Day 2: Basic of Algorithms Analysis

"For me, great algorithms are the poetry of computation. Just like verse, they can be terse, allusive, dense, and even mysterious. But once unlocked, they cast a brilliant new light on some aspect of computing." - Francis Sullivan
As soon as an Analytic Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will arise - By what course of calculation can these results be arrived at by the machine in the shortest time? - Charles Babbage
Computational Tractability

Worst case running time. Obtain bound on largest possible running time of algorithm on input of a given size N, and see how this scales with N.
- Generally captures efficiency in practice.
- Draconian view, but hard to find effective alternative.

Desirable scaling property. When the input size increases by a factor of 2, the algorithm should only slow down by some constant factor C.

There exists constants $c > 0$ and $d > 0$ such that on every input of size $N$, its running time is bounded by $c N^d$ steps.

Def. An algorithm is efficient if it has polynomial running time.

Justification. It really works in practice!
# Why It Matters

<table>
<thead>
<tr>
<th>Run time in nanoseconds --&gt;</th>
<th>1.3 $N^3$</th>
<th>10 $N^2$</th>
<th>$47 N \log_2 N$</th>
<th>48 $N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time to solve a problem of size</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>1.3 seconds</td>
<td>10 msec</td>
<td>0.4 msec</td>
<td>0.048 msec</td>
</tr>
<tr>
<td>10,000</td>
<td>22 minutes</td>
<td>1 second</td>
<td>6 msec</td>
<td>0.48 msec</td>
</tr>
<tr>
<td>100,000</td>
<td>15 days</td>
<td>1.7 minutes</td>
<td>78 msec</td>
<td>4.8 msec</td>
</tr>
<tr>
<td>million</td>
<td>41 years</td>
<td>2.8 hours</td>
<td>0.94 seconds</td>
<td>48 msec</td>
</tr>
<tr>
<td>10 million</td>
<td>41 millennia</td>
<td>1.7 weeks</td>
<td>11 seconds</td>
<td>0.48 seconds</td>
</tr>
<tr>
<td>Max size problem solved in one</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>second</td>
<td>920</td>
<td>10,000</td>
<td>1 million</td>
<td>21 million</td>
</tr>
<tr>
<td>minute</td>
<td>3,600</td>
<td>77,000</td>
<td>49 million</td>
<td>1.3 billion</td>
</tr>
<tr>
<td>hour</td>
<td>14,000</td>
<td>600,000</td>
<td>2.4 trillion</td>
<td>76 trillion</td>
</tr>
<tr>
<td>day</td>
<td>41,000</td>
<td>2.9 million</td>
<td>50 trillion</td>
<td>1,800 trillion</td>
</tr>
<tr>
<td>N multiplied by 10, time multiplied by</td>
<td>1,000</td>
<td>100</td>
<td>10+</td>
<td>10</td>
</tr>
</tbody>
</table>

Reference: More Programming Pearls by Jon Bentley
# Orders of Magnitude

<table>
<thead>
<tr>
<th>Seconds</th>
<th>Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 second</td>
</tr>
<tr>
<td>10</td>
<td>10 seconds</td>
</tr>
<tr>
<td>$10^2$</td>
<td>1.7 minutes</td>
</tr>
<tr>
<td>$10^3$</td>
<td>17 minutes</td>
</tr>
<tr>
<td>$10^4$</td>
<td>2.8 hours</td>
</tr>
<tr>
<td>$10^5$</td>
<td>1.1 days</td>
</tr>
<tr>
<td>$10^6$</td>
<td>1.6 weeks</td>
</tr>
<tr>
<td>$10^7$</td>
<td>3.8 months</td>
</tr>
<tr>
<td>$10^8$</td>
<td>3.1 years</td>
</tr>
<tr>
<td>$10^9$</td>
<td>3.1 decades</td>
</tr>
<tr>
<td>$10^{10}$</td>
<td>3.1 centuries</td>
</tr>
<tr>
<td>...</td>
<td>forever</td>
</tr>
<tr>
<td>$10^{17}$</td>
<td>age of universe</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Meters Per Second</th>
<th>Imperial Units</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^{-10}$</td>
<td>1.2 in / decade</td>
<td>Continental drift</td>
</tr>
<tr>
<td>$10^{-8}$</td>
<td>1 ft / year</td>
<td>Hair growing</td>
</tr>
<tr>
<td>$10^{-6}$</td>
<td>3.4 in / day</td>
<td>Glacier</td>
</tr>
<tr>
<td>$10^{-4}$</td>
<td>1.2 ft / hour</td>
<td>Gastro-intestinal tract</td>
</tr>
<tr>
<td>$10^{-2}$</td>
<td>2 ft / minute</td>
<td>Ant</td>
</tr>
<tr>
<td>1</td>
<td>2.2 mi / hour</td>
<td>Human walk</td>
</tr>
<tr>
<td>$10^2$</td>
<td>220 mi / hour</td>
<td>Propeller airplane</td>
</tr>
<tr>
<td>$10^4$</td>
<td>370 mi / min</td>
<td>Space shuttle</td>
</tr>
<tr>
<td>$10^6$</td>
<td>620 mi / sec</td>
<td>Earth in galactic orbit</td>
</tr>
<tr>
<td>$10^8$</td>
<td>62,000 mi / sec</td>
<td>1/3 speed of light</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Powers of 2</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{10}$</td>
<td>thousand</td>
</tr>
<tr>
<td>$2^{20}$</td>
<td>million</td>
</tr>
<tr>
<td>$2^{30}$</td>
<td>billion</td>
</tr>
</tbody>
</table>

