CPSC 331
Tutorial4

Department of Computer Science
University of Calgary

Running Time

- Number of primitive operations or "steps" (programming language statements) used
  - Worst Case Analysis
  - Average Case Analysis
  - Best Case Analysis

```
public static boolean distinctEntries ( int[] A ) {

  for (int i=1; i <= A.length; i++ ) {
  /*
   * Loop Invariant:
   *  a) i is an integer such that 1 <= i < A.length
   *  b) A[r] is not equal to A[s] for all integers r
   *     and s such that 0 <= r < s < i
   * Loop Variant: A.length - i
   */
      for (int j=1; j <= i; j++ ) {
      /*
      * Loop Invariant:
      *  a) i and j are integers such that
      *     0 <= j < i < A.length
      *  b) A[r] is not equal to A[s] for all
      *     integers r and s such that
      *     0 <= r < s < i
      *  c) A[r] is not equal to A[i] for every
      *     integer r such that
      *     0 <= r < j
      * Loop Variant: i-j
      */
         if (A[j] = A[i]) {
            return false;
         };
      };
   };
   return true;

}
```

Proof of correctness for the outer loop

```
* Loop Invariant:
*  a) i is an integer such that 1 <= i < A.length
*  b) A[r] is not equal to A[s] for all integers r
*     and s such that 0 <= r < s < i
```

- First step
- Base Case i=1
  - A) 1<=i< A.length
  - B) there is no r which is 0<=r<s<1
- At the end of i-th execution
  - 1<=i<=A.length
  - A[r] ≠ A[S] 0<= r<s<=i
- At the beginning of the i= i+1 execution, we knoe that the loop test must pass after the end of the ith execution (i<a.length)
  - 1<=i<A.length
  - A[r] ≠ A[S] 0<= r<s<I

- **Next step:**
- Apply the loop invariant to prove that the algorithm's post conditions hold (prove these yourself similar to the correctness example in tutorial #2)

- Now, suppose you try to simplify the loop invariant for the inner loop by leaving out conditions (a) and (b). What goes wrong?

```
/*
* Loop Invariant:
*  a) i and j are integers such that
*     0 <= j < i < A.length
*  b) A[r] is not equal to A[s] for all
*     integers r and s such that
*     0 <= r < s < i
*  c) A[r] is not equal to A[i] for every
*     integer r such that
*     0 <= r < j
* Loop Variant: i-j
*/
```

As part of the previous tutorial exercise, you tested and debugged a program that would be used to determine whether the entries of a given array are distinct. A corrected version of this program is given.

```
public static boolean distinctEntries ( int[] A ) {
    for (int i=1; i <= A.length; i++ ) {
        /*
        * Loop Invariant:
        *  a) i is an integer such that 1 <= i < A.length
        *  b) A[r] is not equal to A[s] for all integers r
        *     and s such that 0 <= r < s < i
        * Loop Variant: A.length - i
        */
        for (int j=1; j <= i; j++ ) {
            /*
            * Loop Invariant:
            *  a) i and j are integers such that
            *     0 <= j < i < A.length
            *  b) A[r] is not equal to A[s] for all
            *     integers r and s such that
            *     0 <= r < s < i
            *  c) A[r] is not equal to A[i] for every
            *     integer r such that
            *     0 <= r < j
            * Loop Variant: i-j
            */
            if (A[j] = A[i]) {
                return false;
            };
        };
    };
    return true;

}
```

???

For (int i=1;i<A.lenght;i++)

Loop Invariant

Loop variant

???

