Insertion Sort

Review: Asymptotic Performance

- *Asymptotic performance*: How does algorithm behave as the problem size gets very large?
  - Running time
  - Memory/storage requirements
  - Remember that we use the RAM model:
    - All memory equally expensive to access
    - No concurrent operations
    - All reasonable instructions take unit time
      - Except, of course, function calls
    - Constant word size
      - Unless we are explicitly manipulating bits

Review: Running Time

- Number of primitive steps that are executed
  - Except for time of executing a function call most statements roughly require the same amount of time
  - We can be more exact if need be
- Worst case vs. average case

Introduction to Algorithm design and analysis

Example: sorting problem.

Input: a sequence of n number $<a_1, a_2, \ldots, a_n>$

Output: a permutation (reordering) $<a_1', a_2', \ldots, a_n'>$

such that $a_1' \leq a_2' \leq \ldots \leq a_n'$.

Different sorting algorithms:

Insertion sort and Mergesort.

Efficiency comparison of two algorithms

- Suppose n=10^6 numbers:
  - Insertion sort: $c_1 n^2$
  - Merge sort: $c_2 n$ (lg n)
  - Best programmer (c_1=2), machine language, one billion/second computer A.
  - Bad programmer (c_2=50), high-language, ten million/second computer B.
  - 2 $(10^6)^2$ instructions/10^9 instructions per second = 2000 seconds.
  - 50 (10^6 lg 10^6) instructions/10^7 instructions per second $\approx$ 100 seconds.
  - Thus, merge sort on B is 20 times faster than insertion sort on A!
  - If sorting ten million numbers, 2.3 days VS. 20 minutes.
- Conclusions:
  - Algorithms for solving the same problem can differ dramatically in their efficiency.
  - much more significant than the differences due to hardware and software.

Algorithm Design and Analysis

- Design an algorithm
  - Prove the algorithm is correct.
    - Loop invariant.
    - Recursive function.
    - Formal (mathematical) proof.
- Analyze the algorithm
  - Time
    - Worse case, best case, average case.
    - For some algorithms, worst case occurs often, average case is often roughly as bad as the worst case. So generally, worse case running time.
  - Space
- Sequential and parallel algorithms
  - Random-Access-Model (RAM)
  - Parallel multi-processor access model: *PRAM*

Insertion Sort Algorithm (cont.)

```
INSERTION-SORT(A)
for j = 2 to length[A]
   do key ← A[j]
        //insert A[j] to sorted sequence A[1..j-1]
         i ← j-1
        while i >0 and A[i]>key
              do A[i+1] ← A[i]  //move A[i] one position right
                  i ← i-1
       A[i+1] ← key
```

Correctness of Insertion Sort Algorithm

- Loop invariant
  - At the start of each iteration of the for loop, the subarray A[1..*j*-1] contains original A[1..*j*-1] but in sorted order.
- Proof:
  - Initialization : *j*=2, A[1..*j*-1]=A[1..1]=A[1], sorted.
  - Maintenance: each iteration maintains loop invariant.
  - Termination: *j*=*n*+1, so A[1..*j*-1]=A[1..*n*] in sorted order.

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

$i = \varnothing$ $j = \varnothing$ $key = \varnothing$
$A[j] = \varnothing$ $A[j+1] = \varnothing$

```
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] = ∅ 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] = ∅ 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 = 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!

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
  }
}
```

How many times will this loop execute?

Analysis of Insertion Sort

INSERTION-SORT(A)	*cost*	*times*
1. **for** j = 2 to length[A]	c_1	n
2. **do** *key* ← A[j]	c_2	n-1
3. //insert A[j] to sorted sequence A[1..j-1]	0	n-1
4. $i \leftarrow j$-1	c_4	n-1
5. **while** i >0 and A[i]>*key*	c_5	$\sum_{j=2}^{n} t_j$
6. **do** A[i+1] ← A[i]	c_6	$\sum_{j=2}^{n}(t_j - 1)$
7. $i \leftarrow i$-1	c_7	$\sum_{j=2}^{n}(t_j - 1)$
8. A[i+1] ← *key*	c_8	$n - 1$

(t_j is the number of times the while loop test in line 5 is executed for that value of j)

The total time cost $T(n)$ = sum of *cost* × *times* in each line

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

Analysis of Insertion Sort (cont.)

- Best case cost: already ordered numbers
  - t_j=1, and line 6 and 7 will be executed 0 times
  - $T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$

    $=(c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) = cn + c'$
- Worst case cost: reverse ordered numbers
  - $t_j=j$,
  - so $\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} j = n(n+1)/2-1$, and $\sum_{j=2}^{n}(t_j - 1) = \sum_{j=2}^{n}(j - 1) = n(n-1)/2$, and
  - $T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n(n+1)/2 -1) + + c_6(n(n-1)/2 -1) + c_7(n(n-1)/2) + c_8(n-1) =((c_5 + c_6 + c_7)/2)n_2 +(c_1 + c_2 + c_4 + c_5/2 - c_6/2 - c_7/2 + c_8)n - (c_2 + c_4 + c_5 + c_8) = an^2+bn+c$
- Average case cost: random numbers
  - in average, $t_j = j/2$. $T(n)$ will still be in the order of n^2, same as the worst case.

Insertion Sort

Statement	Effort
`InsertionSort(A, n) {`	
`  for i = 2 to n {`	$c_1 n$
`     key = A[i]`	$c_2(n-1)$
`     j = i - 1;`	$c_3(n-1)$
`     while (j > 0) and (A[j] > key) {`	$c_4 T$
`          A[j+1] = A[j]`	$c_5(T-(n-1))$
`          j = j - 1`	$c_6(T-(n-1))$
`     }`	0
`     A[j+1] = key`	$c_7(n-1)$
`  }`	0
`}`	

$T = t_2 + t_3 + \ldots + t_n$ where t_i is number of while expression evaluations for the i^{th} for loop iteration

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
      - Remember, we are doing 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_0 such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$
- Formally
  - $O(g(n)) = \{ f(n)$: $\exists$ positive constants c and n_0 such that $f(n) \leq c \cdot g(n)\ \forall\ n \geq n_0$

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
  - $an^2 + bn + c \le (a + b + c)n^2 + (a + b + c)n + (a + b + c)$
  - $\le 3(a + b + c)n^2$ for $n \ge 1$
  - Let $c' = 3(a + b + c)$ and let $n_0 = 1$
- 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_0 such that $0 \le c \cdot g(n) \le f(n) \quad \forall\ n \ge n_0$
- Proof:
  - Suppose run time is $an + b$
    - Assume a and b are positive (what if b is negative?)
  - $an \le an + b$

Up Next

- Solving recurrences
  - Substitution method
  - Master theorem