CS146: Data Structures and Algorithms
Lecture 2

GETTING STARTED- INSERTION SORT

INSTRUCTOR: KATERINA POTIKA
CS SJSU

Sorting Problem

- Input: A sequence of n numbers $a_1, a_2, \ldots, a_n$
- Output: A permutation a'_1 , $a'_2, \ldots, a'_n$ of the input sequence such that

$$a'_1 \leq a'_2 \leq \cdots \leq a'_n$$

- Example this instance: 31, 41, 59, 26, 41, 58

Sorting is a Fundamental problem

- Why?
- Can you think practical cases where you need sorting?

Correctness

- For *every* input instance, halts with correct output
- **Correct** algorithm then solves the problem

Many algorithms for the same problem

- Which is the best for a given application
  - Number of items
  - Somehow sorted
  - Restrictions on the values
  - Storage to be used
  - etc

Algorithms for sorting

- Can you name some sorting algorithms?

An Example: Insertion Sort


```
InsertionSort(A, n)
  for i = 2 to n
        key = A[i]
        j = i - 1
        while (j > 0) and (A[j] > key)
              A[j+1] = A[j]
              j = j - 1

     A[j+1] = key
```

An Example: Insertion Sort

30	10	40	20
1	2	3	4

i = ∅ j = ∅ key = ∅
A[j] = ∅ A[j+1] = ∅

```
InsertionSort(A, n)
  for i = 2 to n
      key = A[i]
      j = i - 1
      while (j > 0) and (A[j] > key)
            A[j+1] = A[j]
            j = j - 1
      A[j+1] = key
```

Correctness proof

- Use a loop invariant to understand why an algorithm gives the correct answer.

Loop invariant (for InsertionSort)

At the start of each iteration of the "outer" **for** loop (indexed by i) the subarray A[1..i-1] consists of the elements originally in A[1..i-1] but in sorted order.

Correctness proof

- To proof correctness with a loop invariant we need to show three things:

➢ Initialization
Invariant is true prior to the first iteration of the loop.

➢ Maintenance
If the invariant is true before an iteration of the loop, it remains true before the next iteration.

➢ Termination
When the loop terminates, the invariant (usually along with the reason that the loop terminated) gives us a useful property that helps show that the algorithm is correct.

Correctness proof

InsertionSort(A)

```
initialize: sort A[1]
for i = 2 to A.length
    do key = A[i]
        j = i -1
        while j > 0  and A[j] > key
            do  A[j+1] = A[j]
                j = j -1
        A[j +1] = key
```

Loop invariant

At the start of each iteration of the "outer" **for** loop (indexed by j) the subarray A[1..i-1] consists of the elements originally in A[1..i-1] but in sorted order.

Initialization

Just before the first iteration, i = 2   A[1..i-1] = A[1], which is the element originally in A[1], and it is trivially sorted.

Correctness proof

InsertionSort(A)

```
initialize: sort A[1]
for i = 2 to A.length
    do key = A[i]
        j = i -1
        while j > 0  and A[j] > key
            do  A[j+1] = A[j]
                j = j -1
        A[j +1] = key
```

Loop invariant

At the start of each iteration of the “outer” **for** loop (indexed by j) the subarray A[1..i-1] consists of the elements originally in A[1..i-1] but in sorted order.

Maintenance

Strictly speaking need to prove loop invariant for “inner” **while** loop. Instead, note that body of **while** loop moves A[i-1], A[i-2], A[i-3], and so on, by one position to the right until proper position of key is found (which has value of A[i]) ⇒ invariant maintained.

Correctness proof

InsertionSort(A)

```
1.   initialize: sort A[1]
2.   for i = 2 to A.length
3.       do key = A[i]
4.            j = i -1
5.            while j > 0  and A[j] > key
6.                do  A[j+1] = A[j]
7.                     j = j -1
8.            A[j +1] = key
```

Loop invariant

At the start of each iteration of the “outer” **for** loop (indexed by j) the subarray A[1..i-1] consists of the elements originally in A[1..i-1] but in sorted order.

