Algorithm Analysis

Readings: Chapter 1.6-1.7.

How can we determine if we have an efficient algorithm?
Criteria:
- Does it meet specification/work correctly?
- Is it understandable/maintainable/simple?
- How much storage (memory & disk) does it use?
- How much time does it take to execute?

Example:
Fib(N) = Fib(N-1) + Fib(N-2), Fib(1) = 0, Fib(2) = 1

FibSolve(N) {
    If (N==1) return 0;
    if (N==2) return 1;
    return FibSolve(N-1)+FibSolve(N-2);
}

Fib(N) {
    int Result[N+1];
    Result[1] = 0;
    Result[2] = 1;
    for (int i = 3; i <= N; i++)
        Result[i] = Result[i-1] + Result[i+2];
    return Result[N];
}

Runtime ≈ 2

Runtime ≈ N
Performance Measurement

The amount of time a program takes to execute depends on:

- Algorithm
- Computer
- Compiler
- Input size
- Input organization

Approach

For representative inputs {
    Start timer
    Run algorithm (multiple times?)
    Stop timer
}

What about different computers, compilers, etc.?
Complexity Analysis

Time complexity of an algorithm:
amount of work done by an algorithm as a function of the size of the input

Count all operations ("operation" = semantically meaningful program segment
whose runtime independent of input or output size/complexity)

Operations:
  Integer addition
  for (i=1; i < 100; i++) { A[i]=0; }

Not Operations:
  for (i=1; i < INPUT_VAL; i++) { A[i]=0; }
  count(list);

Result: Compiler & Computer-independent metrics
Example - Sequential Search

```plaintext
for (i=1; i <= N; i++) { if (A[i] == search_element) return TRUE; }
return FALSE;
```

Assume Q=Probability element isn't in the list

**Worst-case:**

\[
\text{Average-case: } \quad Q \times (2^N + 1) + (1-Q) \times \frac{2^N}{2}
\]

Note: we typically focus primarily on worst-case.
Constants in Complexity

Question: How meaningful is the exact number of steps when individual steps can take different amounts of time?

Since duration of operations is variable, we ignore constant factors, and just consider orders of magnitude.

For large enough problems, constant factors irrelevant for comparisons

Example:

\[
f(n) = 3n^2 \quad g(n) = 25n \quad \rightarrow \quad g(n) < f(n) \quad \forall n > 8
\]

\[
f(n) = 3n^2 \quad g(n) = 300n \quad \rightarrow \quad g(n) < f(n) \quad \forall n > 100
\]
Asymptotic Complexity

"Big-Oh":
\[ C_1 \cdot f(n) = O(f(n)) \]
\[ f(n) + C_1 = O(f(n)) \]
ith degree polynomials \( C_i N^i + C_{i-1} N^{i-1} + \ldots + C_1 N + C_0 = O(N^i) \)
\[ \log(N) \] is greater than a constant, but less than \( N \)
\[ N + \log(N) \rightarrow O(N) \]
\[ N + N \log(N) \rightarrow O(N \log(N)) \]
\[ N^2 + N \log(N) \rightarrow O(N^2) \]

Note:
"Big-Oh" is normally worst-case.
Example Asymptotic Complexity: Bubble Sort

BubbleSort(array A, length N) {
    int j, k, temp;
    for (j = 1; j <= N; j++) {
        for (k = 1; k < N; k++) {
            if (A[k] > A[k+1]) {
                temp = A[k];
                A[k] = A[k+1];
                A[k+1] = temp;
            }
        }
    }
}

Worst-case: Reversed

$O(N^2)$
Example Asymptotic Complexity: Binary Search


```c
int Search(A) {
    if (A[0] == search_element) return 0;
    if (A[N] == search_element) return N;
    low = 0; high = N;
    while (low < high -1) {
        middle = (low+high)/2;
        if (A[middle] < search_element) low = middle;
        else if (A[middle] > search_element) high = middle;
        else return middle; /* Found the location */
    }
    return ERROR; /* Element should be in the list */
}
```

In each step, half of list is eliminated from consideration. N steps can handle a list of \( \sim 2^N \) length.

