Efficiency and Sorting

MAT 102 - Data Structures
Wednesday, April 16, 2003
Algorithm Analysis

- Algorithm A is considered to be more efficient than algorithm B if a solution that uses algorithm A consistently runs significantly faster than a solution that uses algorithm B.
- Note: Algorithm analysis sometimes also considers space requirements, though the examples we will do will not take space requirements into account.
- There are some important things to be aware of when comparing algorithms. There are external factors that could affect the speed of execution.
External Factors

• **How is the algorithm coded?**
  – It could be that algorithm A is running faster than algorithm B because algorithm B is coded improperly.

• **What computer is being used?**
  – Obviously when comparing algorithms, you should run both on the same computer.
  – In addition, you should be aware that some computers are specifically designed to improve the efficiency of certain tasks.
  – If one algorithm is using a task that has been optimized on the platform, it may be more efficient for that platform, but not necessarily more efficient in general.
External Factors

- Test data should be appropriately chosen.
  - If you choose poor test data, you can get misleading results when comparing algorithms.
  - Example: If you are comparing algorithms to search a sorted array, if you always search for the smallest element, certain techniques will have an advantage (sequential search vs. binary search).
Execution Time

- Suppose we want to traverse a pointer based linked list with \(n\) elements using the code:

```c
Node *cur = head;              // 1 assignment
while (cur != NULL)            // n+1 comparisons
{
    cout << cur->item << endl; // n writes
    cur = cur->next;           // n assignments
}
```

- We will have executed \(3n+1\) operations in the course of printing the \(n\) elements of this list. Since this is proportional to \(n\), we say that this algorithm has an order of \(n\).
- We denote the order using BigO Notation. We say that the algorithm above is \(O(n)\), for order \(n\).
Common Orders

• These common orders are organized from fastest to slowest.
  – $O(1)$, solution does not depend on $n$
  – $O(\log_2 n)$, logarithmic growth in time as $n$ grows
  – $O(n)$, linear growth in time as $n$ grows
  – $O(n \log_2 n)$
  – $O(n^2)$, quadratic growth
  – $O(n^3)$, cubic growth
  – $O(2^n)$, exponential growth
Order is just a rough estimate

- In the linked list example above, the number of operations was $3n+1$. We could have said that the order was $O(3n+1)$ or just $O(n)$.
- Notice that this means that we would consider two algorithms to be of the same order even if one takes three times as long to complete as the other.
Sorting Algorithms and Their Efficiency
Various algorithms

- Suppose we wish to sort the elements of an array. There are a variety of algorithms we can use.
  - Selection Sort
  - Bubble Sort
  - Insertion Sort
  - Merge Sort
  - Quick Sort
  - Others: Radix Sort, Tree Sort, Heap Sort
Selection Sort

• Suppose we have an array of n elements a[0] to a[n-1].
• Start by finding the largest item in the entire array, the swap the largest item with the item in position n-1 (the last item).
• Next, find the largest item among a[0] to a[n-2] and swap that item with the item in position n-2.
• Continue this until you need to find the largest item among a[0] and a[1], then put that item at position 1.
• At this point, the array is sorted.
// selection sort, sample code, array a[n]

// indexOfLargest(a, size) returns the index
// of the largest element between index=0 and
// index=size-1

// swap(a[i], a[j]) swaps elements in the array
// arguments are passed by reference so they can
// be modified

void selectionSort(int a[], int n)
{
    int last, largest;

    for(last = n-1; last >= 1; --last)
    {
        largest = indexOfLargest(a, last+1);
        swap(a[largest], a[last]);
    }
}
// function indexOfLargest, used by selection sort code

int indexOfLargest(const int a[], int size)
{
    int indexSoFar = 0;
    int i;

    for(i=0, i < size; i++)
    {
        if(a[i] > a[indexSoFar])
            indexSoFar = i;
    }
    return indexSoFar;
}

// swap function
void swap(int &x, int &y)
{
    int temp;

    temp = x;
    x = y;
    y = temp;
}
Analysis of Selection Sort

- for loop in selectionSort runs (n-1) times
- indexOfLargest contains (size-1) comparisons and assignments.
- Since size moves from n-1 down to 1, the operations called by indexOfLargest number
  \[2 \times (n-1) + 2 \times (n-2) + 2 \times (n-3) + \ldots + 2 \times 1 = n \times (n-1)\]
- Each call to swap requires 3 assignments, since swap is called (n-1) times, this gives 3*(n-1) operations.
- There are a total of
  \[n \times (n-1) + 3(n-1) = n^2 - n + 3n - 1 = n^2 + 2n - 1\] operations.
- So, this algorithm is of order \(O(n^2 + 2n - 1) = O(n^2)\)