Lecture 2.2

- Dr. Anurag Sharma

Complexity Analysis of Algorithms

Analysis of Algorithms

- How can we say that an algorithm performs better than another?
- The critical resource for a program is most often its running time and space required to run the program.
  - Time efficiency
  - Space efficiency
- Time is not merely CPU cycles - we want to study algorithms *independent* of *implementations*, *platforms* and *hardware*

Cont.

- When analyzing the efficiency of an algorithm in terms of time, we do not determine the actual number of CPU cycles because this depends on the computer on which the algorithm is run.
- We don't count every instruction executed because the number of instructions depends on the programming language used. We want a measure that is independent of the computer, the programming language, the programmer, and all the complex details of the algorithm.

Cont.

- In general, the running time of the algorithm increases with the size of the input. The total running time is proportional to how many times some basic operation is done. Therefore, we analyze an algorithm's efficiency be determining the number of times some basic operation is done as a function of the size of the input. The size of the input is called the **input size**.
- In general, a time complexity analysis of an algorithm is the determination of how many times the basic operation is done for each value of the input size.

Time Complexity

- A **time complexity analysis** of an algorithm is the determination of how many times the basic operation is done for each value of the input size.

Every-Case Time Complexity

- **every-case time complexity** T(*n*) is defined as the number of times the algorithm does the basic operation for an instance of size *n*
- **Example 1: Adding Array Members**

```
int sum(int n, int array[])
{
   int result = 0;
   for (int i = 0; i < n; i ++)
        result = result + array[i]; // basic operation
    return result;
}
```

  T(*n*) = *n*

Example - 2

Problem: Sort n keys in non-decreasing order

Inputs: Positive integer n, array of keys S indexed from 1 to n
Outputs: Sorted array S

```
void exchangesort (int n, keytype S[])
{
index i, j;
for (i=0; i<n; i++)
  for (j=i+1; j<n; j++)
    if (S[j] < S[i])
      exchange S[i] and S[j];
}
```


T(*n*) for exchange sort

- $T(n) = (n-1) + (n-2) + (n-3) + \ldots + 1$
- $= (n-1)n/2$

Example - 3

- **Problem**: Determine the product of two *n x n* matrices
- **Inputs**: a positive integer n, 2-d arrays of numbers *A* and *B*,each of which has both its rows and columns indexed from *1* to *n*.
- **Outputs**: a 2-d array of numbers *C*, which has both its rows and columns indexed from *1* to *n*, containing the product of *A* and B.
- For c++ and java index starts from 0.

```
void matrixmult (int n, const number A[][],
                 const number B[][], number C[][])
{
   index i, j, k;
   for (i=1; i<=n; i++)
      for (j=1; j<=n; j++) {
         C[i][j] = 0;
         for (k=1; k<=n; k++)
            C[i][j] = C[i][j] + A[i][k] * B[k][j];
      }
}
```


T(n) for matrix multiplication

- $T(n) = n * n * n$ //n operations for each loop
- $= n^3$

Worst-Case Time Complexity

- Worst-Case Time Complexity $W(n)$ is defined as the maximum number of times the algorithm will ever do its basic operation for an input size of n. The determination of $W(n)$ is called a worst-case time complexity analysis.
- If T(n) exists, then $W(n) = T(n)$
- Otherwise just take the worst case where maximum computation is required.

Example

- Problem: sequential search

```
static int sequentialSearch(final Comparable key,
final ArrayList<? extends Comparable> in){
    int loc = 0;
    while(loc<in.size() && in.get(loc).compareTo(key) != 0){
        loc++;
    }
    if(loc>=in.size())
        loc = -1;
    return loc;
}
```

W(n) for sequential search

- Worst case?
- When the key to be searched is the last element of the array or
- The key does not exists
- $W(n) = n$

Best-Case Time Complexity

- Best time complexity $\boldsymbol{B(n)}$ is defined as the minimum number of times the algorithm will ever do its basic operation for an input size of n. The determination of $B(n)$ is called a best-case time complexity analysis.
- If T(n) (algorithm does not finish until all n element are considered) exists, then $B(n) = T(n)$

B(n) for sequential search

- Best case?
- When the key to be searched is the first element of the array, is the best case.
- $W(n) = 1$

Average-Case Time Complexity

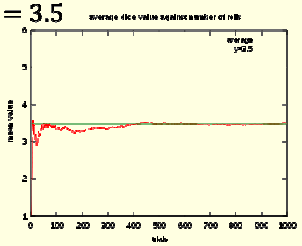
- $\boldsymbol{A(n)}$ is called the average-case time complexity of the algorithm, and the determination of $A(n)$ is called an average-case time complexity analysis.
- If T(n) exists, then $A(n) = T(n)$
- To compute $A(n)$ assign probability to all the input elements of size n.

How to compute A(n)?

- The **expected value** of a discrete random variable X, symbolized as $E(X)$, is often referred to as the *long-term average or* ***mean***. This means that over the long term of doing an experiment over and over, you would expect this average.

$$E(X) = \sum_{i=1}^{n} X_i . P(X_i)$$

- For example, the expected value in rolling a six-sided die is 3.5.
- E(rolling a die) = $1.\frac{1}{6} + 2.\frac{1}{6} + \cdots + 6.\frac{1}{6} = 3.5$
- If n grows, the average will almost surely converge to the expected value, a fact known as the strong law of large numbers.



A(n) for sequential search

- $E(X) = \sum_{i=1}^{N} i \times \frac{1}{N} = \frac{1}{N}\frac{N(N+1)}{2} = \frac{N+1}{2}$

Here we have assumed that an element is in the array. What if in some cases element is not in the list?

Overall A(n) for sequential search

- Element k is either in slot 1 or in slot 2 or in slot i or nowhere in the array.

Prob (slot 1) = 1/n
P(slot 2) = 1/n
P(slot i) = 1/n
P(exists) = p
P(does not exist) = 1-p
E(n) = 1*p/n + 2*p/n + … n*p/n + n*(1-p)
= n(1-p/2)+p/2