Algorithm Analysis

What is an algorithm ?

- **A clearly specifiable set of instructions**
  - **to solve a problem**
- **Given a problem**
  - **decide that the algorithm is correct**
- **Determine how much resource the algorithm will require**
  - **Time**
  - **Space**

Analysis of Algorithms

- **How much resource is required ?**
- **Measures for efficiency**
  - **Execution time → time complexity**
  - **Memory space → space complexity**
- **Observation :**
  - **The larger amount of input data an algorithm has, the larger amount of resource it requires.**
    - **Complexities are functions of the amount of input data (input size).**

What do we use for a yardstick?

- **The same algorithm will run at different speeds and will require different amounts of space when run on different computers, different programming languages, different compilers.**
- **But algorithms usually consume resources in some fashion that depends on the size of the problem they solve : n.**

Sorting integers

```
void sort (int A[], int N)
{
    int i, j, x;
    for (i=1; i<N; i++)
    {
        x = A[i];
        for (j=i; j>0 && x<A[j-1]; j- -)
            A[j] = A[j-1];
        A[j] = x;
    }
}
```

- **We run this sorting algorithm on two different computers, and note the time (in ms) for different sizes of input.**

Array Size n	Home Computer	Desktop Computer
125	12.5	2.8
250	49.3	11.0
500	195.8	43.4
1000	780.3	72.9
2000	3114.9	690.5

Contd.

- **Home Computer :**

  $f_1(n) = 0.0007772\, n^2 + 0.00305\, n + 0.001$

- **Desktop Computer :**

  $f_2(n) = 0.0001724\, n^2 + 0.00040\, n + 0.100$

  - **Both are quadratic function of n.**
  - **The shape of the curve that expresses the running time as a function of the problem size stays the same.**

Complexity classes

- **The running time for different algorithms fall into different *complexity classes*.**
  - **Each complexity class is characterized by a different family of curves.**
  - **All curves in a given complexity class share the same basic shape.**
- **The *O-notation* is used for talking about the complexity classes of algorithms.**

Introducing the language of O-notation

- **For the quadratic function**

  $$f(n) = an^2 + bn + c$$

  we will say that $f(n)$ is $O(n^2)$.

  - **We focus on the dominant term, and ignore the lesser terms; then throw away the coefficient.**

Mathematical background

- **$T(N) = O(f(N))$ if there are positive constants c and n_0 such that $T(N) \leq c\, f(N)$ when $N \geq n_0$.**

  Meaning : As N increases, T(N) grows no faster than f(N).

  The function T is eventually bounded by some multiple of f(N). f(N) gives an upper bound in the behavior of T(N).

- **$T(N) = \Omega(g(N))$ if there are positive constants c and n_0 such that $T(N) \geq c\, g(N)$ when $N \geq n_0$.**

  Meaning : As N increases, T(N) grows no slower than g(N) ; T(N) grows at least as fast as g(N).

T(N) belongs to a family of function.

Contd.

- **$T(N) = \theta(h(N))$ if and only if $T(N) = O(h(N))$ and $T(N) = \Omega(h(N))$**

Meaning : As N increases, T(N) grows as fast as h(N).

- **$T(N) = o(p(N))$ if $T(N) = O(p(N))$ and**

$$T(N) \neq \theta(p(N))$$

Meaning : As N increases, T(N) grows slower than p(N). $\lim_{n\to\infty} T(N)/p(N) = 0$.

Examples

- $\log_e n = O(n)$
- $n^{10} = o(2^n)$
- $3\,n^2 + 5n + 1 = \theta(n^2)$

Concepts in Analysis

1. Worst Case
2. Average case (expected value)
3. Operator count

Why is the analysis of algorithms important ?

Can advance on hardware overcome inefficiency of your algorithm ?

