CS141: Intermediate Data Structures and Algorithms

Analysis of Algorithms

Amr Magdy
Analyzing Algorithms

1. Algorithm Correctness
   a. Termination
   b. Produces the correct output for all possible input.

2. Algorithm Performance
   a. Either runtime analysis,
   b. or storage (memory) space analysis
   c. or both
Algorithm Correctness

Sorting problem

- Input: an array A of n numbers
- Output: the same array in ascending sorted order (smallest number in A[1] and largest in A[n])
Algorithm Correctness

› Sorting problem
  › Input: an array A of n numbers
  › Output: the same array in ascending sorted order (smallest number in A[1] and largest in A[n])

› Insertion Sort

```
INSERTION-SORT(A, n)
  for j = 2 to n
    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
```
Algorithm Correctness

- How does insertion sort work?
## Algorithm Correctness

| 5 | 2 | 4 | 6 | 1 | 3 |
Algorithm Correctness
Algorithm Correctness

5 2 4 6 1 3

1 2 3 4 5 6

5 2 4 6 1 3

1 2 3 4 5 6

2 5 4 6 1 3
Algorithm Correctness

1 2 3 4 5 6

5 2 4 6 1 3

1 2 3 4 5 6

5 2 4 6 1 3

1 2 3 4 5 6

2 5 4 6 1 3

1 2 3 4 5 6

2 4 5 6 1 3
Algorithm Correctness

5 2 4 6 1 3

1 2 3 4 5 6

5 2 4 6 1 3

2 5 4 6 1 3

2 4 5 6 1 3

2 4 5 6 1 3
Algorithm Correctness
Algorithm Correctness
Algorithm Correctness

- Is insertion sort a correct algorithm?
Algorithm Correctness

- Is insertion sort a correct algorithm?
  - Does it halt?
  - Does it produce correct output for all possible input?
Algorithm Correctness

- Is insertion sort a correct algorithm?
  - Does it halt? Yes
    - Two deterministically bounded loops, no infinite loops involved
  - Does it produce correct output for all possible input?

**Insertion-Sort** $(A, n)$

```plaintext
for j = 2 to n
  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
  // Insert whole key.
  A[i + 1] = key
```
Algorithm Correctness

› Is insertion sort a correct algorithm?
  › Does it halt? Yes
  › Does it produce correct output for all possible input?
    › Will check through loop invariants
Algorithm Correctness

› Is insertion sort a correct algorithm?
› Loop invariant:
  › It is a property that is true before and after each loop iteration.
Algorithm Correctness

› Is insertion sort a correct algorithm?
› Loop invariant:
  › It is a property that is true before and after each loop iteration.
› Insertion sort loop invariant (ISLI):
  › The first (j-1) array elements A[1..j-1] are:
    (a) the original (j-1) elements, and (b) sorted.

\[
\text{INSERTION-SORT}(A, n)
\]

\[
\text{for } j = 2 \text{ to } n \\
key = A[j] \\
// Insert } A[j] \text{ into the sorted sequence } A[1 \ldots j - 1]. \\
i = j - 1 \\
\text{while } i > 0 \text{ and } A[i] > key \\
i = i - 1 \\
A[i + 1] = key
\]
Algorithm Correctness

- Is insertion sort a correct algorithm?
  - If ISLI correct, then insertion sort is correct
  - How?
    - Halts and produces the correct output after \((n-1)\) iterations
Algorithm Correctness

Is insertion sort a correct algorithm?
  - If ISLI correct, then insertion sort is correct
  - How?
    - Halts and produces the correct output after \((n-1)\) iterations

Loop invariant (LI) correctness

1. Initialization:
   LI is true prior to the 1st iteration.

2. Maintenance:
   If LI true before the iteration, it remains true before the next iteration

3. Termination:
   After the loop terminates, the output is correct.
Algorithm Correctness

- **ISLI:** The first \((j-1)\) array elements \(A[1..j-1]\) are:
  (a) the original \((j-1)\) elements, and (b) sorted.

1. **Initialization:**
   Prior to the 1\(^{st}\) iteration, \(j=2\), the first \((2-1)=1\) elements is sorted.

