + mystery(3-1) | 4 + mystery(2-1) | 1 \* 1 + mystery( 1-1)]  /\*Yep, it still hasn’t met the base case. So, the function does another recursive call only with 0 as n \*/  Stack after 7: [36 + mystery(6-1) | 25 + mystery(5-1) | 16 + mystery(4-1) | 9 + mystery(3-1) | 4 + mystery(2-1) | 1 + mystery( 1-1) | 0]  /\*Buckle up everybody, we have reached the base case! Now, the function returns 0 and goes back to the previous function. It’s going to be like when you stretch out a tape measure, let it go, and it is now accelerating back towards the container.\*/  [36 + mystery(6-1) | 25 + mystery(5-1) | 16 + mystery(4-1) | 9 + mystery(3-1) | 4 + mystery(2-1) | 1 + 0]  /\* 0 is returned to the mystery(1-1) function and we now have 1 + 0 \*/  [36 + mystery(6-1) | 25 + mystery(5-1) | 16 + mystery(4-1) | 9 + mystery(3-1) | 4 + 1]  /\* 1 is returned to the mystery(2-1) function and we now have 4 + 1 \*/  [36 + mystery(6-1) | 25 + mystery(5-1) | 16 + mystery(4-1) | 9 + 5]  /\* 5 is returned to the mystery(3-1) function and we now have 9 + 5 \*/  [36 + mystery(6-1) | 25 + mystery(5-1) | 16 + 14]  /\* 14 is returned to the mystery(4-1) function and we now have 16 + 14 \*/  [36 + mystery(6-1) | 25 + 30]  /\* 30 is returned to the mystery(5-1) function and we now have 25 + 30 \*/  [36 + 55]  /\* 55 is returned to the mystery(6-1) function and we now have 36 + 55 \*/  [91]  91 is the answer. |

**Q4: [10 Points]**

1. Use the random number generator to generate 1000 random integer values. Use these 1000 integers to create 3 arrays.
2. Modify the bubble sort algorithm by adding a counter to count how many times the elements of the array changed value (swapping is considered as a one change)
3. For each array, apply the modified bubble sort algorithms and save the counter value.
4. Repeat step 1 to 4, using shell sort algorithm, quick sort algorithm and merge sort algorithm.

Use the results to complete the table below:

|  |  |  |  |
| --- | --- | --- | --- |
|  | 1st array counter | 2nd array counter | 3rd array counter |
| Bubble sort | 255,112 | 249,720 | 256,216 |
| Shell sort | 15,511 | 16,664 | 16,117 |
| Quick sort | 5,625 | 5,703 | 6,094 |
| Merge sort | 9,382 | 9,379 | 9,358 |

