Internal Sorting

Objectives

- to present the most important sorting algorithms
- to compare the performance of the algorithms presented
- to analyze the sorting problem

The sorting Problem

Definition. Given a set of records $r_1, r_2, r_3, \ldots, r_n$ with key values $k_1, k_2, k_3, \ldots, k_n$, arrange the records into sequence $r_{i1}, r_{i2}, r_{i3}, \ldots, r_{in}$ such that the keys are ordered, that is $k_{i1} <= k_{i2} <= k_{i3} <= \ldots <= k_{in}$.

- we shall represent the records by their key values
- analysis: comparisons & swaps

Example to be used:

- sort the records with the following key values:

  5, 10, 7, 12, 4, 9, 6, 3, 15, 11

Quadratic Time Sorting

1.Insertion Sort

The idea:

- assume that *i* elements are already sorted and insert the *i+1* -st element into this sorted sequence
- perform this step for *i* from *0* to *n-1*

```
void InsSort(Element* array, int n){
    for(int i=1; i<n; i++){
        for(int j=i;
            (j>0)&&(array(j)<array(j-1)); j--)
         swap(array(j), array(j-1));
    }
}
```

Analysis:

Comparisons:

- worst case performance of insertion sort: $\Theta(n^2)$
- average case performance of insertion sort: $\Theta(n^2)$
- best case performance of insertion sort: $\Theta(n)$

Swaps:

- the number of swaps is $n-1$ less than the number of comparisons

2. Bubble Sort

The idea:

- keep swapping neighboring elements that are not in sequence till the set is sorted

```
void BubbSort(Element* array, int n){
    for (int i=0; i<n-1; i++)
        for (int j=n-1; j>i; j--)
            if (array(j) < array(j-1))
                swap(array(j), array(j-1));
}
```

Analysis:

Comparisons:

- worst case, average case and best case performance of bubble sort are: $\Theta(n^2)$

Swaps:

- the number of swaps will be identical to that performed by insertion sort

3. Selection Sort

The idea:

- select the smallest key and place it in the first position
- repeat this step for the remaining keys

```
void SelSort(Element* array, int n){
    for (int i=0; i<n-1; i++){
        int low = i;
        for (int j=n-1; j>i; j--)
            if (array(j)<array(low))
                low = j;
        swap(array(i), array(low));
    }
}
```

Analysis:

Comparisons:

- same as for bubble sort: $\Theta(n^2)$

Swaps:

- is $\Theta(n)$

Comparing quadratic sort algorithms

	Insertion	Bubble	Selection
Comparisons:			
Best	n	n^2	n^2
Average	n^2	n^2	n^2
Worst	n^2	n^2	n^2
Swaps:			
Best	0	0	n
Average	n^2	n^2	n
Worst	n^2	n^2	n

Fast Internal Sorting

1.Shellsort

The idea:

- break the list into sublists, sort those sublists and reassemble them into a complete list
- the sublists are “conceptual” - not contiguous portions of the original list

```
void ShellSort(Element* array, int n){
    for (int i=n/2; i>2; i/=2)
        for (int j=0; j<i; j++)
            VarInsertSort(&array(j),n-j,i);
        VarInsertSort(array,n,1);
}
```

```
void InsSort(Element *array, int n, int inc){
    for (int i=inc; i<n; i+=inc)
        for (int j=i; (j>=inc)&&
                (array[j]<array[j-inc]); j-=inc)
            swap(array[j], array[j-inc]);
}
```

- the analysis of Shellsort is difficult
- average performance is $O(n^{1.5})$

2. Quicksort

The idea:

- partition the array around a pivot selected randomly such that all the elements less than the pivot will be to its left and all the elements greater than the pivot will be to its right.
- recursively apply the Quicksort method for the subarrays defined by the pivot

```
void QSort(Element* array, int i, int j){
    int pivot = getpivot(i,j);
    swap(array[pivot],array[j]);
    int k = partition(array,i-1,j,array[j]);
    swap(array[k],array[j]);
    if ((k-i) > 1) QSort(array,i,k-1);
    if ((j-k) > 1) Quicksort(k+1,j);
}
```