Reference: More Programming Pearls by Jon Bentley
Asymptotic Order of Growth

**Upper bounds.** \( T(n) \) is \( O(f(n)) \) if there exist constants \( c > 0 \) and \( n_0 \geq 0 \) such that for all \( n \geq n_0 \) we have \( T(n) \leq c \cdot f(n) \).

**Lower bounds.** \( T(n) \) is \( \Omega(f(n)) \) if there exist constants \( c > 0 \) and \( n_0 \geq 0 \) such that for all \( n \geq n_0 \) we have \( T(n) \geq c \cdot f(n) \).

**Tight bounds.** \( T(n) \) is \( \Theta(f(n)) \) if \( T(n) \) is both \( O(f(n)) \) and \( \Omega(f(n)) \).

**Ex:** \( T(n) = 32n^2 + 17n + 32 \).
- \( T(n) \) is \( O(n^2) \), \( O(n^3) \), \( \Omega(n^2) \), \( \Omega(n) \), and \( \Theta(n^2) \).
- \( T(n) \) is not \( O(n) \), \( \Omega(n^3) \), \( \Theta(n) \), or \( \Theta(n^3) \).

**Slight abuse of notation.** \( T(n) = O(f(n)) \).

**Vacuous statement.** Any comparison-based sorting algorithm requires at least \( O(n \log n) \) comparisons.
Properties

Transitivity. If $f = O(g)$ and $g = O(h)$ then $f = O(h)$.

Additivity. If $f = O(h)$ and $g = O(h)$ then $f + g = O(h)$. 
Asymptotic Bounds for Some Common Functions

**Polynomials.** $a_0 + a_1 n + \ldots + a_d n^d$ is $\Theta(n^d)$ if $a_d > 0$.

**Polynomial time.** Running time is $O(n^d)$ for some constant $d$ independent of the input size $n$.

**Logarithms.** $O(\log_a n) = O(\log_b n)$ for any constants $a, b > 0$.

(can avoid specifying the base, assuming it is a constant)

**Logarithms.** For every $x > 0$, $\log n = O(n^x)$.

(log grows slower than every polynomial)

**Exponentials.** For every $r > 1$ and every $d > 0$, $n^d = O(r^n)$.

(every exponential grows faster than every polynomial)
Linear Time: $O(n)$

**Linear time.** Running time is at most a constant factor times the size of the input.

**Computing the maximum.** Compute maximum of $n$ numbers $a_1$, ..., $a_n$.

```plaintext
max ← a_1
for i = 2 to n {
    if (a_i > max) {
        max ← a_i
    }
}
```
Linearithmic Time: $O(n \log n)$

Linearithmic time. Arises in divide-and-conquer algorithms.

Sorting. Mergesort and heapsort are sorting algorithms that perform $O(n \log n)$ comparisons.

Largest empty interval. Given $n$ time-stamps $x_1, \ldots, x_n$ on which copies of a file arrive at a server, what is largest interval of time when no copies of the file arrive?

$O(n \log n)$ solution. Sort the time-stamps. Scan the sorted list in order, identifying the maximum gap between successive time-stamps.
Quadratic Time: $O(n^2)$

**Quadratic time.** Enumerate all pairs of elements.

**Closest pair of points.** Given a list of $n$ points in the plane $(x_1, y_1), \ldots, (x_n, y_n)$, find the pair that is closest.

**$O(n^2)$ solution.** Try all pairs of points.

```plaintext
min ← (x_1 - x_2)^2 + (y_1 - y_2)^2
for i = 1 to n {
    for j = i+1 to n {
        d ← (x_i - x_j)^2 + (y_i - y_j)^2
        if (d < min)
            min ← d
    }
}
```

**Remark.** $\Omega(n^2)$ seems inevitable, but this is just an illusion.  

---  

Chapter 5
Cubic Time: $O(n^3)$

Cubic time. Enumerate all triples of elements.

