Introduction and Motivation

Design and Analysis of Algorithms

- Design means designing an algorithm to solve a problem
- Analysis means measuring efficieny of the algorithm in terms of the input size
- We measure in two ways:
  - Time – Relationship between time taken to run the algorithm and input size
  - Space – Relationship between space consumed and input size

Measuring Running Time

- As space is easily available
- We shall concentrate of the time taken by the algorithms
- Analysis is independent of underlying hardware
    - We don't use actual time
    - Measure in terms of "basic operations"

Measuring Running Time

- Typical basic operations
  - Compare two values
  - assign a value to a variable
  - Swap two numbers
    - tmp <– x
    - x<–y
    - y<–tmp
  - Three assignments constants can be ignored

Measuring Running Time

- Running time depends on input size
- Longer array will take more to sort than shorter arrays
- But input of same size may take different amount of time

Measuring Running Time

Linear Search

Search 'K' in an unsorted array A

```
i <– 0
while i<n and A[i]=k do
        i <– i+1
if i<n return i
else return –1
```

Measuring Running Time

- Complexity of Linear Search
- Input 2 – 34, 12, 36, 78, 52 search for 34 – Best
- Input 3 – 34, 12, 36, 78, 52 search for 70 – Worst
- Average case may be computed for better idea
- But not possible to calculate for all cases
- Need probability distribution over inputs
- Good upper bound is got from worst case

How Important is Complexity?

Sorting Algorithms

Naive – $O(n^2)$

Best – $O(n\log n)$

Typical CPU process upto 10^8 operations per second

Moore's law – speed is saturated

Useful for approximate calculation

Computing Time for Sorting Mobile Users

- Sort a list of mobile numbers in India
- Roughly around 10^9 numbers
- Naive algorithm (n^2) need 10^{18} operations
- 10^8 operations per second so it needs 10^{10} seconds
- 2778000 hours
- 115700 days
- 300 years !

Computing Time for Sorting Mobile Users

- Smart algorithm (n log n)
- need 3 X 10^{10} operations
- about 300 seconds
- 5 minutes

Computing Time for a Video Game

- Several objects in the screen
- Basic step – find closest pair of objects
- Given 'n' objects, naive algorithm takes n^2 time
  - For each pair of objects, compute their distance
  - Report minimum distance over all such pairs
- There is a clever algorithm that takes n log n time

Computing Time for a Video Game

- High resolution monitor has 2500 X 1500 pixels
  - 3.75 million points
- suppose we have 500,000 = 5 X 10^5 objects
- Naive algorithm will take 25 X 10^{10} steps = 2500 secs
- 42 minutes to respond
- Smart n log n algorithm takes a fraction of a second

Typical Functions T(n)

Input	log n	n	n log n	n^2	n^3	2^n	n!
10	3.3	10	33	100	1000	1000	10^6
100	6.6	100	66	10^4	10^6	10^{30}	10^{157}
1000	10	1000	10^4	10^6	10^9		
10^4	13	10^4	10^5	10^8	10^{12}		
10^5	17	10^5	10^6	10^{10}			
10^6	20	10^6	10^7				
10^7	23	10^7	10^8				
10^8	27	10^8	10^9				
10^9	30	10^9	10^{10}				
10^{10}	33	10^{10}					