Termination

The outer **for** loop ends when i > n; this is when i = n+1 ➡ i-1 = n.
Plug n for i-1 in the loop invariant ➡ the subarray A[1..n] consists of the elements originally in A[1..n] in sorted order.

Insertion sort Algorithm

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

InsertionSort is an in place algorithm: the numbers are rearranged within the array with only constant extra space.

Analyzing Insertion Sort

- $T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 T + c_5(T - (n-1)) + c_6(T - (n-1)) + c_7(n-1)$

  $= c_8 T + c_9 n + c_{10}$
- What can T be?
  - Best case -- inner loop body never executed
    - $t_i = 1$ ➔ T(n) is a linear function
  - Worst case -- inner loop body executed for all previous elements
    - $t_i = i$ ➔ T(n) is a quadratic function
  - Average case
    - ???

Analysis

- Simplifications
  - Ignore actual and abstract statement costs
  - *Order of growth* is the interesting measure:
    - Highest-order term is what counts
    - Asymptotic analysis!
    - As the input size grows larger it is the high order term that dominates

Upper Bound Notation

- We say InsertionSort's run time is $O(n^2)$
  - Properly we should say run time is ***in*** $O(n^2)$
  - Read O as "Big-O" (you'll also hear it as "order")
- In general a function
  - $f(n)$ is $O(g(n))$ if there exist positive constants c and n_o such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_o$
- Formally
  - $O(g(n)) = \{ f(n)\text{: } \exists$ positive constants c and n_o such that $f(n) \leq c \cdot g(n)\ \forall\ n \geq n_o$

Insertion Sort Is O(n^2)

- Proof
  - Suppose runtime is $an^2 + bn + c$
    - If any of a, b, and c are less than 0 replace the constant with its absolute value

$$an^2 + bn + c \le (a + b + c)n^2 + (a + b + c)n + (a + b + c)$$

$$\le 3(a + b + c)n^2 \text{ for } n \ge 1$$

Let $c' = 3(a + b + c)$ and let $n_o = 1$

- Question
  - Is InsertionSort O(n^3)?
  - Is InsertionSort O(n)?

Lower Bound Notation

- We say InsertionSort's run time is $\Omega(n)$
- In general a function
  - $f(n)$ is $\Omega(g(n))$ if $\exists$ positive constants c and n_o such that $0 \leq c \cdot g(n) \leq f(n) \quad \forall\ n \geq n_o$
- Proof:
  - Suppose run time is $an + b$
    - Assume a and b are positive (what if b is negative?)
  - $an \leq an + b$

Asymptotic Tight Bound

- A function f(n) is Θ(g(n)) if ∃ positive constants c_1, c_2, and n_o such that

$$c_1\, g(n) \le f(n) \le c_2\, g(n) \;\forall\; n \ge n_o$$

- Theorem
  - f(n) is Θ(g(n)) iff f(n) is both O(g(n)) and Ω(g(n))
  - Proof: someday

Practice

10, 9, 8, 7, 6, 5

What is the order in which insertion sorts

A. 5 10 9 8 7 6
   5 6 10 9 8 7
   5 6 7 10 9 8
   5 6 7 8 10 9
   5 6 7 8 9 10

B. 9 10 8 7 6 5
   8 9 10 7 6 5
   7 8 9 10 6 5
   6 7 8 9 10 5
   5 6 7 8 9 10

Example: Fibonacci numbers

- Fibonacci numbers F(n), for n = 0, 1, 2, …, are
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, …
  - Rabbits in an island

Algorithm 1: Use recursion

- **Formal definition: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …**

$$F(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F(n-1) + F(n-2) & \text{if } n > 1 \end{cases}$$

```
int fib (int n){
    if(n == 0 || n == 1)
        return n;
    else
        return ( fib (n-1) + fib (n-2) );
}
```

Alg 2: For Loop

$$F(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F(n-1) + F(n-2) & \text{if } n > 1 \end{cases}$$

```
int fib (int n){
    if (n == 0 || n == 1){
        return n;
    }else{
        int tmp1 = 0, tmp2 = 1, result;
        for (int i = 2; i <= n; i++){
            result = tmp1 + tmp2;
            tmp1 = tmp2;
            tmp2 = result;
        }
        return result;
    }
}
```

result

tmp1 tmp2

0, 1, 1, 2, 3, 5, 8, 13, 21, …

Which One is Better?

