CS 255, Spring 2014, SJSU

Course introduction: Analysis of Insertion Sort

Fernando Lobo

Agenda

- Course introduction.
- Warmup problem to get us started in analysis of algorithms.
- Review a problem that you should already know.
- Describe it in pseudocode (as in the textbook).
- Start using asymptotic notation for algorithm running time.

Course introduction

- This is a course on design and analysis of algorithms.
- Not a programming class, but assumes you know how to program.
- Assumes you know elementary algorithms and data structures.
- Grading: 7 HWs, 2 midterms, 1 final exam.
- See greensheet and tentative schedule in canvas for further details.

The sorting problem

- Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$
- Output: A permutation $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that:
$$a'_1 \leq a'_2 \leq \ldots \leq a'_n$$
- Assume sequence is stored in an array.
- Example:
  - Input: $\langle 8, 2, 4, 9, 3, 6 \rangle$
  - Output: $\langle 2, 3, 4, 6, 8, 9 \rangle$

Insertion Sort

- Good for sorting small arrays.
- Works more or less in the same way we would sort a hand of cards when picking them one by one.
  - The cards in our hand are always sorted.
  - When we pick a new card we insert it in the correct position.

Pseudocode (convention: arrays start at position 1)

INSERTION-SORT(A)

for $j = 2$ **to** $length[A]$
    $key = A[j]$
    // Insert $A[j]$ into the sorted sequence $A[1\,..\,j-1]$
    $i = j - 1$
    while $i > 0$ and $A[i] > key$
        $A[i+1] = A[i]$
        $i = i - 1$
    $A[i+1] = key$

Example

```
8 2 4 9 3 6    // input
2 8 4 9 3 6    // end of 1st iteration
2 4 8 9 3 6    // end of 2nd iteration
2 4 8 9 3 6    // end of 3rd iteration
2 3 4 8 9 6    // end of 4th iteration
2 3 4 6 8 9    // end of 5th iteration
```


Correctness

- Loop Invariant: At the beginning of each iteration of the **for** loop, the subarray $A[1\,..\,j-1]$ is sorted.
- We use this invariant to prove the algorithm is correct.

Correctness

Need to prove 3 things about the loop invariant:

1. Initialization: It is true before the 1st iteration.
2. Maintenance: If it is true before a given iteration, it remains true at the beginning of the next iteration.
3. Termination: When the loop ends, the array $A[1..n]$ is sorted.

Loop invariants

The utilization of loop invariants is analogous to mathematical induction:

- Initialization $\rightarrow$ base case.
- Maintenance $\rightarrow$ indutive step.
- Termination: $\rightarrow$ no correspondence in mathematical induction. Here the "induction" stops when the loop terminates.

How to analyze the algorithm's running time?

It depends of the input:

- Sorting 100000 numbers takes more time than sorting 30 numbers.
- It can take different time for arrays of the same size (ex: INSERTION-SORT is faster if the input is already sorted.)

How to analyze the algorithm's running time?

The input size depends on the problem:

- Usually it is the number of input elements (n in the sorting problem).
- But it can be something else:
  - In graph algorithms, the input size is usually expressed in terms of 2 quantities: number of nodes, and number of edges.

Running time

The running time of an algorithm on a given input is the number of primitive operations executed.

- Primitive operations: assignments, comparisions, etc.
- Each line of the pseudocode requires a constant amount of time (independent of the input size).
- Different lines might take different amounts of time.
- Function calls also take constant time, but its execution might not.

Analysis of INSERTION-SORT

- For $j = 2, 3, \ldots, n$, let t_j be the number of times that the **while** loop test is executed for that value of j.
- (NOTE: **for** and **while** loop tests are executed one more time than the loop bodies. Why?)
- Running time depends on the $t_{j's}$ (which depend on the input).

Pseudocode

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


Number of times each line is executed

- line 1: n
- line 2: $n-1$
- line 3: $n-1$
- linel 4: $n-1$
- line 5: $t_2 + t_3 + \ldots + t_n$
- line 6: $(t_2 - 1) + (t_3 - 1) + \ldots + (t_n - 1)$
- line 7: $(t_2 - 1) + (t_3 - 1) + \ldots + (t_n - 1)$
- line 8: $n-1$

Running time

- $\sum_i$ (cost of line i) $\times$ (num times line i is executed)
- Let c_i be the cost of line i.
- Let $T(n)$ be the running time of INSERTION-SORT.

$$\begin{aligned} T(n) = {} & c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) \\ & + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) \end{aligned}$$

Best case

- The input array is already sorted. That is,
- $A[i] \leq key$ at the beginning of the **while** loop $\implies$ all $t_{j's} = 1$

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

- $T(n) = an + b$, with a and b constants.
- $\implies$ $T(n)$ is a linear function of n.

Worst case

- The input array is reverse sorted.
- Need to compare *key* with all elements to the left of position $j \implies j - 1$ comparisons.
- **while** loop ends when $i = 0$ ($j - 1$ comparisons, j tests) $\implies$ $t_j = j$

$$\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} j$$

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

(arithmetic progressions)

Worst case

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

- $T(n) = an^2 + bn + c$, with a, b, c constants.
- $\implies$ $T(n)$ is a quadratic function of n.

Average case

- In general it's difficult or even impossible to calculate.
- Need several assumptions. For example, all input sequences are equally likely (which can be far from true).

Average case

- Suppose the input of INSERTION-SORT is an array of n numbers randomly generated from a uniform distribution.
- On average, *key* in $A[j]$ is less than half of the elements in $A[1 \,..\, j-1]$ and is larger than the other half.
- $\implies$ on average the **while** loop has to look at half of the subarray $A[1 \,..\, j-1] \implies t_j = j/2$
- The running time is approx half of the worst case running time. Still a quadratic function of n.

Best, worst, and average cases

- Best case analysis is usually irrelevant.
- Why?
- $\implies$ We can have a very bad sorting algorithm which has linear running time in the best case, and exponential running time in the worst (and average) case.
- Any hint of such an algorithm?

Best, worst, and average cases

- We are usually interested in the worst case analysis: The longest running time on any given input of size n.
- Why?
  - Gives us an upper bound for the total execution time, regardless of the input.
  - Average case is usually as bad as the worst case, and much harder to analyze.
  - In many cases, the worst case happens very often (ex: searching for an element that is not in a collection.)

Order of growth

- Make several simplifications and focus on the essentials.
  - eliminate lower order terms.
  - ignore the coefficient of the higher order term.
- Example: The worst case running time of INSERTION-SORT is given by $an^2 + bn + c$
  - eliminating lower order terms $\implies an^2$
  - eliminating the coefficient $\implies n^2$
- IMPORTANT: Cannot say that $T(n) = n^2$
- $\implies$ but we can say that $T(n)$ has a growth rate proportional to n^2.

Order of growth

- We say that $T(n) = \Theta(n^2)$ to capture the notion that the order of growth is n^2.
- This is called asymptotic analysis.
- We usually say that an algorithm is more efficient than another if its worst case running time is of a lower order (i.e. has a lower asymptotic complexity).