Analysis of Sorting Algorithms

MAT 102 - Data Structures
Friday, April 18, 2003
Selection Sort

• The order of the selection sort algorithm is $O(n^2)$.

• The selection sort requires $O(n^2)$ major comparisons, but only $O(n)$ major data moves (in the swap function).

• Thus, the selection sort might be appropriate in a program where swapping items is costly when compared to comparisons.
Bubble Sort

• The bubble sort compares adjacent items and swaps them if they are out of order.
• Somewhat intuitive, but not particularly efficient.
• The bubble sort algorithm is of order $O(n^2)$. 
// sample code of a bubble sort algorithm

void bubbleSort(int a[], int n)
{
    bool sorted = false;
    int pass, index, nextIndex;

    for(pass = 1; (pass < n) && !sorted; pass++)
    {
        sorted = true;

        for(index = 0; index < n-pass; index++)
        {
            nextIndex = index + 1;

            if(a[index] > a[nextIndex])
            {
                swap(a[index], a[nextIndex]);
                sorted = false;
            }
        }
    }
}
Analysis of Bubble Sort

• The for loop for pass goes from 1 to (n-1).
• The for loop for index goes from 0 to (n-pass).
• This gives a total of (n-1) + (n-2) + ... + 2 + 1 passes. This sum is equal to \((n*(n-1))/2\).
• Inside the second for loop there is a major comparison and a swap.
• Each swap requires 3 major assignments, so there are 4 major operations inside the second loop.
• Thus, there are \(4*(n*(n-1))/2 = 2*(n^2 - n) = 2n^2-2n\) operations.
• Thus, the bubble sort algorithm is of order \(O(n^2)\).
Insertion Sort

• The insertion sort divides an array into two regions: the sorted region and the unsorted region.
• At first, the sorted region is just the first element and the unsorted region is the rest of the array.
• The process is to take the first element in the unsorted region and insert it into the sorted region in its proper place.
• This insertion will require shifting elements to make room for the new item.
• Each insertion increases the size of the sorted region by one and decreases the size of the unsorted region by one.
// sample code for insertion sort

void insertionSort(int a[], int n)
{
    int unsorted;  // first index of the unsorted region
    int loc;       // index of insertion in the sorted region
    int nextItem;  // next data item in the unsorted region

    for(unsorted = 1; unsorted < n; unsorted++)
    {
        nextItem = a[unsorted];
        loc = unsorted;

        // this for loop determines the appropriate
        // location to insert nextItem
        for(; (loc > 0) && (a[loc-1] > nextItem); loc--)
            a[loc] = a[loc-1];

        a[loc] = nextItem;
    }
}
Analysis of Insertion Sort

• The outer loop runs \((n-1)\) times and contains two assignments, one before and one after the inner loop.
• The inner loop contains a comparison and an assignment.
• The inner loop runs \(1+2+3+\ldots+(n-1)\) times.
• Operations involved are
  \[
  2 \times (n-1) + 2 \times (1+2+3+\ldots+(n-1))
  = 2n - 2 + 2 \times \left( \frac{n \times (n-1)}{2} \right)
  = 2n - 2 + \frac{n^2 - n}{2}
  = \frac{n^2 + n - 2}{2}
  \]
• Thus, the order of the insertion sort algorithm is \(O(n^2)\).
Merge Sort

• The merge sort algorithm is a recursive sorting algorithm that is more efficient than the previous sorting algorithms.

• The merge sort algorithm always takes the same amount of time, regardless of the initial order of the array items.

• Thus, the worst-case and best-case scenarios are the same for the merge sort.
// sample code for mergesort algorithm

void mergesort(int a[], int first, int last) {
  if (first < last) {
    int mid = (first + last) / 2;

    // sort the two halves of the array
    mergesort(a, first, mid);
    mergesort(a, mid+1, last);

    // merge the two halves
    merge(a, first, mid, last);
  }
}
// merge function, used in mergesort algorithm

const int MAT_SIZE = maximum-size-of-array;
void merge(int a[], int first, int mid, int last)
{
    int tempA[MAX_SIZE];

    int first1 = first; int last1 = mid;
    int first2 = mid + 1; int last2 = last;

    int index = first1;
    for(; (first1 <= last1) && (first2 <= last2); index++) {
        if(a[first1] < a[first2]) {
            tempA[index] = a[first1]; // this loop merges the
            first1++;
        } else{
            tempA[index] = a[first2]; // two arrays, using at most
            first2++;
        }
    }

    for(; first1 <= last1; first1++, index++) // finish off first array
        tempA[index] = a[first1];

    for(; first2 <= last2; first2++, index++) // finish off second array
        tempA[index] = a[first2];

    for(index = first; index <= last; index++) // move items from temp array
        a[index] = tempA[index]; // back to original array
}
Analysis of MergeSort

• Start by analysing merge.
  – (n-1) comparisons
  – n moves to the temp array
  – n moves back to the original array
  – 3*n - 1 operations for each call to merge

• Mergesort calls itself recursively. If n = 2^k, then the recursion goes k = \log_2 n levels deep.