→ **NO !**

Model of computation

- **A normal computer, instructions executed sequentially.**
  - **addition, multiplication, comparison, assignment, etc.**
  - **all are assumed to take a single time unit.**

Running time of algorithms

Assume speed S is 10^7 instructions per second.

size n	10	20	30	50	100	1000	10000
n	.001 ms	.002 ms	.003 ms	.005 ms	.01 ms	.1 ms	1 ms
nlogn	.003 ms	.008 ms	.015 ms	.03 ms	.07 ms	1 ms	13 ms
n^2	.01 ms	.04 ms	.09 ms	.25 ms	1 ms	100 ms	10 s
n^3	.1 ms	.8 ms	2.7 ms	12.5 ms	100 ms	100 s	28 h
2^n	.1 ms	.1 s	100 s	3 y	3x 10^{13}c	inf	inf

Observations

- **There is a big difference between polynomial time complexity and exponential time complexity**
- **Hardware advances affect only efficient algorithms and do not help inefficient algorithms.**

Maximum subsequence sum problem

- **Given (possibly negative) integers $\langle A_1\ A_2 \ldots A_N \rangle$ find the maximum value of $\sum_{k=i}^{j} A_k$.**
  - **For convenience, the maximum subsequence sum is considered to be 0 if all the integers are negative.**
- **Example :**
  - **For input <-2,11,-4,13,-5,2> the answer is 20 (A_2 to A_4)**

Algorithm 1

```
int MaxSubSum (int A[], int N)  {
    int thissum, maxsum, i,j,k;
    maxsum = 0;
    for (i=0; i<N; i++)
        for (j=i; j<N; j++)     {
            thissum = 0;
            for (k=i; k <= j; k++)
                thissum += A[k];
            if (thissum > maxsum)
                maxsum = thissum;
        }
    return maxsum;
}
```

- **The loop at line 2 is of size N.**
- **The second loop has size N-i.**
- **The third loop has size j-i+1**
- **Total : about N^3 steps**
- $\sum_{k=i}^{j} 1 = j-i+1$
- $\sum_{k=i}^{j} (j-i+1) = (N-i+1)(N-i)/2$
- $\sum_{i=0}^{N-1} (N-i+1)(N-i)/2 = (N^3 + 3N^2 + 2N)/6$

Improve the running time

- **Remove the second for loop**
- **Observe :**
  - $\sum^{j}_{k=i} A_k = A_j + \sum^{j-1}_{k=i} A_k$

Algorithm 2

```
int MaxSubSum2 (int A[], int N)
{
      int thissum, maxsum, i, j;
1. maxsum = 0;
2.    for (i=0; i<N; i++)
3.    {
3.          thissum = 0;
4.          for (j=i; j < N; j++)
5.          {
5.                thissum += A[j];
6.                if (thissum > maxsum)
7.                      maxsum = thissum;
            }
      }
8.    return maxsum;
}
```

Complexity : $O(N^2)$

Search in a sorted array

- Given an integer X, and integers $<A_0\ A_1 . . . A_{N-1}>$ which are presorted and already in memory, find i such that $A_i = X$, or return i = -1 if X is not in the input.

Linear Search

```
int search (int A[], int X, int N)
{
        int i;
        for (i=0; i<N; i++)
                if (A[i] == X)
                        return i;
        return -1;
}
```