Algorithm 1

```
int fib (int n){
        if(n == 0 || n == 1)
                return n;
        else
                return ( fib (n-1) + fib (n-2) );
}
```

Algorithm 2

```
int fib (int n){
        if (n == 0 || n == 1){
                return n;
        }else{
                int tmp1 = 0, tmp2 = 1, result;
                for (int i=2; i<=n; i++){
                        result = tmp1 + tmp2;
                        tmp1 = tmp2;
                        tmp2 = result;
                }
                return result;
        }
}
```

Analysis of Algorithm 1

Algorithm 1

```
int fib (int n){
        if(n == 0 || n == 1)
                return n;
        else
                return ( fib (n-1) + fib (n-2) );
}
```

fib(100) — $1 = 2^0$

fib(99) fib(98) — $2 = 2^1$

fib(98) fib(97) fib(97) fib(96) — $4 = 2^2$

100 fib(97) fib(96) … fib(96) fib(95) … — $8 = 2^3$

⋮

fib(1) fib(0)

Exponential to *n*

Analysis of Algorithm 2


```
int fib (int n){
        if (n == 0 || n == 1){
                return n;
        }else{
                int tmp1 = 0, tmp2 = 1, result;
                for (int i=2; i<=n; i++){
                        result = tmp1 + tmp2;
                        tmp1 = tmp2;
                        tmp2 = result;
                }
                return result;
        }
}
```

$3*(n-1)=3n-3$

Linear to *n*

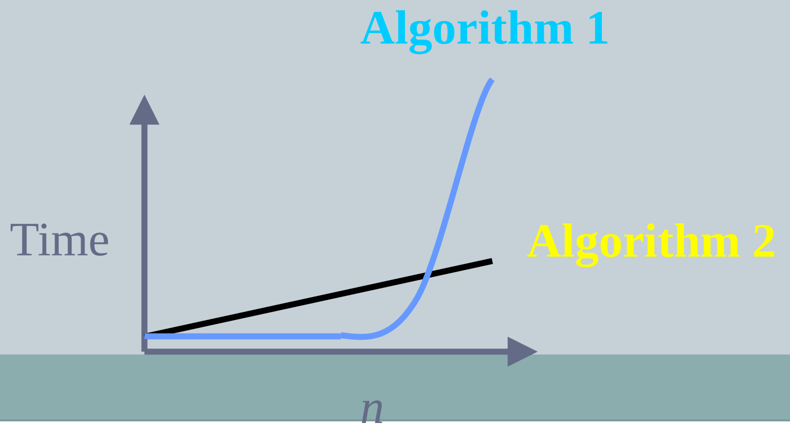
result

tmp1 tmp2

0, 1, 1, 2, 3, 5, 8, 13, 21, …

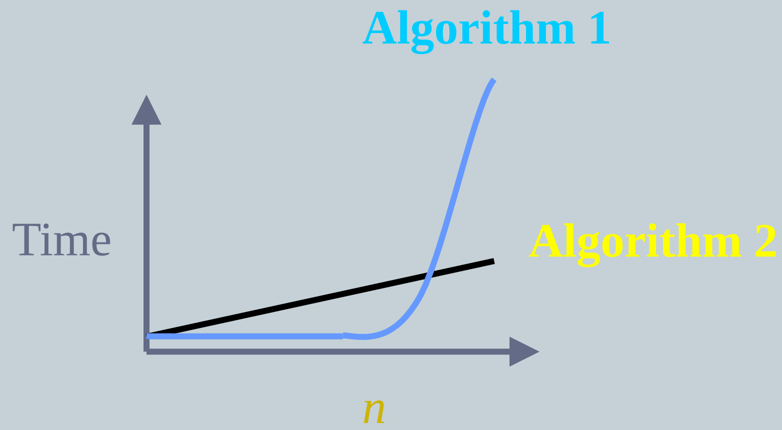
Which One is Better?