State your observations and conclusion in the space provided below

|  |
| --- |
| Use this space to answer this question  There are two things we can understand from this table. First, to state the obvious, we see that Quick sort has the least amount of swapping (with an average of 5,807.33 swaps) followed by Merge sort (average of 9,373 swaps), then Shell sort (average of 16,097.33 swaps), with Bubble sort (average of 253,682.67 swaps) in last. The second and less obvious fact we can see from the table is the variation of the data. Merge sort’s three numbers only vary in the 10’s of swaps unlike Bubble sort which varies in the 1,000’s of swaps. So if we wanted to make a new ranking with this in mind, we would see:   1. Merge sort 2. Quick sort 3. Shell sort 4. Bubble sort   In conclusion, the best sorter is between Merge sort and Quick sort. There’s no real purpose to go with Shell sort over Merge sort because Merge sort is an improved version of Shell sort. And Bubble sort is useless with larger amounts of data. So, it comes down to Merge sort, Quick sort, and what you need for your data. If your data is mostly sorted and/or you want a sorting system that is consistent in the number of swaps it does, you would want to go with Merge sort. If your data isn’t sorted and/or you want a sorting system with the least amount of swaps, you would want to go with Quick sort.  //Chad Huntebrinker  //CS 303  #include <iostream>  #include <time.h>  using namespace std;  //The four sorting methods  void bubbleSort(int \*array, int size, int &bubbleTotal);  void shellSort(int \*array, int size, int &shellTotal);  void quickSort(int \*array, int low, int high, int &quickTotal);  void mergeSort(int \*array, int low, int high, int &mergeTotal);  int main() {  int randArray1[1000], randArray2[1000], randArray3[1000], randArray4[1000];  int bubbleTotal = 0, shellTotal = 0, quickTotal = 0, mergeTotal = 0;  srand(time(NULL));  //Have one array store the random numbers, then have that array store it in three other arrays so  //each function will have the same elements in an array.  for (int i = 0; i < 1000; ++i) {  randArray1[i] = rand() % 100000;  randArray2[i] = randArray1[i];  randArray3[i] = randArray1[i];  randArray4[i] = randArray1[i];  }  //Calls to the four different functions  bubbleSort(randArray1, 1000, bubbleTotal);  shellSort(randArray2, 1000, shellTotal);  quickSort(randArray3, 0, 999, quickTotal);  mergeSort(randArray4, 0, 999, mergeTotal);  //Output the totals  cout << "Bubble total swaps: " << bubbleTotal << endl;  cout << "Shell total swaps: " << shellTotal << endl;  cout << "Quick total swaps: " << quickTotal << endl;  cout << "Merge total swaps: " << mergeTotal << endl;  }  //This method does the bubble sort.  void bubbleSort(int \*array, int size, int &bubbleTotal) {  for (int i = 0; i < (size - 1); i++) {  for (int j = 0; j < (size - i - 1); j++) {    //If spot lower is greater than the spot higher  if (array[j] > array[j + 1]) {  swap(array[j + 1], array[j]); //Swap the elements.  ++bubbleTotal; //Increase the bubble sort swap counter.  }  }  }  }  //Shell sort function  void shellSort(int \*array, int size, int &shellTotal) {  for (int gapSize = size / 2; gapSize > 0; gapSize = gapSize / 2) {  for (int i = gapSize; i < size; i += 1) {  int temp = array[i];  int j;  for (j = i; j >= gapSize && array[j - gapSize] > temp; j -= gapSize) {  array[j] = array[j - gapSize];  ++shellTotal; //Increase the shell sort swap total  }  array[j] = temp;  ++shellTotal; //Increase the shell sort swap total  }  }  }  void quickSort(int \*array, int low, int high, int &quickTotal) {  if (low < high) {  int pivot = array[high];  int temp = (low - 1); //temp is the index of smaller element.  for (int i = low; i <= high - 1; i++)  {  //If current element is smaller than the pivot.  if (array[i] < pivot)  {  ++temp; //Increase index of smaller element.  swap(array[temp], array[i]); //Swap the two places.  ++quickTotal; //Increase the quick sort swap total  }  }  ++temp; //Increase temp by one so you can swap the pivot and a number that is greater than the pivot.  swap(array[temp], array[high]);  ++quickTotal; //Increase the quick sort swap total  //Call the function again, ranging from the lowest position to 1 before the pivot because the pivot  //is now in the right spot. The second function is from 1 after the pivot to the highest position.  quickSort(array, low, temp - 1, quickTotal);  quickSort(array, temp + 1, high, quickTotal);  }  }  void mergeSort(int \*array, int low, int high, int &mergeTotal) {  if (low < high) {  int middle = low + (high - low) / 2;  //Call the same function multiple times so you can split the original array to smaller arrays.  mergeSort(array, low, middle, mergeTotal);  mergeSort(array, middle + 1, high, mergeTotal);  //Get the size of the two different arrays.  int size1 = middle - low + 1;  int size2 = high - middle;  //Create new temp arrays  int\* tempLow = new int[size1];  int\* tempHigh = new int[size2];  //Copy data to temp arrays tempLow[] and tempHigh[]  for (int i = 0; i < size1; i++)  tempLow[i] = array[low + i];  for (int j = 0; j < size2; j++)  tempHigh[j] = array[middle + 1 + j];  //Combine the temp arrays back into the orginal array  int initial1 = 0; //Initial index of first subarray  int initial2 = 0; //Initial index of second subarray  int initialMerged = low; //Initial index of merged subarray  //This loop will continue going while inital index of first array is smaller than the initial index of  //of the second array and while the initial index of the second array is smaller than than the size of  //the second subarray  while (initial1 < size1 && initial2 < size2)  {  //If element in tempLow is smaller than element in tempHigh  if (tempLow[initial1] <= tempHigh[initial2])  {  array[initialMerged] = tempLow[initial1];  ++initial1;  ++mergeTotal; //Increase merge swap total  }  else  {  array[initialMerged] = tempHigh[initial2];  ++initial2;  ++mergeTotal; //Increase merge swap total  }  ++initialMerged;  }  //Insert the remaining elements of tempLow  while (initial1 < size1)  {  array[initialMerged] = tempLow[initial1];  ++initial1;  ++initialMerged;  ++mergeTotal;  }  //Insert the remaining elements of tempHigh  while (initialMerged < size2)  {  array[initialMerged] = tempHigh[initial2];  ++initial2;  ++initialMerged;  ++mergeTotal;  }  //Delete the temp arrays  delete[] tempLow;  delete[] tempHigh;  }  } |