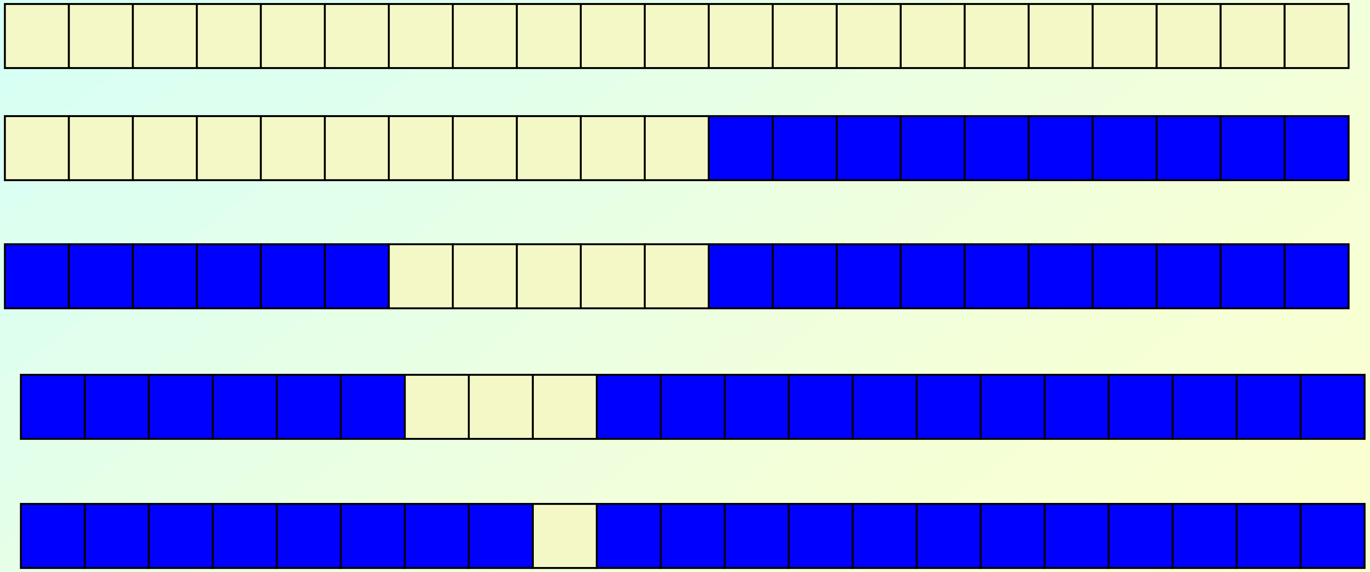
Complexity : $\theta(N)$

Binary Search

```
int BinarySearch (int A[], int X, int N)      {
    int low, mid, high;
    while (low <= high)         {
        mid = (low+high)/2;
        if (A[mid] < X)  low = mid+1;
        else if (A[mid] > X) high = mid-1;
        else return mid;
    }
    return -1;
}
```

Binary Search Illustrated

- possible positions for what we are looking for
- ruled out as a possible position for what we are looking for



Analysis of binary search

- **All the work done inside the loop takes O(1) time per iteration.**
- **Number of times the loop is executed :**
  - **The loop starts with high -low = N-1**
  - **Finishes with high -low ≥1**
  - **Every time through the loop the value of high -low is at least halved from its previous value.**

is at most $\lceil \log_2(N-1) \rceil + 2 = O(\log N)$.

Sorting integers

```
void sort (int A[], int N)      {
    int i, j, x;
    for (i=1; i<N; i++) {
        x = A[i];
        for (j=i; j>0 && x<A[j-1]; j--)
            A[j] = A[j-1];
        A[j] = x;
    }
}
```

$T(N) =$
$1+2+ \ldots + N-1$
$= N(N-1)/2$
$\in \theta(N^2)$

Worst Case Analysis

- Suppose that all the cases fall in one of n cases: x_1, x_2, ... , x_n
  c_i denotes the cost for case x_i.
- Worst case complexity = $\max\{c_i | 1 <= i <= n\}$
- Example : Sequential search on a table.
- There are n+1 cases
- Worst case time complexity = n

Average Case Analysis

- Suppose that all the cases fall in one of n cases: $x_1, x_2, \ldots, x_n$
  c_i denotes the cost for case x_i.
  p_i denotes the probability of x_i.
- Average case complexity = $\Sigma^n_{i=1} p_i c_i$
- Example : Sequential search on a table (the key is in the table and every key is equally likely)
- There are n cases, each w.p. 1/n.
- Average case time complexity = $\Sigma^n_{i=1} i / n$
  $= (n+1)/2$