CS 375 – Analysis of Algorithms

Professor Eric Aaron

Lecture – M W 1:00pm

Lecture Meeting Location: Davis 117

Business

- Smaller Assignment 0 due by beginning of class Sept. 14
  - Follow file naming conventions (See assignment sheet)
  - Submit to your SubmittedWork Folder in Google Drive space (see email)
  - Graded work will be returned to you in your SubmittedWork folder, too
- Problem Set 0 out tonight, due Sept. 21 (by beginning of class, as usual)
- Any questions about submission instructions for PS's and SA's?

- TA Hours, held in Davis 122 (also posted on course website)
  - Sundays 4-5:30pm
  - Tuesdays 6:30-8pm
  - Thursdays 7-8pm
- If anyone has trouble reading lecture notes from course website, please let me know!
- Thank you for your emails! I will to reply to each of them soon if I haven't already

What's The Fastest Sorting Algorithm?

- Of all the things we do with data, sorting is among the most important
  - Improves usability
  - Real-world data is often found sorted!
- Sorting is among the most important algorithms
  - How do we sort efficiently?
  - Many classic algos to choose from!
  - **What sorting algos do you know?**

This is a sneaky-useful real-world tip… pre-sorting data before using them as input to an algorithm can sometimes enable more efficient algorithms.

But sometimes it doesn't help!

Analyzing the algo and understanding the input are what helps us make good design choices.

Reminder: Specification of the sorting problem

Input: A sequence L of n numbers (a_0, …, a_{n-1})

Output: A sequence L' of n numbers (b_0, …, b_{n-1}) that re-orders the input sequence (perhaps leaving them unchanged) such that $b_0 \leq b_1 \leq \ldots \leq b_{n-1}$

What's The Fastest Sorting Algorithm?

- Of all the things we do with data, sorting is among the most important
  - Improves usability
  - Real-world data is often found sorted!
- Sorting is among the most important algorithms
  - How do we sort efficiently?
  - Many classic algos to choose from!
  - **What sorting algos do you know?**

This is a sneaky-useful real-world tip… pre-sorting data before using them as input to an algorithm can sometimes enable more efficient algorithms.

But sometimes it doesn't help!

Analyzing the algo and understanding the input are what helps us make good design choices.

- The answer to what sorting algo to use for time efficiency depends on understanding properties of algorithms and properties of input data….

Insertion Sort (In Pseudocode)

Are you familiar with pseudocode?

- Are you familiar with Insertion Sort?
- In English, how would you describe how it works?

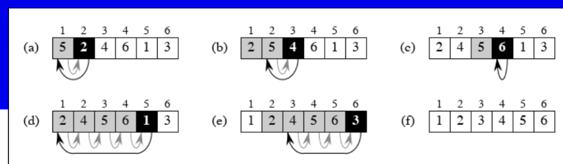
```
INSERTION-SORT(A)
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted
        sequence A[1 .. j − 1].
    i = j − 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```

Be sure to read CLRS Ch. 2.1 about pseudocode conventions—those conventions will apply throughout CS375.

Insertion Sort (In Pseudocode)

- Are you familiar with Insertion Sort?
- In English, how would you describe how it works?

Are you familiar with pseudocode?

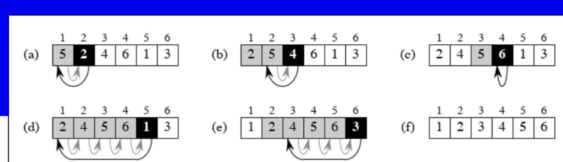


```
INSERTION-SORT(A)
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted
        sequence A[1..j-1].
    i = j - 1
    while i > 0 and A[i] > key
        A[i+1] = A[i]
        i = i - 1
    A[i+1] = key
```

Be sure to read CLRS Ch. 2.1 about pseudocode conventions—those conventions will apply throughout CS375.

Insertion Sort (In Pseudocode)

- Is Insertion Sort the fastest sorting algo?
  - And how would we know?



