Insertion Sort

Code

```
function insertionSort(A: number[]) {
    for(let j = 1; j < A.length; j++) {
        const key = A[j];
        // insert A[j] into the sorted sequence A.slice(0, j - 1)
        let i = j - 1;
        while (i >= 0 && A[i] > key) {
            A[i + 1] = A[i];
            i--;
        }
        A[i + 1] = key;
    }
    return A;
}
```

Design

- uses an **incremental** approach
  - having sorted the subarray `A.slice(0, j - 1)`, we insert the single element `A[j]` into its proper place

Correctness

- refer to the code
- notice that `j` holds the "current card" being sorted into the "hand"
- we state properties of subarray `A.slice(0, j - 1)` as a **loop invariant**
  - the subarray `A.slice(0, j - 1)` are elements *originally* in positions 0 through $j - 1$ but in sorted order
- to understand why an algorithm is correct, we must show 3 things about a loop invariant
  1. **initialization**: it is true prior to first iteration of the loop
  2. **maintenance**: if it is true before an iteration of the loop, it remains true before the next iteration
  3. **termination**: when the loop terminates, the invariant gives us a useful property that helps show the algorithm is correct
- applying this to `insertionSort`
  1. **initialization**: `j = 1` before the first iteration such that `A.slice(0, j - 1)` consists of a single element `A[0]` such that the subarray is trivially sorted
  2. **maintenance**: (informally) the body of the for loop moves `A[j - 1]`, `A[j - 2]`, `A[j - 3]`, and so on by one position to the right until it finds the proper position for `A[j]` such that `A.slice(0, j)` will hold elements originally in positions 0 through j but in sorted order. Thus, incrementing `j` for the next iteration of the for loop preserves the loop invariant

3. **termination**: the for loop terminates when `j == A.length == n` such that `A.slice(0, n)` must be sorted at termination

Runtime Analysis

Counting Approach for Iterative Algorithms

- on approach for *iterative algorithms* is to count the number of times each statement is executed
- define constants c for the execution time of each statement
- finally, develop a function describing runtime as a function of the problem size n

line number	cost	times	comments
2	c_2	n	The for loop condition check runs n times.
3	c_3	$n-1$	Its body breaks before the last check.
5	c_5	$n-1$	
6	c_6	$\sum_{j=2}^{n} t_j$	t_j is the number of times the while loop test is executed for `j` (i.e. at most 1 the first time, 2 the second, etc...)
7	c_7	$\sum_{j=2}^{n} (t_j - 1)$	
8	c_8	$\sum_{j=2}^{n} (t_j - 1)$	
19	c_{10}	$n-1$	

Developing a Function

$$T(n) = c_2 n + c_3(n-1) + c_5(n-1) + c_6 \sum_{j=2}^{n} t_j$$

$$+c_7 \sum_{j=2}^{n} (t_j - 1) + c_8 \sum_{j=2}^{n} (t_j - 1) + c_{10}(n-1)$$

Best Case

- in the *best case*, the array sorted and $t_j = 1$ for all j such that
  - $\sum_{j=2}^{n} t_j = (n-1) \cdot 1$
  - $\sum_{j=2}^{n} (t_j - 1) = 0$

$$\begin{aligned} T(n) &= c_2 n + c_3(n-1) + c_5(n-1) + c_6(n-1) + c_{10}(n-1) \\ &= an + b \end{aligned}$$

In the best case, we have **linear growth function**.

Worst Case

- in the *worst case*, the array is in reverse order such that the $t_j = j$ for all j
- recall that the sum of an arithmetic series is $S_n = \frac{n}{2} \cdot (a_1 + a_n)$
  - where n is the number of terms in the series
  - a_1 is the first term
  - a_n is the last term
- thus
  - $\sum_{j=2}^{n} t_j = \sum j = \frac{n-1}{2}(2+n) = (n-1)(n+2)/2$
  - $\sum_{j=2}^{n} (t_j - 1) = \sum (j-1) = \frac{n-1}{2}(1+n-1) = n(n-1)/2$

$$\begin{gathered} T(n) = c_2 n + c_3(n-1) + c_5(n-1) \\ + c_6(n-1)(n+2)/2 + c_7 n(n-1)/2 + c_8 n(n-1)/2 + c_{10}(n-1) \\ = an^2 + bn + c \end{gathered}$$

In the worst case, we have **quadratic polynomial function**.

Average Case

In the average case, $t_j = j/2$ for all j and it will also yield a quadratic polynomial function.