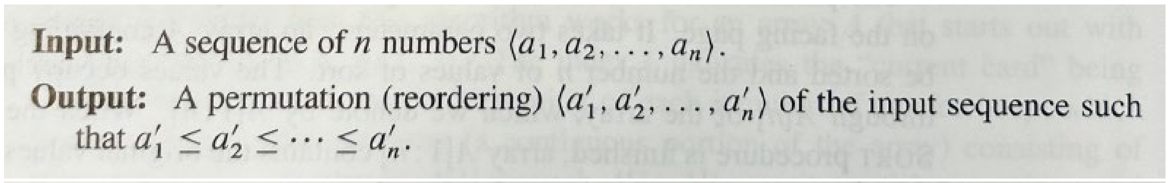
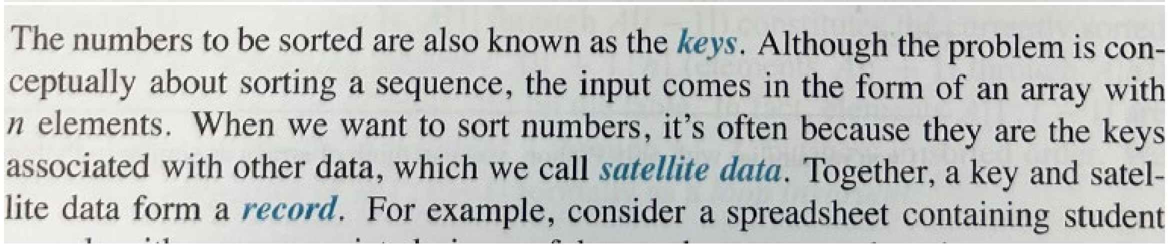
Sorting problem defined



Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \le a'_2 \le \cdots \le a'_n$.

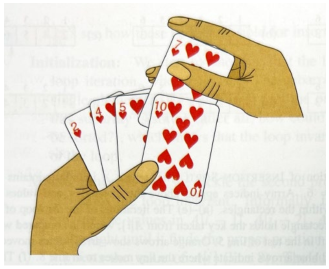


The numbers to be sorted are also known as the ***keys***. Although the problem is conceptually about sorting a sequence, the input comes in the form of an array with n elements. When we want to sort numbers, it's often because they are the keys associated with other data, which we call ***satellite data***. Together, a key and satellite data form a ***record***. For example, consider a spreadsheet containing student

The first sort we will discuss is (the aptly named) ***<u>Insertion-Sort</u>***

First, note that a list (or array) of length 1 is already sorted…

Description of Insertion-Sort using a deck of cards.



<u>In-class w/***assigned groups of four (4)***, write 0. and 1. on board as ground rule(s):</u>

0. No devices.
1. Intro: Name, POE, Icebreaker.
2. Pick someone on the group who will report out.
3. Work through the algorithm, “think like a computer!”
4. Try the Insertion-Sort algorithm on already sorted cards.
5. Try the Insertion-Sort algorithm on reverse sorted cards.
6. Discuss pseudocode to implement Insertion-Sort, assuming you are given an unsorted list.

Insertion-Sort is an ***in-place*** sort, which means at most a constant number of elements of the input array are ever stored outside of the array.

Remember the Random Access Machine (RAM) model:

- Instructions execute sequentially, one after another, no concurrency.
- Each instruction takes the same time to execute and has fixed cost.
- Each data access takes the same amount of time, not concerned with memory hierarchy.
- Data “in-core” and fixed word length.
- No “magic” commands like ***sort***.

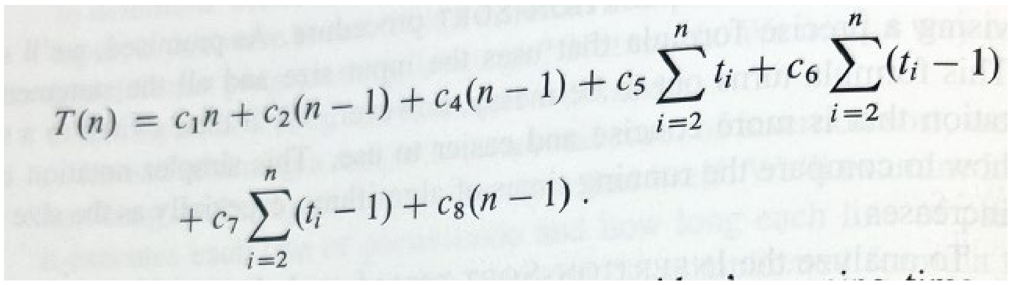
Based on description, develop Insertion-Sort pseudo-code. From the textbook:

