ALGORITHMS & ADVANCED DATA STRUCTURES (#2)

GETTING STARTED-INSERTION SORT

ADAPTED FROM
CS 146 SJSU (KATERINA POTIKA)

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
  $$a'_1 \le a'_2 \le \cdots \le a'_n$$
- Example this instance: 31, 41, 59, 26, 41, 58

Sorting

- Sorting. Given n elements, rearrange in ascending order.

Obvious sorting applications.
- List files in a directory.
- Organize an MP3 library.
- List names in a phone book.
- Display Google PageRank results.

Problems become easier once sorted.
- Find the median.
- Find the closest pair.
- Binary search in a database.
- Identify statistical outliers.
- Find duplicates in a mailing list.

Non-obvious sorting applications.
- Data compression.
- Computer graphics.
- Interval scheduling.
- Computational biology.
- Minimum spanning tree.
- Supply chain management.
- Simulate a system of particles.
- Book recommendations on Amazon.
- Load balancing on a parallel computer.
- . . .

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

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
```

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
        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
```

Loop invariant

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.

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
        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
```

Loop invariant

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.

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)

```
initialize: sort A[1]
for i = 2 to A.length
        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
```

Loop invariant

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.

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

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

- What can t be?
  - Best case -- inner loop body never executed
    - $t_j = 1$ ➔ T(n) is a linear function
  - Worst case -- inner loop body executed for all previous elements
    - $t_j = i$ ➔ T(n) is a quadratic function

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
  - $O(g(n)) = \{ f(n): \exists$ 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 $O(n)$?

Big O Fact

- A polynomial of degree k is $O(n^k)$
- Proof:
  - Suppose $f(n) = b_k n^k + b_{k-1} n^{k-1} + \ldots + b_1 n + b_0$
    - Let $a_i = | b_i |$
  - $f(n) \le a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$

$$\le \quad n^k \sum a_i \frac{n^i}{n^k} \quad \le \quad n^k \sum a_i \quad \le \quad cn^k$$

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 $0 \le c \cdot g(n) \le f(n) \quad \forall\, n \ge n_o$

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))

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*

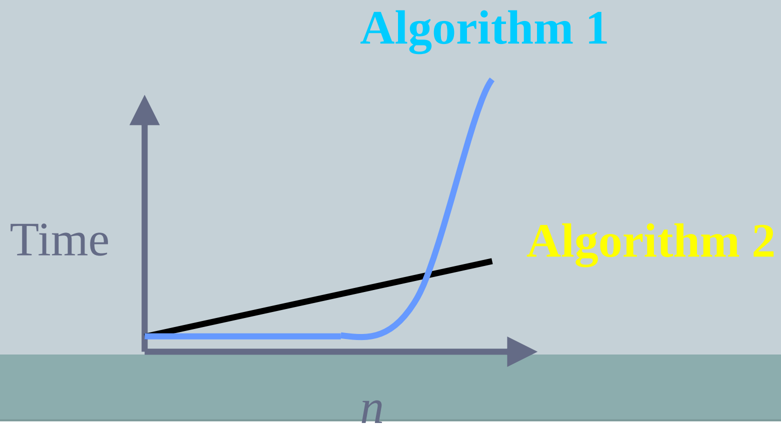
result

tmp1 tmp2

0, 1, 1, 2, 3, 5, 8, 13, 21, …

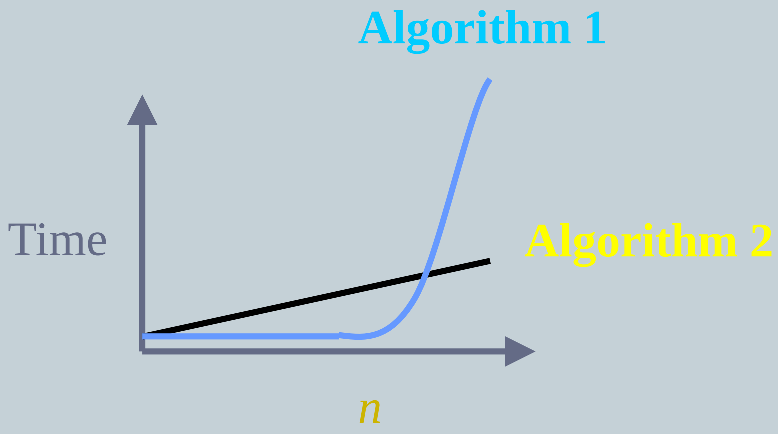
Which One is Better?

- 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.



Selection Sort

- A relatively easy to understand algorithm
- Sorts an array in ***passes***
  - Each pass selects the next smallest element
  - At the end of the pass, places it where it belongs
- Efficiency is $O(n^2)$, hence called a ***quadratic*** sort
- Performs:
  - $O(n^2)$ comparisons
  - $O(n)$ exchanges (swaps)