```
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
```

What do you recall about *time complexity* from CS231?

Let's do some *complexity analysis* warmup / review…

Insertion Sort (In Pseudocode)

- Is Insertion Sort the fastest sorting algo?
  - And how would we know? Time complexity analyses!

- Note: for loops are *inclusive* of boundaries—here, the algo goes through the loop for every value of j from 2 to A.length *inclusive.*
- At the end of a for loop, the ***index variable*** is incremented beyond the boundary condition—here, the loop ends with j having value A.length + 1

INSERTION-SORT(A)	*cost*	*times*
for $j = 2$ **to** $A.length$	c_1	n
$key = A[j]$	c_2	$n-1$
// Insert $A[j]$ into the sorted sequence $A[1 \,.\,.\, j-1]$.	0	$n-1$
$i = j-1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
$A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$i = i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$A[i+1] = key$	c_8	$n-1$

We'll return to Insertion Sort on the next slide. But first…

Analyzing Algorithms: ~~Sum~~ Some Mathematical Foundations

- Summations
  - Arithmetic: $\Sigma_{i=1:n}\, i = ??$
  - Geometric: $\Sigma_{i=0:n}\, c^i = ??$ (for constant $c \neq 1$)

(Sum puns are just too easy….)

Sum Nights—fun.

What about for c = 1?

See CLRS Appendix A.1 for these summation formulas, and please let me know if you have any questions!

These summation formulas will be relevant on our next Smaller Assignment, to be assigned soon and due next Monday, Sept. 19

Insertion Sort (In Pseudocode)

- Is Insertion Sort the fastest sorting algo?
  - And how would we know? Time complexity analyses!

- Note: for loops are *inclusive* of boundaries—here, the algo goes through the loop for every value of j from 2 to A.length *inclusive.*
- At the end of a for loop, the ***index variable*** is incremented beyond the boundary condition—here, the loop ends with j having value A.length + 1

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 $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
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	$n-1$

n **stands for the input size—here, the length of array A[1..n]**

Is there an off-by-one error on line 1? Why is it *n* times when the loop goes from *2* to *n*?

Insertion Sort (In Pseudocode)

- Is Insertion Sort the fastest sorting algo?
  - And how would we know? Time complexity analyses!

- Note: for loops are *inclusive* of boundaries—here, the algo goes through the loop for every value of j from 2 to A.length *inclusive.*
- At the end of a for loop, the ***index variable*** is incremented beyond the boundary condition—here, the loop ends with j having value A.length + 1

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 $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
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	$n-1$

n **stands for the input size—here, the length of array A[1..n]**

t_j stands for the number of times the *while* loop condition is tested for a given value of *j*

Insertion Sort (In Pseudocode)

- Is Insertion Sort the fastest sorting algo?
  - And how would we know? Time complexity analyses!
- What's the time complexity of Insertion Sort?

INSERTION-SORT(A)	*cost*	*times*
for $j = 2$ **to** $A.length$	c_1	n
$key = A[j]$	c_2	$n-1$
// Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
$i = j - 1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
$A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$i = i - 1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$A[i+1] = key$	c_8	$n-1$

What are the ***best case*** and ***worst case*** inputs to Insertion Sort, for a given input size *n*?

Insertion Sort (In Pseudocode)

- Is Insertion Sort the fastest sorting algo?
  - And how would we know? Time complexity analyses!
- What's the time complexity of Insertion Sort?
  - Best case complexity? Worst case complexity?

INSERTION-SORT(A)	*cost*	*times*
for $j = 2$ **to** $A.length$	c_1	n
$key = A[j]$	c_2	$n-1$
// Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
$i = j - 1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
$A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$i = i - 1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$A[i+1] = key$	c_8	$n-1$

What are the ***best case*** and ***worst case*** inputs to Insertion Sort, for a given input size *n*?

- Best case: A[1..n] already in sorted order
- Worst case: A[1..n] in ***REVERSE*** sorted order
- Do you see why?

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?

	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 $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
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	$n-1$

- **Lines 1-4 and 8, all together, add up to something *linear*—some constant times *n***
- **Lines 5-7, the *inner loop*, are a bit more complicated… they add up to something that involves that summation formula (see Appendix A.1)…**

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?

	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 $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
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	$n-1$

- **Lines 1-4 and 8, all together, add up to something *linear*—some constant times *n***
- **Lines 5-7, the *inner loop*, are a bit more complicated… they add up to something that involves that summation formula (see Appendix A.1)…**
- **… but the term being added (t_j) is always *j* in the worst case**

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]:

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

INSERTION-SORT(A)	*cost*	*times*
for $j = 2$ **to** $A.length$	c_1	n
$key = A[j]$	c_2	$n-1$
// Insert $A[j]$ into the sorted sequence $A[1\,..\,j-1]$.	0	$n-1$
$i = j - 1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
$A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$i = i - 1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$A[i+1] = key$	c_8	$n-1$

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

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?

In worst case, $t_j = j$ each time, so…

What does our expression for runtime *T(n)* turn into?

INSERTION-SORT(A)	*cost*	*times*
for $j = 2$ **to** $A.length$	c_1	n
$key = A[j]$	c_2	$n-1$
// Insert $A[j]$ into the sorted sequence $A[1\,..\,j-1]$.	0	$n-1$
$i = j - 1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
$A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$i = i - 1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$A[i+1] = key$	c_8	$n-1$

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

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)

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

In worst case, $t_j = j$ each time, so…

What does our expression for runtime *T(n)* turn into?

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

See CLRS, Ch 2.2, pg. 26-27

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

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)

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

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

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

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?

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 $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
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	$n-1$

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…

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 $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
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	$n-1$

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…

Plug in $t_j = 1$, which means $(t_j - 1) = 0$

So, the c_6 and c_7 terms go away, and the rest go… well, like the below

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

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

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