CS 375 – Analysis of Algorithms

Professor Eric Aaron

Lecture – M W 1:00pm

Lecture Meeting Location: Davis 117

Business

- Smaller Assignment 0 due already
  - Graded work will be returned to you in your SubmittedWork folder
  - I'll email the entire class when it's been returned
- Smaller Assignment 1 out, due Sept. 19
- Problem Set 0 out, due Sept. 21

For the semester, please assume all PS and SA deadlines are 1pm on deadline day, whether or not it's explicitly stated in lecture

Business, pt. 2

- Project 1 out today
  - *Please read the full project assignment!* Instructions are given there
  - *Please note project-specific lateness policy* on assignment sheet
- Some key points about Proj1
  - Deadline: *End of day (11:59pm)* on *September 28*
  - Part of it: Asymptotic complexity analysis on methods in a Java class—you're given the source code
  - Part of it: Analyzing and conjecturing about asymptotic complexity when you're given data from runtime performance, *not* source code
- **Project 1 is to be done in teams of 2 or 3**
  - IMPORTANT: By end of day Saturday, Sept. 17, one person from each team should email me *and everyone on the team* to let me know they're teaming up
  - If you'd like my help finding a team for you, please let me know!

Time Complexity of Insertion Sort

See CLRS, Ch 2.2, pg. 26-27

- What's the time complexity of Insertion Sort?
  - Our default is to look at the *worst-case* complexity of the algo, on an input of size *n*

Time complexity from adding [cost * times]:

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

- So, what's the *worst case* complexity?

Plug in $t_j = j$ ... (Note: summation is same for c_6, c_7)

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

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

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

In worst case, $t_j = j$ each time, so T(n) is order of n^2

We'd say Insertion Sort is an n^2 algorithm

Time Complexity of Insertion Sort

- What's the time complexity of Insertion Sort?
  - Our default is to look at the *worst-case* complexity of the algo, on an input of size *n*

Add up the [cost * times] for each row… what do we get?

Time complexity from adding [cost * times]:

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

- So, what's the *best case* complexity?

In best case, $t_j = 1$ each time, so…

- This means Insertion Sort is *linear* in the best case!
- But we don't consider it a linear algo, because that's not its worst case time complexity

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

Space Complexity of Insertion Sort

- While we're at it, what's the *space complexity* of Insertion Sort?
  - That is, how much space is used beyond the storage for the input?

```
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
```

Space Complexity of Insertion Sort

- While we're at it, what's the *space complexity* of Insertion Sort?
  - That is, how much space is used beyond the storage for the input?

- Other than input A, there are a few variables for storage
  - *j, key, i*
  - So, constant space complexity… and this is true in *best case*, *worst case*, and *average case*

Space complexity: *constant*

```
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
```

Back to Time Complexity

- So… that counted…
  - We just counted numbers of operations for best case and worst case of Insertion Sort

    Well, we kinda did… those c_i constants weren't super precise….
  - How does that help us talk about which algorithms are faster than others?
- Big idea: Consider time complexity on large input sizes n
  - Lots of algorithms are usable on small inputs
  - The algorithms that are faster on large inputs are the ones we're going to consider fastest
  - … but how do we define that, for rigorous algorithm analysis?

Introduction to Time Complexity Analysis of Algorithms

Let's take the big-picture view. How, in principle, could we measure the time efficiency of an *algorithm?*

- Could use a timer or stopwatch (or clock… or calendar…) to measure how fast a program is *on a given size of input*…
  - (called *empirical analysis*…)
  - But that doesn't really measure the *algorithm* speed
  - How much clock time passes is dependent on things other than just the algorithm (processor speed, memory access speed, etc.)!
- Better idea: Count how many operations an algorithm does *on a given size of input* as a measure of how long it takes!
  - Assume some unit of time for each operation
  - This gives a measure of time usage (i.e., speed) that is dependent upon the *algorithm as written*, not external factors!

Introduction to Time Complexity Analysis of Algorithms, cont.

- But even that kind of counting depends on how an algorithm is implemented
  - If the insertion sort idea is implemented with even minor differences…
  - Operation count could change… but the algorithm is the essentially the same, independent of minor coding details!
  - We don't want to say the *algorithm* has different speeds just because of many slightly different implementations.
- We want to discuss algorithm time complexity at a level a little bit more abstract than just a literal count of operations
  - If somehow we could capture the essential character of how many operations insertion sort takes …
    - on input of a given size (e.g., an array of size n)
  - …without getting caught up in small details…

Asymptotic Analysis / Big-O Notation

- With insertion sort, if we gloss over minor details, we can see the number of operations (worst case) is *on the order of* n^2
  - i.e., it is $c*n^2$ + (lower order terms)
  - … for some constant c
  - … where n is the size of the input

$$\left(\frac{c_5}{2}+\frac{c_6}{2}+\frac{c_7}{2}\right)n^2+\left(c_1+c_2+c_4+\frac{c_5}{2}-\frac{c_6}{2}-\frac{c_7}{2}+c_8\right)n - (c_2+c_4+c_5+c_8)\,.$$

- Definition: An algorithm runs in time $O(f(n))$ (read: "order of f(n)") means:
  - There exist $c > 0$, $n_0 > 0$ s.t. …
  - …for all $n \geq n_0$, the running time of the algorithm is less than $c*f(n)$
  - (Basically, that means that for every input "big enough," the running time is less than a constant times $f(n)$)
- This running time measure captures some essential characteristic of an algorithm
  - $O(n^2)$ algorithms differ from $O(n^3)$, from $O(n \log n)$, etc.

Asymptotic Examples

- In what *Big-O classes* are the following?
  - n + 7
  - $n^2 + 3n$
  - For the next one, we use the shorthand that an algorithm is in f(n) + O(g(n)) if the running time is f(n) + t(n) for some t(n) in O(g(n)). Similarly for O(f(n)) * O(g(n)), or other arithmetic combinations.
  - $(n \lg n + O(n) + O(1)) * O(n^2)$
- Formal definition describes what we intuitively mean by not worrying about lower-order terms

Common complexity measures and how they relate to input sizes

- Algorithms are sometimes described by their time complexity. There are
  - Logarithmic algorithms
  - Quadratic algorithms
  - Exponential algorithms
  - Factorial algorithms
  - etc.
- To see which kind is fastest, see how these functions grow with increases in the input size:

n	$\log_{10} n$	n^2	2^n	n!
1	0	1	2	1
10	1	100	1024	3628800
50	1.70	2500	1.13e15	3.04e64
100	2	10000	1.27e30	9.44e157

Conventional Wisdom about Big-O Classes

- If two algorithms are in different big-O classes, then there seems to be something substantially different about their speeds
  - Even though, for some small values of n, an $O(2^n)$ algorithm could be faster than an $O(n^2)$ algorithm…
  - It is nonetheless true that 2^n *grows* faster than n^2…
  - Thus, an $O(2^n)$ algorithm is, in a relevant sense, *inherently* slower than an $O(n^2)$ algorithm

Important Vocab (see CLRS, pg. 28): These functions of *n* have very different *orders of growth*—i.e., how fast they grow as *n* gets larger

- For an $O(n)$ algorithm (called "linear")
  - Doubling the input size does what to the running time?
  - Increasing input size by factor of 100 does what to running time?
- For an $O(n^2)$ algorithm ("quadratic")
  - Doubling the input size does what to the running time?
  - Increasing input size by factor of 100 does what to running time?
- For an $O(2^n)$ algorithm ("exponential")
  - Doubling the input size does what to the running time?