- Algorithm 2 runs faster in average and worst cases.
- If the Fibonacci number is quite small, Algorithm 1.



Which One is Better?

- We are more interested in how an algorithm behaves as the problem size goes large.
  - All algorithms behave similar under a small problem size.



Appendix

An Example: Insertion Sort

30	10	40	20
1	2	3	4

i = ∅ j = ∅ key = ∅
A[j] = ∅ A[j+1] = ∅

```
InsertionSort(A, n)
  for i = 2 to n
      key = A[i]
      j = i - 1
      while (j > 0) and (A[j] > key)
            A[j+1] = A[j]
            j = j - 1
      A[j+1] = key
```

An Example: Insertion Sort

30	10	40	20
1	2	3	4

i = 2   j = 1   key = 10
A[j] = 30   A[j+1] = 10

```
InsertionSort(A, n)
  for i = 2 to n
      key = A[i]
      j = i - 1
      while (j > 0) and (A[j] > key)
            A[j+1] = A[j]
            j = j - 1
      A[j+1] = key
```

An Example: Insertion Sort

30	30	40	20
1	2	3	4

i = 2 j = 1 key = 10

A[j] = 30 A[j+1] = 30

```
InsertionSort(A, n)
  for i = 2 to n
        key = A[i]
        j = i - 1
        while (j > 0) and (A[j] > key)
              A[j+1] = A[j]
              j = j - 1
        A[j+1] = key
```

An Example: Insertion Sort

30	30	40	20
1	2	3	4

i = 2   j = 1   key = 10

A[j] = 30   A[j+1] = 30

```
InsertionSort(A, n)
  for i = 2 to n
        key = A[i]
        j = i - 1
        while (j > 0) and (A[j] > key)
              A[j+1] = A[j]
              j = j - 1
        A[j+1] = key
```

An Example: Insertion Sort

30	30	40	20
1	2	3	4

$i = 2$ $j = 0$ $key = 10$

$A[j] = \varnothing$ $A[j+1] = 30$

```
InsertionSort(A, n)
  for i = 2 to n
      key = A[i]
      j = i - 1
      while (j > 0) and (A[j] > key)
            A[j+1] = A[j]
            j = j - 1
      A[j+1] = key
```

An Example: Insertion Sort

30	30	40	20
1	2	3	4

$i = 2$ $j = 0$ $key = 10$

$A[j] = \varnothing$ $A[j+1] = 30$

```
InsertionSort(A, n)
  for i = 2 to n
        key = A[i]
        j = i - 1
        while (j > 0) and (A[j] > key)
              A[j+1] = A[j]
              j = j - 1
        A[j+1] = key
```

An Example: Insertion Sort

10	30	40	20
1	2	3	4

i = 2   j = 0   key = 10

A[j] = ∅   A[j+1] = 10

```
InsertionSort(A, n)
  for i = 2 to n
      key = A[i]
      j = i - 1
      while (j > 0) and (A[j] > key)
            A[j+1] = A[j]
            j = j - 1
      A[j+1] = key
```

An Example: Insertion Sort

10	30	40	20
1	2	3	4

i = 3 j = 0 key = 10

A[j] = ∅ A[j+1] = 10

```
InsertionSort(A, n)
  for i = 2 to n
        key = A[i]
        j = i - 1
        while (j > 0) and (A[j] > key)
              A[j+1] = A[j]
              j = j - 1
        A[j+1] = key
```