2. **Maintenance:**
   The \((j-1)^{th}\) iteration inserts the \(j^{th}\) element in a sorted order, so after
   the iteration, the first \((j-1)\) elements remains the same and sorted.

3. **Termination:**
   The loop terminates after \((n-1)\) iterations, \(j=n+1\), so the first \(n\) elements are sorted, then the output is correct.

\[
\text{INSERTION-SORT}(A, n)
\]

\[
\text{for } j = 2 \text{ to } n
\]

\[
key = A[j]
\]

\[
// \text{ Insert } A[j] \text{ into the sorted sequence } A[1..j-1].
\]

\[
i = j - 1
\]

\[
\text{while } i > 0 \text{ and } A[i] > key
\]

\[
A[i + 1] = A[i]
\]

\[
i = i - 1
\]

\[
A[i + 1] = key
\]
Algorithm Correctness

ISLI: The first \((j-1)\) array elements \(A[1..j-1]\) are:
(a) the original \((j-1)\) elements, and (b) sorted.

1. Initialization:
   Prior to the 1\(^{\text{st}}\) iteration, \(j=2\), the first \((2-1)=1\) elements is sorted.

2. Maintenance:
   The \((j-1)\)\(^{\text{th}}\) iteration inserts the \(j^{\text{th}}\) element in a sorted order, so after
   the iteration, the first \((j-1)\) elements remains the same and sorted.

3. Termination:
   The loop terminates after \((n-1)\) iterations, \(j=n+1\), so the first \(n\) elements are sorted, then the output is correct.

   \text{INSERTION-SORT}(A, n)

   \begin{align*}
   &\text{for } j = 2 \text{ to } n \\
   &\quad \text{key} = A[j] \\
   &\quad // \text{ Insert } A[j] \text{ into the sorted sequence } A[1 \ldots j-1]. \\
   &\quad i = j - 1 \\
   &\quad \text{while } i > 0 \text{ and } A[i] > \text{key} \\
   &\quad \quad A[i + 1] = A[i] \\
   &\quad \quad i = i - 1 \\
   &\quad A[i + 1] = \text{key}
   \end{align*}
Analyzing Algorithms

1. Algorithm Correctness
   a. Termination
   b. Produces the correct output for all possible input.

2. Algorithm Performance
   a. Either runtime analysis,
   b. or storage (memory) space analysis
   c. or both
Algorithms Performance Analysis

- Which criteria should be taken into account?
  - Running time
  - Memory footprint
  - Disk IO
  - Network bandwidth
  - Power consumption
  - Lines of codes
  - …
Algorithms Performance Analysis

Which criteria should be taken into account?

- Running time
- Memory footprint
- Disk IO
- Network bandwidth
- Power consumption
- Lines of codes
- ...
Average Case vs. Worst Case

Running Time

Different inputs of the same size

Worst case
Average case
Best case
Insertion Sort Best Case
Insertion Sort Best Case

- Input array is sorted
Insertion Sort Best Case

- Input array is sorted

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

(n-1) operations
Insertion Sort Best Case

- Input array is sorted

**Algorithm**

\[
\text{INSERTION-SORT}(A, n)
\]

\[
\text{for } j = 2 \text{ to } n \quad \frac{c_1}{n-1}
\]

\[
\text{key} = A[j] \quad \frac{c_2}{n-1}
\]

\[
// \text{ Insert } A[j] \text{ into the sorted sequence } A[1 \ldots j-1]. \quad 0
\]

\[
i = j - 1 \quad \frac{c_3}{n-1}
\]

\[
\text{while } i > 0 \text{ and } A[i] > \text{key} \quad \frac{c_4}{n-1}
\]

\[
A[i + 1] = A[i] \quad \frac{1}{1}
\]

\[
i = i - 1
\]

\[
A[i + 1] = \text{key} \quad \frac{c_5}{n-1}
\]

\[
T(n) = (n-1)(c_1 + c_2 + 0 + c_3 + 1(c_4 + 0) + c_5)
\]

\[
T(n) = cn - c, \quad \text{const } c = c_1 + c_2 + c_3 + c_4 + c_5
\]
Insertion Sort Worst Case
Insertion Sort Worst Case