For (int j=0; j<I;j++)

Loop Invariant

Loop variant

A[j] == A[i]

```
public static boolean distinctEntries ( int[] A ) {

  for (int i=1; i < A.length; i++ ) {
  /*
   * Loop Invariant:
   *  a) i is an integer such that 1 <= i < A.length
   *  b) A[r] is not equal to A[s] for all integers r
   *     and s such that 0 <= r < s < i
   * Loop Variant: A.length - i
   */
      for (int j=0; j < i; j++ ) {
      /*
      * Loop Invariant:
      *  a) i and j are integers such that
      *     0 <= j < i < A.length
      *  b) A[r] is not equal to A[s] for all
      *     integers r and s such that
      *     0 <= r < s < i
      *  c) A[r] is not equal to A[i] for every
      *     integer r such that
      *     0 <= r < j
      * Loop Variant: i-j
      */
         if (A[j] == A[i]) {
            return false;
         };
      };
   };
   return true;
}
```

Analyzing running time

- Why running time?
  - Why need algorithm analysis ?
    - writing a working program is not good enough
    - The program may be inefficient!
    - If the program is run on a large data set, then the running time becomes an issue

Public static Boolean distinctEntires (int[] A){	**cost**	**time**
for (int i=1;i<A.lenght;i++) {	c1	n
for (int j = 0; j<i; j++){	c2	~~(n-1)(n)~~ 2+3+…+n
if (A[j] == A[i]) {	c3	~~(n-1)(n-1)~~ 1+2+3+… +(n-1)
return false;	c4	1
};		
};		
};		
return true;	c4	1
}		

- $\sum_{j=0}^{i} t_j$
- At the worse case i=n-1
- $t_j = j$, j =0,1,2,

Questions

a)

Analyze the *worst case running time* of this program.

Answer:

O(n^2)

```
public static boolean distinctEntries ( int[] A ) {

  for (int i=1; i < A.length; i++ ) {
  /*
   * Loop Invariant:
   *  a) i is an integer such that 1 <= i < A.length
   *  b) A[r] is not equal to A[s] for all integers r
   *     and s such that 0 <= r < s < i
   * Loop Variant: A.length - i
   */
      for (int j=0; j < i; j++ ) {
      /*
      * Loop Invariant:
      *  a) i and j are integers such that
      *     0 <= j < i < A.length
      *  b) A[r] is not equal to A[s] for all
      *     integers r and s such that
      *     0 <= r < s < i
      *  c) A[r] is not equal to A[i] for every
      *     integer r such that
      *     0 <= r < j
      * Loop Variant: i-j
      */
         if (A[j] == A[i]) {
            return false;
         };
      };
   };
   return true;
}
```

Questions

b) Try to analyze the *best case running time* of this program.

Answer:

O(1)

```
public static boolean distinctEntries ( int[] A ) {

  for (int i=1; i < A.length; i++ ) {
  /*
   * Loop Invariant:
   *  a) i is an integer such that 1 <= i < A.length
   *  b) A[r] is not equal to A[s] for all integers r
   *     and s such that 0 <= r < s < i
   * Loop Variant: A.length - i
   */
      for (int j=0; j < i; j++ ) {
      /*
      * Loop Invariant:
      *  a) i and j are integers such that
      *     0 <= j < i < A.length
      *  b) A[r] is not equal to A[s] for all
      *     integers r and s such that
      *     0 <= r < s < i
      *  c) A[r] is not equal to A[i] for every
      *     integer r such that
      *     0 <= r < j
      * Loop Variant: i-j
      */
         if (A[j] == A[i]) {
            return false;
         };
      };
   };
   return true;
}
```

Average time

- Average-case running time
  - May be difficult to define what "average" means
- Consider the average (or expected) amount of resources (such as average running time) used by the algorithm, for an input of a given size
- Advantage of This Type of Analysis:
  - captures resource consumption for typical inputs
- Disadvantages of This Type of Analysis:
  - executions on some inputs of the given size can take much longer than the average case
  - may be difficult to determine what the average case actually is
  - some assumption about the distribution of the inputs is always needed

Questions

- c) What, if anything, can be said about the *average* (or "*expected*") running time of this program, based on what you have discovered?

Answer:

n/2 * n/2 => O(n^2)