An Example: Insertion Sort

10	30	40	20
1	2	3	4

i = 3   j = 0   key = 40
A[j] = ∅   A[j+1] = 10

```
InsertionSort(A, n)
  for i = 2 to n
        key = A[i]
        j = i - 1
        while (j > 0) and (A[j] > key)
              A[j+1] = A[j]
              j = j - 1
        A[j+1] = key
```

An Example: Insertion Sort

10	30	40	20
1	2	3	4

i = 3 j = 0 key = 40
A[j] = ∅ A[j+1] = 10

```
InsertionSort(A, n)
  for i = 2 to n
      key = A[i]
      j = i - 1
      while (j > 0) and (A[j] > key)
            A[j+1] = A[j]
            j = j - 1
      A[j+1] = key
```

An Example: Insertion Sort

10	30	40	20
1	2	3	4

i = 3   j = 2   key = 40
A[j] = 30   A[j+1] = 40

```
InsertionSort(A, n)
  for i = 2 to n
      key = A[i]
      j = i - 1
      while (j > 0) and (A[j] > key)
            A[j+1] = A[j]
            j = j - 1
      A[j+1] = key
```

An Example: Insertion Sort

10	30	40	20
1	2	3	4

i = 3 j = 2 key = 40

A[j] = 30 A[j+1] = 40

```
InsertionSort(A, n)
  for i = 2 to n
      key = A[i]
      j = i - 1
      while (j > 0) and (A[j] > key)
            A[j+1] = A[j]
            j = j - 1
      A[j+1] = key
```

An Example: Insertion Sort

10	30	40	20
1	2	3	4

i = 3 j = 2 key = 40

A[j] = 30 A[j+1] = 40

```
InsertionSort(A, n)
  for i = 2 to n
      key = A[i]
      j = i - 1
      while (j > 0) and (A[j] > key)
            A[j+1] = A[j]
            j = j - 1
      A[j+1] = key
```

An Example: Insertion Sort

10	30	40	20
1	2	3	4

$i = 4$ $j = 2$ key = 40
$A[j] = 30$ $A[j+1] = 40$

```
InsertionSort(A, n)
  for i = 2 to n
      key = A[i]
      j = i - 1
      while (j > 0) and (A[j] > key)
            A[j+1] = A[j]
            j = j - 1
      A[j+1] = key
```

An Example: Insertion Sort

10	30	40	20
1	2	3	4

i = 4 j = 2 key = 20

A[j] = 30 A[j+1] = 40

```
InsertionSort(A, n)
  for i = 2 to n
      key = A[i]
      j = i - 1
      while (j > 0) and (A[j] > key)
            A[j+1] = A[j]
            j = j - 1
      A[j+1] = key
```

An Example: Insertion Sort

10	30	40	20
1	2	3	4

i = 4   j = 2   key = 20

A[j] = 30   A[j+1] = 40

```
InsertionSort(A, n)
  for i = 2 to n
      key = A[i]
      j = i - 1
      while (j > 0) and (A[j] > key)
            A[j+1] = A[j]
            j = j - 1
      A[j+1] = key
```

An Example: Insertion Sort

10	30	40	20
1	2	3	4

i = 4   j = 3   key = 20
A[j] = 40   A[j+1] = 20

```
InsertionSort(A, n)
  for i = 2 to n
      key = A[i]
      j = i - 1
      while (j > 0) and (A[j] > key)
            A[j+1] = A[j]
            j = j - 1
      A[j+1] = key
```

An Example: Insertion Sort

10	30	40	20
1	2	3	4

i = 4 j = 3 key = 20
A[j] = 40 A[j+1] = 20

```
InsertionSort(A, n)
  for i = 2 to n
        key = A[i]
        j = i - 1
        while (j > 0) and (A[j] > key)
              A[j+1] = A[j]
              j = j - 1
        A[j+1] = key
```

An Example: Insertion Sort

10	30	40	40
1	2	3	4

i = 4 j = 3 key = 20

A[j] = 40 A[j+1] = 40

```
InsertionSort(A, n)
  for i = 2 to n
      key = A[i]
      j = i - 1
      while (j > 0) and (A[j] > key)
            A[j+1] = A[j]
            j = j - 1
      A[j+1] = key
```

