Algorithm Analysis

- **Example: Sequential Search.**
  - Given an array of n elements, determine if a given number val is in the array.
  - If so, set loc to be the index of the first occurrence of val, and return *true*.
  - Otherwise, return false.
- **The Algorithm:**

```
bool SeqSearch(int A[],int n,int val,int &loc) {
   loc = 0;
   while( loc < n && A[loc] != val )
      ++ loc;
   return loc < n;
}
```

- How much time does it take to execute this program?
- Is this the best question to ask? What are some of the important considerations?

Factors Affecting Run Time

- Characteristics of the computer system (e.g. processor speed, amount of memory, file-system type, etc.)
- The way the algorithm is implemented
- The particular instance of data the algorithm is operating on (e.g., amount of data, type of data).

Conclusion?

Given these facts

- What should we use as a measure of how “good” an algorithm is?
- By what should we compare two algorithms with each other?

This is what algorithm analysis is all about.

Good Measures, Bad Measures

- The obvious choices for characterizing algorithms is
  - The amount of time required (*time complexity*)
  - The amount of space required (*space complexity*)
- We are usually more interested in the time complexity.
- What is *time*? It can be any of the following:
  - Wall-clock or real time
  - CPU-time
  - Number of instructions executed

Which Measure?

- We usually use the the *number of instructions executed* as our measure of time. Why?
- We are not that interested in determining *exactly* how much time a given algorithm will take. Why?
- Instead, we will try to determine the *rate of growth* of the running time.
- Using the rate of growth, we can compare algorithms, independent of the implementation details.

Simplifying Assumptions

- As stated several times before, the characteristics of the particular computer system that the algorithm will execute on are considered irrelevant.
- Often the implementation of the algorithm is also ignored, although later we'll see an example where it matters.
- Each instruction, no matter how simple or complex, is considered to take one "unit" of time. Why?
- Some simple measure of the "size" of the data that the algorithm is operating on is made, e.g., the size of an array, the number of nodes in a graph, the dimensions of a matrix.

Example Revisited

Sequential Search

```
SeqSearch(int A[],int n,int val,int &loc)
{
   loc := 0;
    while (loc < n && A[loc] != val)
        ++ loc;
    return loc < n;
}
```

- What is the size of the input?
- How many operations, as a function of the array size n, are required by SeqSearch?

Analyzing the Running Time

- Identify typical input data
- Identify abstract operations
- Derive a mathematical analysis
- Associate the algorithm to a *complexity class* (This is the topic of the next section)

Typical Input Data

- We first need to determine what the input is, and *how much data* is being input.
- We need to determine which of the data affects the running time.
- We usually use n to denote the number of data items to be processed.
- This could be
  - size of a file
  - size of an array or matrix
  - number of nodes in a tree or graph
  - degree of a polynomial

Abstract Operations

- We talk about abstract operations when we consider operations in a hardware independent fashion.
- Recall that we are interested in rate of growth, not the exact running time. Thus, we can pick operations that will run most often in the code.
- Determine the number of times these operations will be executed as a function of the size of the input data.
- It is crucial that we pick the operations that are executed most often, and that we recognize when an operation can or cannot be performed in a constant amount of time.

Example: factorial

```
factorial(n) {
   if(n==1)
       return 1
   else
      return n * factorial(n-1)
}
```

- We focus on the comparison (==) (this is the abstract operation) and ignore the other instructions.
- For example, if we calculate the number of operations in this function based on the comparison operator, we have:
  - for factorial(1), 1 operation
  - for factorial(2), 2 operations

    $\vdots$
  - for factorial(n), n operations

Example: Search for Maximum

```
int max(int a[],int n) {
    int max = int.MIN_VAL;
    for (int i=0; i<n; i++)
        max = MAXIMUM(max, a[i]);
    return max;
}
```

- We focus on the assignment (=) inside the loop and ignore the other instructions.
- for an array of length 1, 1 comparison
- for an array of length 2, 2 comparisons

$$\vdots$$

- for an array of length n, n comparisons

Mathematical Analysis

There are three types of analysis that can performed on an algorithm.

- **Best-case analysis**
  Analysis of the performance of the algorithm assuming the “easiest” instance of data input.

  –This is the most useless one. Why?

- **Average-case analysis**
  Analysis of the performance of the algorithm assuming an “average” instance of data input.

  –This may be difficult. Why?

- **Worst-case analysis**
  Analysis of the performance of the algorithm assuming the “worst” instance of data input.

  –This is the most practical. Why?

Analysis Example: Insertion Sort

```
void insertion(itemType A[], int n) {
     int i, j; itemType  v;
     for (i=1;i<=n;i++) {
            v=A[i];  j=i-1;
            while (j > 0 && A[j] > v)
                  { A[j+1] = A[j];  j--; }
            A[j+1]=v;}
}
```

- Assume we use the comparisons in the "while" loop as our abstract operation.
  (Is this a good choice?)
- **Worst-case:** When the array A is sorted in descending order, $A[j] > v$ for 1 to $i-1$ for every iteration of the "for" loop. The total number of comparisons is $\sum_{i=2}^{n}(i-1) = n(n-1)/2 \approx n^2/2$.
- **Best-case:** When the array A is already sorted in ascending order, the algorithm only executes n comparisons!

Analysis Example: SumOfProducts

```
double SumOfProducts(double A[], int size) {
     double V;
     for (int i=1; i<=size; i++) {
         for (int j=1; j<=size; j++) {
             V=A[i]*A[j];
         }
     }
}
```

- We will use the assign (`V=A[i]*A[j]`) as our abstract operation. (Is this a good choice?)
- Since there are no conditionals (`if`, `while`) the worst, average, and best case will be the same.
- Notice that j ranges from 1 to $size$.
- Thus, each time the inner loop executes, it uses $size$ operations.
- The outer loop also executes $size$ times, each time executing the inner loop.
- Thus, the number of operations is $size * size = size^2$.