Input array is reversed

6 5 4 3 2 1
5 6 4 3 2 1
4 5 6 3 2 1
3 4 5 6 2 1
2 3 4 5 6 1
1 2 3 4 5 6
Insertion Sort Worst Case

Input array is reversed

Insertion-Sort \((A, n)\)

\[
\text{for } j = 2 \text{ to } n \quad \rightarrow \quad c_1 \\
\text{key} = A[j] \quad \rightarrow \quad c_2 \\
\text{// Insert } A[j] \text{ into the sorted sequence } A[1 \ldots j - 1]. \quad \rightarrow \quad 0 \\
i = j - 1 \quad \rightarrow \quad c_3 \\
\text{while } i > 0 \text{ and } A[i] > key \quad \rightarrow \quad c_4 \\
\quad A[i + 1] = A[i] \quad \rightarrow \quad c_5 \\
\quad i = i - 1 \quad \rightarrow \quad c_6 \\
\quad A[i + 1] = \text{key} \quad \rightarrow \quad c_7
\]
Insertion Sort Worst Case

Input array is reversed

\[ T(n) = (n-1)*(c_1+c_2+0+c_3+i*(c_4+c_5+c_6)+c_7) \]
\[ T(n) = (n-1)*(c_1+c_2+0+c_3+c_7) + \sum_i^*(c_4+c_5+c_6), \text{ for all } 1 \leq i < n \]
\[ T(n) = (c_n-c) + \sum_i^*d, \text{ c & d are constants} \]
\[ \sum_i^*d = 1*d+2*d+3*d+\ldots+(n-1)*d = d *(1+2+3+\ldots(n-1)) = d*n(n-1)/2 \]
\[ T(n) = (c_n-c) + dn^2/2-dn/2 = d*n^2+c_{11}*n+c_{12}, \text{ c's & d are consts} \]
Insertion Sort Average Case

- Average = (Best + Worst)/2
- $T(n) = cn^2 + dn + e$, $c, d, e$ are consts
Which case we consider?
Which case we consider?

- The worst case
Which case we consider?

- The worst case
  - Why?
Which case we consider?

- The worst case
  - Why?
    - It gives guarantees on the upper bound performance
Growth of Functions

- It is hard to compute the actual running time for more complex algorithms.
- The cost of the worst-case is a good measure.
- The growth of the cost function is what interests us (when input size is large).
- We are more concerned with comparing two cost functions, i.e., two algorithms.
Growth of Functions

\[ f(x) = 50x \]
\[ f(x) = x^3 \]
\[ f(x) = 2^x \]
O-notation

\[ f(n) = O(g(n)) \]

\[ \exists c > 0, n_0 > 0 \]
\[ 0 \leq f(n) \leq cg(n) \]
\[ n \geq n_0 \]

\( g(n) \) is an asymptotic upper-bound for \( f(n) \)
**Ω-notation**

\[ f(n) = \Omega(g(n)) \]

\[ \exists c > 0, n_0 > 0 \]

\[ 0 \leq cg(n) \leq f(n) \]

\[ n \geq n_0 \]

---

**g(n) is an asymptotic lower-bound for f(n)**
**Θ-notation**

\[ f(n) = \Theta(g(n)) \]

\[ \exists c_1, c_2 > 0, n_0 > 0 \]
\[ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \]
\[ n \geq n_0 \]

\( g(n) \) is an asymptotic tight-bound for \( f(n) \)
\( f(n) = o(g(n)) \)

\( \forall c > 0 \)
\( \exists n_0 > 0 \)
\( 0 \leq f(n) \leq cg(n) \)
\( n \geq n_0 \)

g(n) is a non-tight asymptotic upper-bound for f(n)
ω-notation

∀c > 0
∃n₀ > 0
0 ≤ cgn(n) ≤ f(n)
n ≥ n₀

f(n) = ω(g(n))

g(n) is a non-tight asymptotic lower-bound for f(n)
Comparing Two Functions

\[ \lim_{n \to \infty} \frac{f(n)}{g(n)} \]

- \(0:\) \quad f(n) = o(g(n))
- \(c > 0:\) \quad f(n) = \Theta(g(n))
- \(\infty:\) \quad f(n) = \omega(g(n))
## Analogy to Real Numbers