	`Insertion-Sort(A,n)`	***constant * # executions***
1	`for i=2 to n:`	c_1 * n
2	`   key = A[i]`	c_2 * (n-1)
3	`// insert A[i] into sorted seq A[1]..A[i-1]`	0 * (n-1)
4	`   j = i-1`	c_4 * (n-1)
5	`   while j>0 and A[j]>key:`	c_5 * $(\sum_{i=2}^{n} t_i)$
6	`      A[j+1] = A[j]`	c_6 * $(\sum_{i=2}^{n} t_i - 1)$
7	`      j = j-1`	c_7 * $(\sum_{i=2}^{n} t_i - 1)$
8	`   A[j+1] = key`	c_8 * (n-1)

The subscript ***i*** indicates the "current card" being inserted into the hand.

What is t_i?
The while loop starting on line 5 runs a variable number of times, depending on the input array. We capture that by using a variable ***t***, and using subscripts, so for example t_5 would be the number of times the while loop runs when ***i=5***, and then in general t_i would be the number of times the while loop runs, for a given value of ***i***. In general we don't know t_i, except if we make some assumptions about the data (see below).

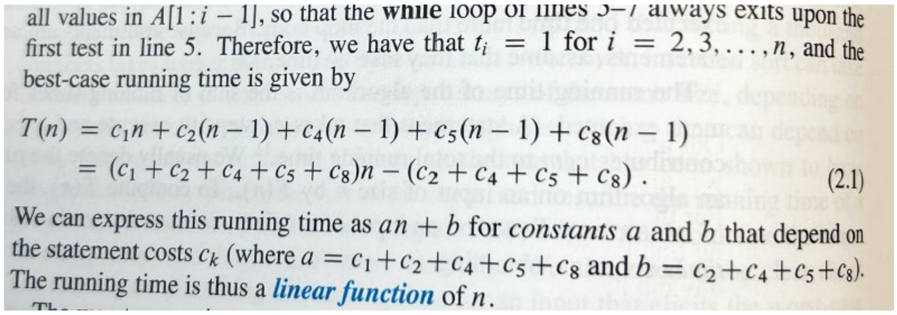
Derive recurrence relation ***T(n)***:



$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{i=2}^{n} t_i + c_6 \sum_{i=2}^{n} (t_i - 1) + c_7 \sum_{i=2}^{n} (t_i - 1) + c_8(n-1) .$$

Best and Worst cases:

Best case? Already sorted data. The while loop starting on line 5 is exited after the first test, so t_i = 1, which results in a linear function of the problem size ***n***.

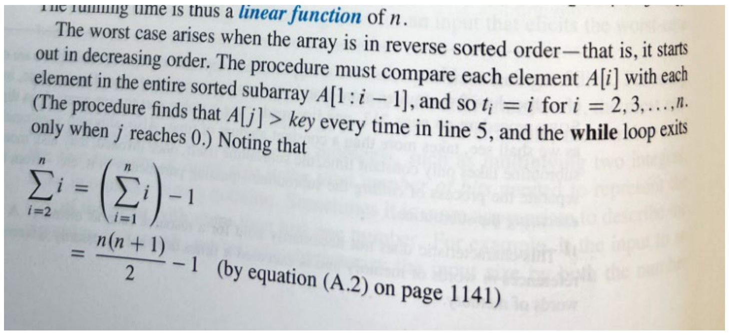


all values in $A[1:i-1]$, so that the **while** loop of lines 5–7 always exits upon the first test in line 5. Therefore, we have that $t_i = 1$ for $i = 2, 3, \ldots, n$, and the best-case running time is given by

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) . \qquad (2.1) \end{aligned}$$

We can express this running time as $an + b$ for *constants* a and b that depend on the statement costs c_k (where $a = c_1 + c_2 + c_4 + c_5 + c_8$ and $b = c_2 + c_4 + c_5 + c_8$). The running time is thus a ***linear function*** of n.