10^9 operations is considered to be fine

– 10 secs

10^{10} is too long

Quantifying Efficiency

- Big O – Upper bound
- Omega – Lower bound
- Theta – Tight bound

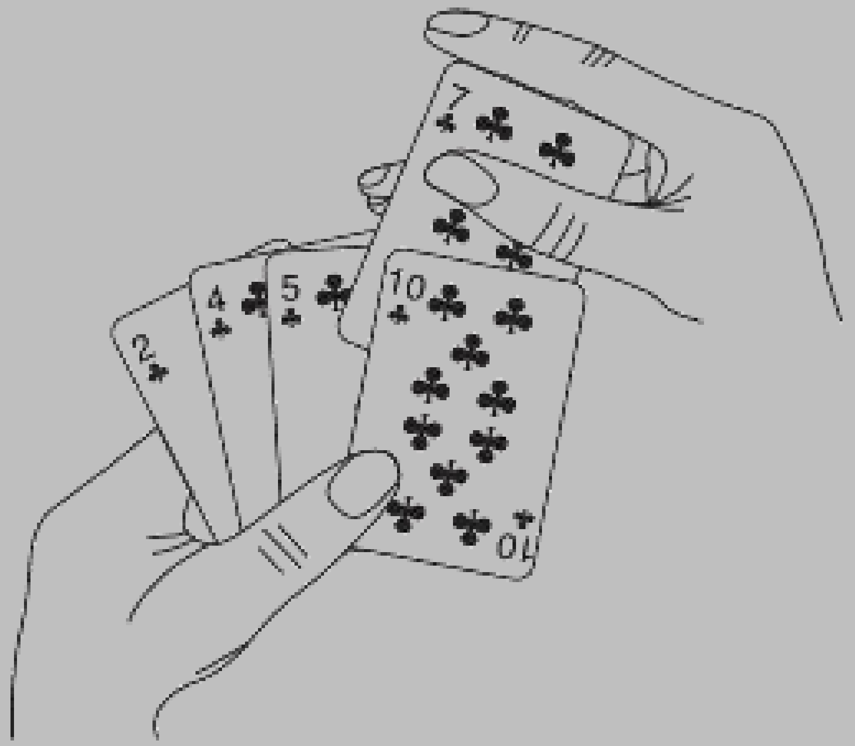
Insertion sort

- **Input:** A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.
- **Output:** A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leqslant a'_2 \leqslant \ldots \leqslant a'_n$
- Numbers that we wish to sort are also known as the keys
- Although conceptually we are sorting a sequence, the input comes to us in the form of an array with n elements.

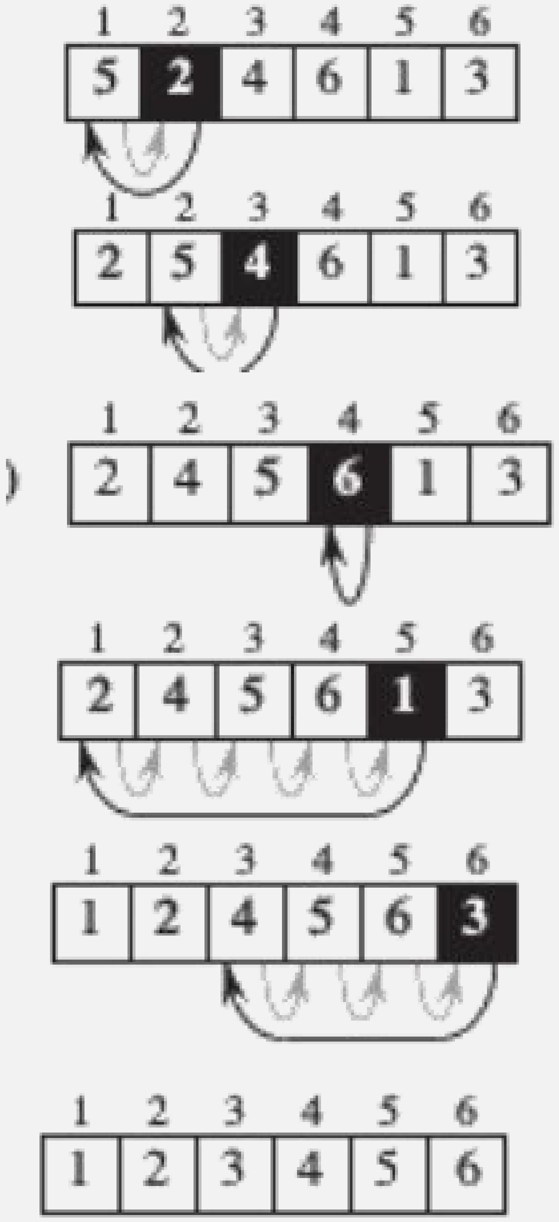
Insertion sort

- Works the way many people sort a hand of playing cards
- Start with an empty left hand and the cards face down on the table
- then remove one card at a time from the table and insert it into the correct position in the left hand
- To find correct position for a card, we compare it with each of the cards already in the hand, from right to left