An Example: Insertion Sort

10	30	40	40
1	2	3	4

i = 4 j = 3 key = 20
A[j] = 40 A[j+1] = 40

```
InsertionSort(A, n)
  for i = 2 to n
      key = A[i]
      j = i - 1
      while (j > 0) and (A[j] > key)
            A[j+1] = A[j]
            j = j - 1
      A[j+1] = key
```

An Example: Insertion Sort

10	30	40	40
1	2	3	4

i = 4 j = 3 key = 20
A[j] = 40 A[j+1] = 40

```
InsertionSort(A, n)
  for i = 2 to n
      key = A[i]
      j = i - 1
      while (j > 0) and (A[j] > key)
            A[j+1] = A[j]
            j = j - 1
      A[j+1] = key
```

An Example: Insertion Sort

10	30	40	40
1	2	3	4

i = 4 j = 2 key = 20

A[j] = 30 A[j+1] = 40

```
InsertionSort(A, n)
  for i = 2 to n
      key = A[i]
      j = i - 1
      while (j > 0) and (A[j] > key)
            A[j+1] = A[j]
            j = j - 1
      A[j+1] = key
```

An Example: Insertion Sort

10	30	40	40
1	2	3	4

i = 4 j = 2 key = 20

A[j] = 30 A[j+1] = 40

```
InsertionSort(A, n)
  for i = 2 to n
        key = A[i]
        j = i - 1
        while (j > 0) and (A[j] > key)
              A[j+1] = A[j]
              j = j - 1
        A[j+1] = key
```

An Example: Insertion Sort

10	30	30	40
1	2	3	4

i = 4   j = 2   key = 20
A[j] = 30   A[j+1] = 30

```
InsertionSort(A, n)
  for i = 2 to n
        key = A[i]
        j = i - 1
        while (j > 0) and (A[j] > key)
              A[j+1] = A[j]
              j = j - 1
        A[j+1] = key
```

An Example: Insertion Sort

10	30	30	40
1	2	3	4

i = 4 j = 2 key = 20

A[j] = 30 A[j+1] = 30

```
InsertionSort(A, n)
  for i = 2 to n
        key = A[i]
        j = i - 1
        while (j > 0) and (A[j] > key)
              A[j+1] = A[j]
              j = j - 1
        A[j+1] = key
```

An Example: Insertion Sort

10	30	30	40
1	2	3	4

i = 4 j = 1 key = 20

A[j] = 10 A[j+1] = 30

```
InsertionSort(A, n)
  for i = 2 to n
      key = A[i]
      j = i - 1
      while (j > 0) and (A[j] > key)
            A[j+1] = A[j]
            j = j - 1
      A[j+1] = key
```

An Example: Insertion Sort

10	30	30	40
1	2	3	4

i = 4 j = 1 key = 20

A[j] = 10 A[j+1] = 30

```
InsertionSort(A, n)
  for i = 2 to n
        key = A[i]
        j = i - 1
        while (j > 0) and (A[j] > key)
              A[j+1] = A[j]
              j = j - 1
        A[j+1] = key
```

An Example: Insertion Sort

10	20	30	40
1	2	3	4

i = 4   j = 1   key = 20
A[j] = 10   A[j+1] = 20

```
InsertionSort(A, n)
  for i = 2 to n
      key = A[i]
      j = i - 1
      while (j > 0) and (A[j] > key)
            A[j+1] = A[j]
            j = j - 1
      A[j+1] = key
```

An Example: Insertion Sort

10	20	30	40
1	2	3	4

i = 4   j = 1   key = 20
A[j] = 10   A[j+1] = 20

```
InsertionSort(A, n)
  for i = 2 to n
      key = A[i]
      j = i - 1
      while (j > 0) and (A[j] > key)
            A[j+1] = A[j]
            j = j - 1
      A[j+1] = key
```

Done!