\[ \mathcal{O}(\log_2 N) \]
Example Asymptotic Complexity: Searching lists

Algorithm 1:
for (i=1; i <= N; i++) { if (A[i]==search_element) return TRUE; }
return FALSE; \( O(N) \)

Algorithm 2:
BubbleSort(array, N);
BinarySearch(array);
\( O(N^2) \) \( \Rightarrow \) \( O(N^2 + \log N) = O(N^2) \)

Algorithm 3:
UltraSort(array, N) //assume UltraSort is O(lgN)
BinarySearch(array);
\( O(lg^2 N) \) \( \Rightarrow \) \( O(lg^2 N + lg^2 N) \)
\( O(lg^2 N) \)
\( = O(2lg^2 N) \)
\( = O(lg^2 N) \)
### Practical Complexity

**Assume 1GHz machine (1 billion instructions per second)**

<table>
<thead>
<tr>
<th>n</th>
<th>log₂n</th>
<th>n</th>
<th>nlog₂n</th>
<th>n²</th>
<th>n³</th>
<th>n⁴</th>
<th>n¹₀</th>
<th>2ⁿ</th>
<th>n!</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.003 us</td>
<td>0.01 us</td>
<td>0.033 us</td>
<td>0.1 us</td>
<td>1. us</td>
<td>10. us</td>
<td>10. sec</td>
<td>1.02 us</td>
<td>3.63 ms</td>
</tr>
<tr>
<td>20</td>
<td>0.004 us</td>
<td>0.02 us</td>
<td>0.086 us</td>
<td>0.4 us</td>
<td>8. us</td>
<td>160. us</td>
<td>2.84 hr</td>
<td>1.05 ms</td>
<td>77.1 yr</td>
</tr>
<tr>
<td>30</td>
<td>0.005 us</td>
<td>0.03 us</td>
<td>0.15 us</td>
<td>0.9 us</td>
<td>27. us</td>
<td>810. us</td>
<td>6.83 day</td>
<td>1.07 sec</td>
<td>8.41E+15 yr</td>
</tr>
<tr>
<td>40</td>
<td>0.005 us</td>
<td>0.04 us</td>
<td>0.213 us</td>
<td>1.6 us</td>
<td>64. us</td>
<td>2.56 ms</td>
<td>121 day</td>
<td>18.3 min</td>
<td>2.59E+31 yr</td>
</tr>
<tr>
<td>50</td>
<td>0.006 us</td>
<td>0.05 us</td>
<td>0.282 us</td>
<td>2.5 us</td>
<td>125. us</td>
<td>6.25 ms</td>
<td>3.10 yr</td>
<td>13.0 day</td>
<td>9.64E+47 yr</td>
</tr>
<tr>
<td>100</td>
<td>0.007 us</td>
<td>0.1 us</td>
<td>0.664 us</td>
<td>10. us</td>
<td>1. ms</td>
<td>100. ms</td>
<td>3.17E+03 yr</td>
<td>4.02E+13 yr</td>
<td>2.96E+141 yr</td>
</tr>
<tr>
<td>1k</td>
<td>0.01 us</td>
<td>1. us</td>
<td>9.97 us</td>
<td>1. ms</td>
<td>1. sec</td>
<td>16.7 min</td>
<td>3.17E+13 yr</td>
<td>3.40E+284 yr</td>
<td></td>
</tr>
<tr>
<td>10k</td>
<td>0.013 us</td>
<td>10. us</td>
<td>133 us</td>
<td>100. ms</td>
<td>16.7 min</td>
<td>116 day</td>
<td>3.17E+23 yr</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100k</td>
<td>0.017 us</td>
<td>100. us</td>
<td>1.66 ms</td>
<td>10. sec</td>
<td>11.6 day</td>
<td>3170 yr</td>
<td>3.17E+33 yr</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

us = microsecond (10⁻⁶), ms = millisecond (10⁻³)

**Given twice as much time, how much can you do:**

- \( \mathcal{O}(\log n) \): \( n^2 \)
- \( \mathcal{O}(n) \): \( 2^n \)
- \( \mathcal{O}(n^2) \): \( n^{2^{1/2}} \)
- \( \mathcal{O}(n^3) \): \( n^{2^{1/3}} \)
- \( \mathcal{O}(2^n) \): \( n+1 \)