Insertion sort



Insertion sort



Insertion sort

```
INSERTION-SORT(A)
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1 .. j − 1].
    i = j − 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```

- j indicates “current element” being inserted into sorted list
- At the beginning of each iteration , subarray consisting of elements A[1.. j–1] constitutes sorted elements, and remaining subarray A[j+1.. n] corresponds to elements yet to be sorted

Correctness of Insertion sort

- Identifying Loop Invariant
- Showing that the loop invariant is true in the Initialization, Maintenance and Termination

Correctness of Insertion sort

- Identifying Loop Invariant
- Fact: A[1, ..., j–1] are the elements originally in positions 1 through j–1, but now in sorted order.
- At the start of each iteration of the for loop of lines 1 – 8, the subarray A[1, ..., j–1] consists of the elements originally in A[1, ..., j–1] , but in sorted order.

Correctness of Insertion sort

- Initialization: It is true prior to the first iteration of the loop
- Maintenance: If it is true before an iteration of the loop, it remains true before the next iteration
- Termination: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct

Correctness of Insertion sort

- We use first two properties similar to mathematical induction
- First condition is similar to base case and the second condition is something similar to inductive step
- We use loop invariant along with the condition that caused the loop to terminate.
- Induction goes infinitely

Correctness of Insertion sort – Initialization

- When j =2, we have a subarray of size 1, an array of size is sorted by itself

Correctness of Insertion sort – Maintenance

- Body of for loop works by moving A[j–1], A[j–2], A[j–3] , and so on by one position to right until it finds proper position for A[j] (lines 4 – 7), at which point it inserts value of A[j] (line 8)
- Subarray A[1 .. j] then consists of elements originally in A[1 .. j] , but in sorted order
- Incrementing j for next iteration of for loop then preserves the loop invariant.

Correctness of Insertion sort – Termination

- What happens when the loop terminates
- Condition causing for loop to terminate is that $j >$ A.length
- Because each loop iteration increases j by 1, we must have $j = n+1$ at that time
- Substituting $n+1$ for j in the wording of loop invariant, we have that the subarray A[1..n] consists of the elements originally in A[1..n], but in sorted order
- Hence the entire array is sorted.

Frequency Count Method

- We count number of times one instruction is executing

```
Algorithm Sum(A,n)
{
    s = 0;
    for(i=0;i<n;i++)
        s = s+A[i];
    return s;
}
```

1

1 + (n+1) + n

n

f(n) = 2 + (n+1) + 2*n

f(n) = 3+3*n

f(n) = O(g(n)) => f(n) **≤ c*n for some c and n0**

f(n) = O(n)

Frequency Count Method

```
Algorithm Sum1(A,n)
{
    s = 0;
    for(i=1;i<n;i=i+2)
        s = s+A[i];
    return s;
}
```

1

1 + (n+1)/2 + n/2

n/2

f(n) = 1.5 + n

f(n) = O(g(n)) => f(n) ≤ **c*n for some c and n0**

f(n) = O(n)

Frequency Count Method

```
Algorithm nested(A,n)
{
    s = 0;
    for(i=0;i<n;i++)
    for(j=0;j<n;j++)
        s = s+A[i]+A[j];
    return s;
}
```

1

n+1

n*(n+1)

n*n

$f(n) = 2*n^2 + n + 1$

$f(n) = O(g(n)) \Rightarrow f(n) \leq c*n^2$ **for some c and n0**

$f(n) = O(n^2)$

Frequency Count Method

```
Algorithm nested(A,n)
{
    s = 0;
    for(i=0;i<n;i++)
    for(j=0;j<i;j++)
        s = s+A[i]+A[j];
    return s;
}
```

1

n+1

1+2+3+4+....

n*(n+1)/2

$f(n) = n^2/2 + n / 2$

$f(n) = O(g(n)) \Rightarrow f(n) \leq c*n^2$ **for some c and n0**

$f(n) = O(n^2)$

Insertion sort

```
INSERTION-SORT(A)
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1 .. j − 1].
    i = j − 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```