CST8130 – Data Structures

Professor : Dr. Anu Thomas
Email: thomasa@algonquincollege.com
Office: T314

Sorting

Sorting

- Bubble Sort
- Selection Sort
- Insertion Sort
- Merge Sort
- Quick Sort

Bubble Sort

Algorithm Essentials:

-Conceptually, divide the list into sorted and unsorted parts

-Outer Loop

- Each pass results in next largest item bubbling to top
- One item becomes part of sorted part of list
- Inner Loop
    - Compare adjacent items in unsorted portion of list
    - Exchange items it item on left is larger than item on right
    - Exchange requires three assignments and use of a temporary variable

Bubble Sort

Efficiency Observations

-Each movement of one element from the unsorted part into the sorted part is considered a pass

-Given *n* elements, needs *n-1* passes

-Each pass requires scan of all remaining unsorted elements

-Efficiency: Two nested counting loops O(n^2)

Bubble Sort

Algorithm:

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n-i-1; j++) {
        if (list[j] > list[j+1]) {
            int temp = list[j];
            list[j] = list[j+1];
            list[j+1]=temp;
        }
    }
}
```

Bubble Sort

Efficiency Improvements:

- In inner loop – set a flag if no exchanges are made…list is sorted already – can eliminate some passes.

```
bool ChangesMade = true;
for (int i = 0; i < n && ChangesMade; i++) {
     ChangesMade = false;
    for (int j = 0; j < n-i-1; j++) {
          if (list[j] > list[j+1]) {
               int temp = list[j];
               list[j] = list[j+1];
               list[j+1]=temp;
               ChangesMade = true;
         }
     }
}
```

Selection Sort

Algorithm Essentials:

-Conceptually, divide the list into sorted and unsorted parts. Search the unsorted side for the smallest item, move it to the sorted side.

-**Initial State**: list is entirely unsorted.

-**Outer Loop**

- Manages the boundary between sorted and unsorted
- Arbitrarily, the first item in unsorted side is identified as smallest item (so far). The smallest is identified by using its index position in the array.
- Inner Loop
  - Examine all subsequent items in the unsorted side, looking for an item that is smaller than the currently indexed smallest item. If none is found, set the index to smallest to the newly discovered smallest value
  - No data movement in inner loop – only setting index to smallest
- Swap item at index to smallest with next location in sorted side, proceed to next iteration of outer loop

Selection Sort

Efficiency Observations

- Each movement of one element from the unsorted part into the sorted part is considered a pass
- Given *n* elements, needs *n-1* passes
- Each pass requires scan of all remaining unsorted elements
- Efficiency: Two nested counting loops O(n^2)

Selection Sort

Algorithm:

```
for (int current = 0; current < n; current++) {
     smallest = current;  // start smallest at next
    for (int j = current+1; j < n; j++) {
          if (list[j] < list[smallest]) {
               smallest = j;
         }
     }
     int temp=list[current];
     list[current]=list[smallest];
     list[smallest]= temp;
}
```

Insertion Sort

Algorithm Essentials:

-Conceptually, divide the list into sorted and unsorted parts.

-**Initial State**: First item is deemed to be in correct order (since there is only one item).

-**Outer Loop**

- Select the first item in the list of remaining unsorted elements. Put this element into a temporary storage location
- Inner Loop
    - Search the sorted list for the correct insertion point
    - With each comparison, another element in the sorted list is shifted one position to the right in the array to make room for the item to be inserted
- Move the element from temporary storage location to the correct insertion point

Insertion Sort

Efficiency Observations

- Each movement of one element from the unsorted part into the sorted part is considered a pass
- Given *n* elements, needs *n-1* passes
- Each pass requires scan of all remaining unsorted elements
- Efficiency: Two nested counting loops O(n^2)

Insertion Sort

Algorithm:

You will develop this in your assignment….

Quick Sort

Algorithm Essentials:

-Find the pivot and partition the data set:

- Assume data set has indexes ranging from *start* to *end*
- Select index of first element as *pivot*
- Create two index counters, *left* (at *start* +1) and *right* (at *end*)
- Scan and exchange while *left* index is less than *right* index
  - Find *left* item larger than *pivot* (while *left* index < *right* index)
  - Find *right* item smaller than *pivot* (while *left* index < *right* index)
  - If *left* and *right* indexes have not met (ie we are still partitioning),
    - Swap left and right items
    - Increment left and decrement right
- Move the pivot to its correct final position

-Recursively call quicksort to sort left side of pivot, then right side of pivot.

Quick Sort

Efficiency Observations

- In the best case, we are dividing the list in half – that gives us logarithmic efficiency in the number of divisions needed
- Within each division though, we basically look through almost all of the items

- O($n\ log_2\ n$)

Quick Sort - *algorithm*

```
void quicksort ( int[] list, int start, int end) {
   if (start >= end) return;
   int pivot = start, left = start+1,right=end;
   while (left < right) {           // partition
            while (list[left] < list[pivot] && left <right)  left ++;
            while (list[right] > list[pivot] && left < right) right --;
            int temp = list [left];     // swap
            list[left] = list[right];
            list[right] = temp;
            if (left < right) {
                  left++; right --;
            }
      }
      if (list[left] > list[pivot]){                 // move pivot and call recursively
            int temp = list[pivot];
            list[pivot] = list[left-1];
            list[left-1] = temp;
            quicksort (list, start, left-2);
            quicksort (list, left, end);
      } else {
            int temp = list[pivot];
            list[pivot] = list[left];
            list[left] = temp;
            quicksort (list, start, left-1);
            quicksort (list, left+1, end);
      }
```

Merge Sort

Algorithm Essentials:

- Conceptually, list is divided into two halves (each sorted).
- The two halves are merged together into temporary storage
  - Compare and copy smaller item from each half until one data set is exhausted
  - Copy remainder of left data set (if any)
  - Copy remainder of right data set (if any)
- The temporary storage is copied back to original location

Merge Sort

Efficiency Observations

- In the best case, we are dividing the list in half – that gives us logarithmic efficiency in the number of divisions needed
- Within each division though, we basically look through almost all of the items

- Efficiency: O($n \; log_2 \; n$)

Merge Sort

Algorithm: given Array, int start, int end

1. Finds midpoint…………..midpoint = (end-start)/2
2. Calls *recursively* for both sides of midpoint:

   Merge (Array, start, midpoint);

   Merge (Array, midpoint + 1, end);
3. *Merge()* result

   Create temporary storage

   Compare (i through start….midpoint) and (j through midpoint+1….end) - copying smaller item into temporary array (k from 0…..?) until one data set exhausted.

   Copy remainder of left data set (if any)

   Copy remainder of right data set (if any).

   Copy data from temporary storage back to original location.

Questions?