<table>
<thead>
<tr>
<th>Functions</th>
<th>Real numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f(n) = O(g(n)) )</td>
<td>( a \leq b )</td>
</tr>
<tr>
<td>( f(n) = \Omega(g(n)) )</td>
<td>( a \geq b )</td>
</tr>
<tr>
<td>( f(n) = \Theta(g(n)) )</td>
<td>( a = b )</td>
</tr>
<tr>
<td>( f(n) = o(g(n)) )</td>
<td>( a &lt; b )</td>
</tr>
<tr>
<td>( f(n) = \omega(g(n)) )</td>
<td>( a &gt; b )</td>
</tr>
</tbody>
</table>
Simple Rules

- We can omit constants
- We can omit lower order terms
- $\Theta(an^2+bn+c)$ becomes $\Theta(n^2)$, a, b, c are constants
- $\Theta(c1)$ and $\Theta(c2)$ become $\Theta(1)$, c’s are constants
- $\Theta(\log_{k1}n)$ and $\Theta(\log_{k2}n)$ become $\Theta(\log n)$, k’s are constants
- $\Theta(\log(n^k))$ becomes $\Theta(\log n)$, k is constant
Popular Classes of Functions

Constant: \( f(n) = \Theta(1) \)

Logarithmic: \( f(n) = \Theta(\log(n)) \)

Sublinear: \( f(n) = o(n) \)

Linear: \( f(n) = \Theta(n) \)

Super-linear: \( f(n) = \omega(n) \)

Quadratic: \( f(n) = \Theta(n^2) \)

Polynomial: \( f(n) = \Theta(n^k); \ k \text{ is a constant} \)

Exponential: \( f(n) = \Theta(k^n); \ k \text{ is a constant} \)
Insertion Sort Worst Case (Revisit)

Input array is reversed

**Insertion-Sort**\((A, n)\)

\[
\text{for } j = 2 \text{ to } n \\
\text{key} = A[j] \\
\text{// Insert } A[j] \text{ into the sorted sequence } A[1 \ldots j - 1]. \\
i = j - 1 \\
\text{while } i > 0 \text{ and } A[i] > \text{key} \\
i = i - 1 \\
A[i + 1] = \text{key}
\]

\[T(n) = (n-1)*n = O(n^2)\]
Comparing two algorithms

- $T_1(n) = 2n + 1000000$
- $T_2(n) = 200n + 1000$
- Which is better? Why?
  - In terms of order of growth?
Comparing two algorithms

- $T_1(n) = 2n + 1000000$
- $T_2(n) = 200n + 1000$
- Which is better? Why?
  - In terms of order of growth? Same
Comparing two algorithms

- $T_1(n) = 2n + 1000000$
- $T_2(n) = 200n + 1000$

Which is better? Why?

- In terms of order of growth? Same
- In terms of actual runtime?
Comparing two algorithms

- $T_1(n) = 2n + 1000000$
- $T_2(n) = 200n + 1000$

Which is better? Why?

- In terms of order of growth? Same
- In terms of actual runtime?
  - For $n \leq 5045$, $T_2$ is faster, otherwise $T_1$ is faster
Comparing two algorithms

- $T_1(n) = 2n + 1000000$
- $T_2(n) = 200n + 1000$

Which is better? Why?
- In terms of order of growth? Same
- In terms of actual runtime?
  - For $n \leq 5045$, $T_2$ is faster, otherwise $T_1$ is faster

What is the main usage of asymptotic notation analysis?
Analyzing Algorithms

Algorithm 1

for i = 1 to n
  j = 2*i
for j = 1 to n/2
  print j
Analyzing Algorithms

Algorithm 2

for i = 1 to n/2 {
    print i
    for j = 1 to n, step j = j*2
        print i*j
}


Analyzing Algorithms

Algorithm 3

for i = 1 to n/2
    print i
for j = 1 to n, step j = j*2
    print i*j
Analyzing Algorithms

- Algorithm 4

  input x (+ve integer)
  while x > 0
    print x
    \[ x = \lfloor x/5 \rfloor \]
Credits & Book Readings

Book Readings
- 2.1, 2.2, 3.1, 3.2

Credits
- Prof. Ahmed Eldawy notes
- Online websites
  - https://commons.wikimedia.org/wiki/File:Exponential.svg