```
INSERTION-SORT(A)
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1 .. j − 1].
    i = j − 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```

Loop invariants and correctness of insertion sort

$A[1..j]$: elements currently sorted

$A[j+1..n]$: elements "on the table" that need to be sorted

Note: Elements in $A[1..j-1]$ are the elements that were originally

in place of indices from 1 to $j-1$, but now they are sorted. This property is called a <u>loop invariant</u> and it is used to establish correctness of the algorithm.

To establish correctness, we need to verify the following three things of a loop invariant:

<u>Initialization</u>: it holds before the first iteration starts

<u>Maintenance</u>: if it holds before an iteration, it should also hold before the next iteration.

<u>Termination</u>: when algorithm halts, one should be able to evaluate its correctness.

Three conditions of loop invariant resemble the method of mathematical induction:

initialization $\leftrightarrow$ base

maitenance $\leftrightarrow$ induction step

In induction, there are infinitely many induction steps. In an algorithm, the "induction" stops when algorithm finishes its execution.

To terminate, one usually adds some stopping condition(s).

Example of the induction method proof

Show that

$$1+2+3+\ldots+(n-1)+n=\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$$

Solution

Base: Check case $n=1$

$$1 = \frac{1(1+1)}{2}$$

$$1 = 1 \cdot \frac{2}{2}$$

$$1 = 1 \quad \checkmark$$

or $n=2$

$$1+2 = \frac{2(2+1)}{2}$$

$$3 = \frac{2 \cdot 3}{2}$$

$$3 = 3 \quad \checkmark$$

Induction assumption: assume the statement is true for $n=k$, i.e.

$$1+2+3+\ldots+(k-1)+k = \sum_{j=1}^{k} j = \frac{k(k+1)}{2}$$

Induction step: show that the statement is true for $n=k+1$, i.e.

$$1+2+\ldots+k+(k+1) = \frac{(k+1)((k+1)+1)}{2} = \frac{(k+1)(k+2)}{2}$$

Indeed,

$$\underbrace{1+2+\ldots+k}_{\frac{k(k+1)}{2}\ \text{by induction assumption}}+(k+1) = \frac{k(k+1)}{2}+(k+1) =$$

$$= (k+1)\left(\frac{k}{2}+1\right) = (k+1)\,\frac{k+2}{2} =$$

$$= \frac{(k+1)(k+2)}{2} = \frac{(k+1)\big((k+1)+1\big)}{2} \quad \checkmark$$

$\blacksquare$

Back to Insertion_Sort procedure

```
INSERTION-SORT(A)
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1 .. j − 1].
    i = j − 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```

1) <u>Initialization</u>: $j=2$ $A[2]$

To the left of $A[2]$, the subarray $A[1..j-1]$ is just $A[1]$. This is, of course, sorted initially.

2) <u>Maintenance</u>: need to show that iteration preserves loop invariant.

Start at j. Subarray $A[1..j]$ has elements from originally $A[1..j]$ but in the sorted order. At most we shift elements $A[j-1]$, $A[j-2]$ etc. one position to the right until we find a proper position for $A[j]$. Then we insert the value of $A[j]$ (line 8). At the end of iteration we get elements $A[1..j]$ but in the sorted order. Hence, the loop invariant is preserved.

Termination

Stopping condition: in <u>for</u> loop

$j > A.length$ $\qquad$ $A.length = n$

When $j = n+1$, we leave the <u>for</u> loop.

For $j = n+1$, the subarray $A[1..j]$ is $A[1..n]$ but it is in the sorted order.

$A[1..n]$ is in fact the entire array that we need to sort. Hence, sorting is completed and we conclude that the algorithm is correct.

2.2 Analyzing algorithms

Analyzing algorithm means we need to predict resources algorithm requires.

Resources: memory, communication bandwidth

We will concentrate on computational time.

We need to use a simple but realistic "technology": random-access machine (RAM), one-processor, it can execute one instruction at a time, no concurrent operations.

RAM's instructions: arithmetic (+, −, *, /, remainder, floor, ceiling),

data movement (load, store, copy), and control (conditional and unconditional branch, subroutine call and return).

We assume that each instruction "costs" a finite amount of time.

Data types: integers and floating point data (real numbers).

Assume that words of data are not too long. In fact, for input of size n, the length of word is $\leq c \cdot \lg n$, $c = \text{const} \geq 1$

Analysis of insertion sort

Time of implementation depends on how many numbers are sorted: more numbers we have, more time it will

take to sort. For sequences of the same length, the time will depend how far sequences are from being sorted.

<u>Input size</u> depends on a problem

- sorting or for computing discrete Fourier transform, the input size is # of elements
- to add two integers, the input size would be the total sum of bits to represent given #s.
- algorithm uses a graph than one would need two numbers: to define vertices and edges

<u>Running time</u>: # of primitive operations or 'steps' that need to be executed.

For insertion sort, assume that every line (step) has cost (time) c_i, $i=1,..,8$. Assume comments are not executed $\Rightarrow$ time $=0$, $c_i=0$.