- selecting the pivot can be done in many ways

```
int part(Element* array, int l, int r, int p){
    do {
        while (array[++l] < pivot);
        while (r && array[--r] > pivot);
        swap(array[l], array [r]);
    } while (l<r);
    swap(array[l], array[r]);
    return l;
}
```

- the partition function takes linear time
- the worst case for Quicksort happens when the pivot generates a bad partition each time it is called
- the average case performance of Quicksort can be computed from the recurrence relation

$$T(n) = cn + \frac{1}{n}\sum_{k=0}^{n}(T(k) + T(n-1-k))$$

Improvements for Quicksort

- improve the pivot selection method (e.g. median of three)
- for small number of elements replace Quicksort with a faster algorithm
- when the array is almost sorted (experiments show ~ 9 elements) use Insertion Sort

3. Mergesort

The idea:

- split the array into two equal subarrays
- recursively apply the same algorithm to the subarray then merge the results

```
void MSort(Element* array, int l, int r){
    Element* temp = new Element(r - l +1);
    if (l == r)
        return;
    else{
        int mid=(left+right)/2;
        MergeSort(array,left,mid);
        MergrSort(array,mid+1,right);
    merge_halves(array,r-l+1);
    }
}
```

4. Heapsort

The idea:

- use a max-heap
- heapify the list of inputs and then remove the elements one at a time

```
void HeapSort(Element* array, int n){
    heap H(array, n, n);
    for (int i=0; i<n; i++)
        H.removemax();
}
```

- Heapsort takes *nlgn* time in the worst, average and best cases

Non-comparison sorting

1. Binsort

The idea:

- use the key values to assign the records to a finite number of bins
- assuming that the values to be fall into the range 0..MaxKey

```
void Binsort(Element* array, int n){
    list Bin(MaxKey);
    for (int i=0; i<n; i++)
        Bin(array(i)).append(array(i));
    for (int i=0; i<MaxKey; i++)
        for (Bin(i).first(); Bin(i).isLast();
                                Bin(i).next())
            output(B(i).currValue());
}
```

- while the time required by the algorithm seems linear, it also depends on the value of MaxKey

2. **Bucket sort**

- generalization of Binsort where each bin (bucket) had associated a range of keys
- the records stored in a are sorted with some other sorting method
- the computation time for bucket sort is linear

3. **Radix sort**

- the computation of the buckets is done by modulo some radix

```
void RadSort(Element *A, Element B*,
        int n, int k, int r, Element * count){
    for (int i=0, rtok = 1; i<k; i++, rtok *= r)
        for (int j=0; j<r; j++)
            count(j) = 0;
        for (int j=0; j<n; j++)
            count((A(j)/rtok)%r)++;
        for (int j=1; j<r; j++)
            count(j) = count(j-1) + count(j);
        for (int j=n-1; j>=0; j--)
            B(--count((A(j)/rtok)%r)) = A(j);
        for(int j=0; j<n; j++) A(j) = B(j);
    }
}
```

Lower Bounds on Sorting

- we are going to study the **complexity of the sorting problem** as opposed to a specific algorithm
- **upper bound** for a problem: the asymptotic cost of the fastest algorithm that solves the problem
- **lower bound** for a problem: the best possible efficiency that any algorithm could achieve

Theorem. No sorting algorithm based on key comparison can be faster than $\Omega(n)$ in the worst case

Proof. The idea:

- any sorting algorithm can be modeled as a binary decision tree whose internal nodes correspond to comparisons
- the minimum number of leaves of a decision tree with n internal nodes is *n!* (the number of permutations which could yield the sorted input)
- the tree must have at least $\Omega(\lg n!)$ levels
- $\Omega(\lg n!) = \Omega(n \lg n)$...