Set disjointness. Given $n$ sets $S_1, \ldots, S_n$ each of which is a subset of $1, 2, \ldots, n$, is there some pair of these which are disjoint?

$O(n^3)$ solution. For each pairs of sets, determine if they are disjoint.

```plaintext
foreach set $S_i$ {
    foreach other set $S_j$ {
        foreach element $p$ of $S_i$ {
            determine whether $p$ also belongs to $S_j$
        }
        if (no element of $S_i$ belongs to $S_j$)
            report that $S_i$ and $S_j$ are disjoint
    }
}
```
Polynomial Time: $O(n^k)$ Time

**Independent set of size $k$.** Given a graph, are there $k$ nodes such that no two are joined by an edge?

**$O(n^k)$ solution.** Enumerate all subsets of $k$ nodes.

```
foreach subset S of k nodes {
    check whether S in an independent set
    if (S is an independent set)
        report S is an independent set
}
```

- Check whether $S$ is an independent set = $O(k^2)$.
- Number of $k$ element subsets = $\binom{n}{k} = \frac{n(n-1)(n-2)\cdots(n-k+1)}{k(k-1)(k-2)\cdots(2)(1)} \leq \frac{n^k}{k!}$ assuming $k$ is a constant
- $O(k^2 \frac{n^k}{k!}) = O(n^k)$.
Exponential Time

Independent set. Given a graph, what is maximum size of an independent set?

\[ O(n^2 2^n) \] solution. Enumerate all subsets.

\[
\begin{align*}
S^* & \leftarrow \emptyset \\
\text{foreach subset } S \text{ of nodes } \{ \\
\quad & \text{check whether } S \text{ in an independent set} \\
\text{if } & (S \text{ is largest independent set seen so far}) \\
\quad & \text{update } S^* \leftarrow S
\}
\end{align*}
\]
Two Key Types of Algorithms

Iterative Algorithms
Recursive Algorithms

Algorithms:
- **Precondition(s):** what is true when the algorithm starts
  - If it isn't true, the algorithm can do whatever it wants
  - Preconditions are statements about the input
- **Postconditions:** What is guaranteed to be true after the algorithm completes
  - Postconditions are statements about the output
Iterative Algorithms

Take one step at a time towards the final destination

loop (until done)
    take step
end loop
Loop Invariants

A good way to structure iterative algorithms

- Store the key information you currently know in some data structure
- In the loop
  - take a step towards destination
  - by making a simple change to the data
Maintain Loop Invariant

If algorithm is in a safe place, it doesn’t move to an unsafe location
- That is, loop invariant is always maintained

But, how do we know it starts in a safe place?
- We must establish the loop invariant based on the preconditions of the algorithm
Loop invariant is always true

That’s why it’s called invariant

How do we know it is always true
  ▪ It starts true
  ▪ Each time through the loop it stays true
  ▪ Induction tells us it is always true
Ending the Algorithm

Define an exit condition

Termination
- With sufficient progress, the exit condition will be met

When we exit
- We know the exit condition is true
- We know the loop invariant is true
- From these, we must establish the postcondition is true
Insertion Sort Example

Precondition: input is list of (possibly repeated) numbers
Postcondition: Output is list of same numbers in non-decreasing order
Define Loop Invariant

Some subset of the elements are sorted
Remaining elements are off to the side
Loop

Choose an element from unsorted elements
Insert it where it belongs elements

before loop

after loop
Making progress

Progress

- Amount of the input consumed

7 to go

6 to go
Beginning and Ending

n to go

14 23 25 30 31 52 62 79 88 98

0 to go
Pseudocode

InsertionSort(B)
Precondition: B is bag of elements
Postcondition: returns L, list of each element from B in non-decreasing order

L = empty list
R = B
loop
    loop-invariant: L is a non-decreasing list of elements where R U L = B
    exit when R is empty
    Remove an element e from R
    Insert e into the appropriate location in L
end loop
return L
Loop Invariant Example

Initialization
- Since $L$ is empty, $L$ is non-decreasing list of elements
- Since $R=B$, $R \cup L = B$

Maintenance
- $L''$ is $L'$ with an element added to it in the appropriate location. Therefore, $L''$ is a non-decreasing list of elements
- $B = R' \cup L' = (R' - e) \cup (L' \cup e) = R'' \cup L''$

Termination
- $R$ is empty (by exit condition)
- $R \cup L = B$ (by LI)
- $\emptyset \cup L = B \rightarrow L = B$
- $L$ is a non-decreasing list of elements (by LI)
For Tuesday (Day 3)

Read Chapter 4 of Kleinberg