For each $j=2,..$ A.length, we have <u>while</u> loop. Denote by t_j the number of times we test <u>while</u> loop condition

<u>Note</u>: test is executed one time more than the body of <u>for</u> or <u>while</u> loop.

	INSERTION-SORT(A)	Cost	times
1.	for $j=2$ to A.length	c_1	n
2	key = A[j]	c_2	$n-1$
3	// Insert A[j] in corr. pos. inside A[1..j-1]	0	
4	$i=j-1$	c_4	$n-1$
5	while $i>0$ and $A[i]>key$	c_5	$\sum_{j=2}^{n} t_j$
6	A[i+1] = A[i]	c_6	$\sum_{j=2}^{n}(t_j-1)$
7	$i=i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
8	A[i+1] = key	c_8	$n-1$

Total time =

$$= C_1 \cdot n + C_2 \cdot (n-1) + C_4 \cdot (n-1) + C_5 \cdot \sum_{j=2}^{n} t_j$$

$$+ C_6 \sum_{j=2}^{n} (t_j - 1) + C_7 \sum_{j=2}^{n} (t_j - 1) + C_8 (n-1)$$

For fixed array size n, the running time will be smaller if sequence is close to being sorted. The best case occurs when array A is sorted. Then we ~~che~~ test condition in line 5 only once

$\Rightarrow t_j = 1$

Then

$$c_5(n-1) = c_5 \sum_{j=2}^{n} 1 = c_5 \sum_{j=2}^{n} \overset{=1}{t_j}$$

$$T(n) = c_1 \cdot n + c_2(n-1) + c_4(n-1) + c_5(n-1) +$$

$$+ c_8(n-1) =$$

$$= \underbrace{(c_1 + c_2 + c_4 + c_5 + c_8)}_{a} n + \underbrace{(-c_2 - c_4 - c_5 - c_8)}_{b}$$

$\Rightarrow T(n) = an + b$: <u>linear function in n</u>

where a, b are some constants that depend on costs c_i.

If an array is in reverse order, i.e. decreasing order, we have the worst case we will have to compare $A[j]$ with every element in $A[1..j-1]$ for $j=2,\dots n$

$\Rightarrow t_j = j$

$$\sum_{j=2}^{n} \underset{\substack{\| \\ j}}{t_j} = \sum_{j=2}^{n} j =$$

$$= \sum_{j=1}^{n} j - 1 = \frac{n(n+1)}{2} - 1$$

$$\sum_{j=2}^{n} (\underset{\substack{\| \\ j}}{t_j} - 1) = \sum_{j=2}^{n} (j-1) \circledcirc$$

let $k = j-1$, $j=2 \Rightarrow k=1$

$j=n \Rightarrow k=n-1$

$$\circledcirc \sum_{k=1}^{n-1} k = \frac{(n-1)((n+1)-1)}{2} = \frac{n(n-1)}{2}$$

$n \to n-1$

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$$

$$S = \sum_{k=1}^{n} k$$

$$2S = \sum_{k=1}^{n} k + \sum_{k=1}^{n} (n-k+1)$$

$$= \sum_{k=1}^{n} [\cancel{k} + (n-\cancel{k}+1)]$$

$$= \sum_{k=1}^{n} (n+1) = n(n+1)$$

$$\Rightarrow S = \frac{n(n+1)}{2}$$

Hence,

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1, \quad \sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

Then total running time of insertion sort in the worst case is

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) +$$

$$+ c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6 \cdot \frac{n(n-1)}{2} + c_7 \frac{n(n-1)}{2}$$

$$+ c_8(n-1) =$$

$$= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2}\right.$$

$$\left. - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n + (-c_2 - c_4 - c_5 - c_8)$$

$= an^2 + bn + c$, where a, b, c do not depend on n, only on costs c_i

$T(n) = an^2 + bn + c$: <u>quadratic function in n</u>

In this class, we will study mostly the worst case running time since it gives an upper bound, or the longest time an algorithm will work. For some algorithms, the worst case happen quite often (like finding an element in $A[1..n]$ that equals a given value v if no element in A equals v)

"Average case". Array A has random order of elements. When one wants to compare $A[j]$ with elements in $A[1..j-1]$ then approximately half of elements in subarray $A[1..j-1]$ is ~~sorted~~ less than $A[j]$, another half is ~~not sorted~~ greater than $A[j]$. Then $t_j \approx \frac{j}{2}$. In this case, total running time is still of order n^2, i.e. has terms proportional to n^2.

Order of growth / rate of growth

We neglected actual cost c_i of each line

$T(n) = an + b$: for best case

or

$T(n) = an^2 + bn + c$: worst case

For large n $(n \gg 1)$, $bn \ll an^2$, $c \ll an^2$

$$\Rightarrow \underbrace{an^2}_{\text{leading order term}} + bn + c \approx an^2 = \Theta(n^2)$$

order of growth or rate of growth

for large n, constant ~~or~~ of the highest power of n is not so important if one compares two algorithms with different order of growth

$\Theta(n^2)$ $\Theta(n^3)$

this algorithm will work faster for large n