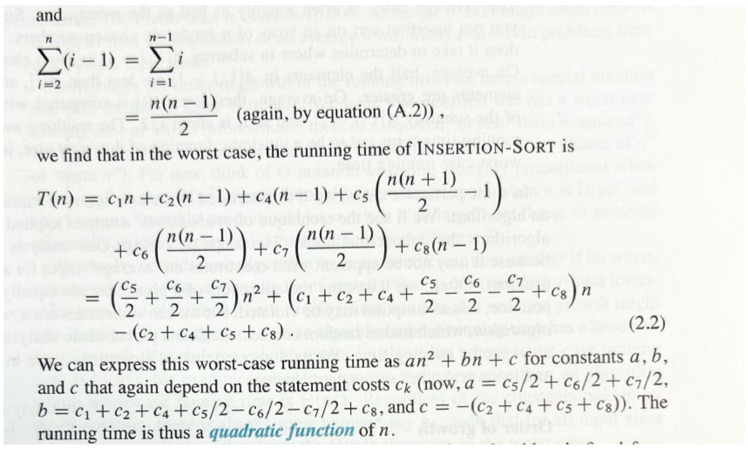
Worst case? Reverse sorted data, so ***t_i* =** i, which results in a quadratic function, ***n^2*** of the problem size ***n***.



The running time is thus a *linear function* of n.

The worst case arises when the array is in reverse sorted order—that is, it starts out in decreasing order. The procedure must compare each element $A[i]$ with each element in the entire sorted subarray $A[1:i-1]$, and so $t_i = i$ for $i = 2, 3, \ldots, n$. (The procedure finds that $A[j] > key$ every time in line 5, and the **while** loop exits only when j reaches 0.) Noting that

$$\begin{aligned}\sum_{i=2}^{n} i &= \left(\sum_{i=1}^{n} i\right) - 1 \\ &= \frac{n(n+1)}{2} - 1 \quad \text{(by equation (A.2) on page 1141)}\end{aligned}$$



and

$$\begin{aligned}\sum_{i=2}^{n} (i-1) &= \sum_{i=1}^{n-1} i \\ &= \frac{n(n-1)}{2} \quad \text{(again, by equation (A.2))},\end{aligned}$$

we find that in the worst case, the running time of INSERTION-SORT is

$$\begin{aligned}T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\ &\quad - (c_2 + c_4 + c_5 + c_8)\,. \end{aligned} \tag{2.2}$$

We can express this worst-case running time as $an^2 + bn + c$ for constants a, b, and c that again depend on the statement costs c_k (now, $a = c_5/2 + c_6/2 + c_7/2$, $b = c_1 + c_2 + c_4 + c_5/2 - c_6/2 - c_7/2 + c_8$, and $c = -(c_2 + c_4 + c_5 + c_8)$). The running time is thus a *quadratic function* of n.

This is a really detailed analysis.

What is a ***<u>shortcut</u>***, or intuitive approach?

For an <u>iterative</u> algorithm, find the line of code that executes the most often, and focus on it. How many loops is it embedded in? In insertion this line is inside two loops, so generally (except for Best Case) we have n^2.

Why do we usually consider the worst case running times for the algorithms we analyze?

- It's an upper-bound on running time for any input.
- "Murphy's Law," worst case seems to happen often.
- The average case is usually asymptotically as bad the worst case. For insertion sort, the average case is if ***t_i*** was equal to i/2, rather than i, and here the summations will still be ***n^2***.

How would we determine correctness? Using loop invariants. In this case, at the start of each iteration of the for-loop from lines 1-8, the subarray `A[1 : i-1]` consists of the elements originally in positions `1` thru `i-1`, but in sorted order. And then correctness would be proven via induction (see the text).

Starting with `insertionPoC.ipynb`, which includes the pseudocode above, write an `insertion` function. Step through the code and output below:

```
import random
N=5
A = list(range(N))
random.shuffle(A)
print(A)
insertion(A,True)
print(A)
```

```
[3, 2, 1, 0, 4]
[2, 3, 1, 0, 4]
[1, 2, 3, 0, 4]
[0, 1, 2, 3, 4]
[0, 1, 2, 3, 4]
[0, 1, 2, 3, 4]
```