Order Analysis of Algorithms

Debdeep Mukhopadhyay
IIT Kharagpur

Sorting problem

- Input: A sequence of n numbers, a1,a2,…,an
- Output: A permutation (reordering) (a1’,a2’,…,an’) of the input sequence such that a1’≤a2’ ≤… ≤an’
  - Comment: The number that we wish to sort are also known as keys

Insertion Sort

- Efficient for sorting small numbers

- In place sort: Takes an array A[0..n-1] (sequence of n elements) and arranges them in place, so that it is sorted.

It is always good to start with numbers

5	2	4	6	1	3

1	2	3	4	5	6

j=5

Invariant property in the loop:

At the start of each iteration of the algorithm, the subarray a[0...j-1] contains the elements originally in a[0..j-1] but in sorted order

Pseudo Code

- **Insertion-sort(A)**

```
for j=1 to (length(A)-1)
     do key = A[j]
#Insert A[j] into the sorted sequnce A[0...j-1]
     i=j-1
      while i>0 and A[i]>key
         do A[i+1]=A[i]
              i=i-1
      A[i+1]=key  //as A[i]<=key, so we place
                  //key on the right side of A[i]
```

Loop Invariants and Correctness of Insertion Sort

- **Initialization:** Before the first loop starts, j=1. So, A[0] is an array of single element and so is trivially sorted.

- **Maintenance:** The outer for loop has its index moving like j=1,2,…,n-1 (if A has n elements). At the beginning of the for loop assume that the array is sorted from A[0..j-1]. The inner while loop of the jth iteration places A[j] at its correct position. Thus at the end of the jth iteration, the array is sorted from A[0..j]. Thus, the invariance is maintained. Then j becomes j+1.
  - ***Also, using the same inductive reasoning the elements are also the same as in the original array in the locations A[0..j].***

Loop Invariants and Correctness of Insertion Sort

- **Termination:** The for loop terminates when j=n, thus by the previous observations the array is sorted from A[0..n-1] and the elements are also the same as in the original array.

Thus, the algorithm indeed sorts and is thus correct!

Analyzing Algorithms

The RAM Model

- A generic one processor Random Access Machine (RAM) model of computation.
- Instructions are executed sequentially (and not concurrently)
- We have to use the model so that we do not go too deep (into the machine instructions) and yet not abuse the notions (by say assuming that there exists a sorting instruction)

The RAM Model

- Our model has instructions commonly found in real computers:
  - arithmetic (add, subtract, multiply, divide)
  - data movement (load, store, copy)
  - control (conditional and unconditional branch, subroutine call and function)
- Each such instruction takes a constant time

Data types & Storage

- In the RAM model the data types are float and int.
- Assume the size of each block or word of data is so that an input of size n can be represented by word of **clog(n)** bits, $c \geq 1$
- $c \geq 1$, so that each word can hold the value of n.
- c cannot grow arbitrarily, because we cannot have one word storing huge amount of data and also which could be operated in constant time.

Gray areas in the RAM model

- Is exponentiation a constant time operation? NO
- Is computation of 2^n a constant time operation? Well…
- Many computers have a "shift left" operation by k positions (in constant time)
- Shift left by 1 position multiplies by 2. So, if I shift left 2, k times…I obtain 2^k in constant time !
  - (as long as k is no more than the word length).

Some further points on the RAM Model

- **We do not model the memory hierarchy**
  - No caches, pages etc
  - May be necessary for real computers and real applications. But the discussions are too specialized and we do use such modeling when required. As they are very complex and difficult to work with.
  - Fortunately, RAM models are excellent predictors! Still quite challenging. We require knowledge in logic, inductive reasoning, combinatorics, probability theory, algebra, and above all observation and intuition!

Lets analyze the Insertion sort

- The time taken to sort depends on the fact that we are sorting how many numbers
- Also, the time to sort may change depending upon whether the array is almost sorted (can you see if the array was sorted we had very little job).
- So, we need to define the meaning of the **input size** and **running time**.

Input Size

- Depends on the notion of the problem we are studying.
- Consider sorting of n numbers. The input size is the cardinal number of the set of the integers we are sorting.
- Consider multiplying two integers. The input size is the total number of bits required to represent the numbers.
- Sometimes, instead of one numbers we represent the input by two numbers. E.g. graph algorithms, where the input size is represented by both the number of edges (E) and the number of vertices (V)

Running Time

- Proportional to the Number of primitive operations or steps performed.
- Assume, in the pseudo-code a constant amount of time is required for each line.
- Assume that the ith line requires ci, where ci is a constant.
- Keep in mind the RAM model which says that there is no concurrency.

Run Time of Insertion Sort

Steps	Cost	Times
for j=1 to n-1	c_1	n
key=A[j]	c_2	n-1
i=j-1	c_3	n-1
while i>0 and A[i]>key	c_4	$\sum_{j=1}^{n-1} t_j$
do A[i+1]=A[i]	c_5	$\sum_{j=1}^{n-1} (t_j - 1)$
i=i-1	c_6	$\sum_{j=1}^{n-1} (t_j - 1)$
A[i+1]=key	c_7	(n-1)

In the RAM model the total time required is the sum of that for each statement:

$$T(n)=c_1 n + c_2(n-1) + c_3(n-1) + c_4\sum_{j=1}^{n-1} t_j + c_5\sum_{j=1}^{n-1}(t_j - 1) + c_6\sum_{j=1}^{n-1}(t_j - 1) + c_7(n-1)$$

Best Case

- **If the array is already sorted:**
  - *While* loop sees in 1 check that A[i]<key and so while loop terminates. Thus t_j=1 and we have:

$$T(n)=c_1 n + c_2(n-1) + c_3(n-1) + c_4\sum_{j=1}^{n-1} 1 + c_5\sum_{j=1}^{n-1}(1-1) + c_6\sum_{j=1}^{n-1}(1-1) + c_7(n-1)$$

$$=(c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7)$$

<u>The run time is thus a linear function of n</u>

Worst Case: The algorithm cannot run slower!

- If the array is arranged in reverse sorted array:
  - *While* loop requires to perform the comparisons with A[j-1] to A[0], that is $t_j = j$

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\sum_{j=1}^{n-1} j + c_5\sum_{j=1}^{n-1}(j-1) + c_6\sum_{j=1}^{n-1}(j-1) + c_7(n-1)$$

$$= \left(\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2}\right)n^2 + \left(c_1 + c_2 + c_3 - \frac{c_4}{2} - \frac{3c_5}{2} - \frac{3c_6}{2}\right)n + (c_5 + c_6 - c_2 - c_3 - c_7)$$

<u>The run time is thus a quadratic function of n</u>

Average Case

- Instead of an input of a particular type (as in best case or worst case), all the inputs of the given size are equally <u>*probable*</u> in such an analysis.
  - E.g. coming back to our insertion sort, if the elements in the array A[0..j-1] are randomly chosen. We can assume that half the elements are greater than A[j] while half are less. On the average, thus $t_j = j/2$. Plugging this value into T(n) still leaves it quadratic. Thus, in this case average case is equivalent to a worst case run of the algorithm.
  - <u>*Does this always occur?*</u> NO. The average case may tilt towards the best case also.