Basics of Algorithms Analysis

2.1 Computational Tractability

Lecture 3

Course instructor:
Sikder Huq
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
Brute force. For many non-trivial problems, there is a natural brute force search algorithm that checks every possible solution.
- Typically takes $2^N$ time or worse for inputs of size $N$.
- Unacceptable in practice.

Desirable scaling property. When the input size doubles, 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 poly-time if the above scaling property holds.
Worst-Case Analysis

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

Average case running time. Obtain bound on running time of algorithm on random input as a function of input size $N$.
- Hard (or impossible) to accurately model real instances by random distributions.
- Algorithm tuned for a certain distribution may perform poorly on other inputs.
Worst-Case Polynomial-Time

Def. An algorithm is efficient if its running time is polynomial.

Justification: It really works in practice!
- Although $6.02 \times 10^{23} \times N^{20}$ is technically poly-time, it would be useless in practice.
- In practice, the poly-time algorithms that people develop almost always have low constants and low exponents.

Exceptions.
- Some poly-time algorithms do have high constants and/or exponents, and are useless in practice.
- Some exponential-time (or worse) algorithms are widely used because the worst-case instances seem to be rare.
Why It Matters

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds $10^{25}$ years, we simply record the algorithm as taking a very long time.

<table>
<thead>
<tr>
<th></th>
<th>$n$</th>
<th>$n \log_2 n$</th>
<th>$n^2$</th>
<th>$n^3$</th>
<th>$1.5^n$</th>
<th>$2^n$</th>
<th>$n!$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n = 10$</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>4 sec</td>
</tr>
<tr>
<td>$n = 30$</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>18 min</td>
<td>$10^{25}$ years</td>
</tr>
<tr>
<td>$n = 50$</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>11 min</td>
<td>36 years</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 100$</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>1 sec</td>
<td>12,892 years</td>
<td>$10^{17}$ years</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 1,000$</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>1 sec</td>
<td>18 min</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 10,000$</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>2 min</td>
<td>12 days</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 100,000$</td>
<td>&lt; 1 sec</td>
<td>2 sec</td>
<td>3 hours</td>
<td>32 years</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 1,000,000$</td>
<td>1 sec</td>
<td>20 sec</td>
<td>12 days</td>
<td>31,710 years</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
</tbody>
</table>
Why It Matters

The graph shows various functions and their growth rates as a function of $n$. The functions are:

- $f(n) = 2^n$
- $f(n) = n^3$
- $f(n) = n^2$
- $f(n) = n^2 \log n$
- $f(n) = 100 \log n$
- $f(n) = 500$
- $f(n) = 20n$

The graph illustrates how different functions grow at different rates as $n$ increases.
2.2 Asymptotic Order of Growth
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 \( \Theta(n^3) \).

Meaningless statement. Any comparison-based sorting algorithm requires at least \( O(n \log n) \) comparisons.
   - Statement doesn't "type-check."
   - Use \( \Omega \) for lower bounds.
Properties

Transitivity.
- If \( f = O(g) \) and \( g = O(h) \) then \( f = O(h) \).
- If \( f = \Omega(g) \) and \( g = \Omega(h) \) then \( f = \Omega(h) \).
- If \( f = \Theta(g) \) and \( g = \Theta(h) \) then \( f = \Theta(h) \).

Additivity.
- If \( f = O(h) \) and \( g = O(h) \) then \( f + g = O(h) \).
- If \( f = \Omega(h) \) and \( g = \Omega(h) \) then \( f + g = \Omega(h) \).
- If \( f = \Theta(h) \) and \( g = \Theta(h) \) then \( f + g = \Theta(h) \).
Q1: Understanding big-Oh notation

Suppose you have functions $f$ and $g$ such that $f(n)$ is $O(g(n))$. Is it the case that:

(a) $\log_2 f(n)$ is $O(\log_2 g(n))$ ?
(b) $2^f(n)$ is $O(2^g(n))$ ?

Answers:

(a) This is false in general, since it could be that $g(n) = 1$ for all $n$, $f(n) = 2$ for all $n$, and then $\log_2 g(n) = 0$, whence we cannot write $\log_2 f(n) \leq c \log_2 g(n)$.

On the other hand, if we simply require $g(n) \geq 2$ for all $n$ beyond some $n_1$, then the statement holds. Since $f(n) \leq c g(n)$ for all $n \geq n_0$, we have $\log_2 f(n) \leq \log_2 g(n) + \log_2 c \leq (\log_2 c)(\log_2 g(n))$ once $n \geq \max(n_0, n_1)$.

(b) This is false: take $f(n) = 2n$ and $g(n) = n$. Then $2^f(n) = 4^n$, while $2^g(n) = 2^n$. 

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 \( a, b > 0 \).

\[
\uparrow \\
\text{can avoid specifying the base}
\]

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

\[
\uparrow \\
\text{log grows slower than every polynomial}
\]

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

\[
\uparrow \\
\text{every exponential grows faster than every polynomial}
\]
Q2: Sort in ascending order

\[
\begin{align*}
10^n & \quad (\log n)^5 \quad (\log n) \quad 2^{2n} \\
100 & \quad n^{1/3} \quad n^n \quad 2\sqrt{\log_2 n} \quad (\log n)^{10^{100}} \\
9 \log n + 5(\log n)^3 + 2n^2 & \quad n!
\end{align*}
\]

- One proof strategy: use \( \lim_{n \to \infty} \frac{f(n)}{g(n)} \)

Textbook claim (2.1)
2.4 A Survey of Common Running Times
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 $a_1, \ldots, a_n$.

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

**Merge.** Combine two sorted lists $A = a_1, a_2, \ldots, a_n$ with $B = b_1, b_2, \ldots, b_n$ into sorted whole.

Claim. Merging two lists of size $n$ takes $O(n)$ time.

**Pf.** After each comparison, the length of output list increases by 1.
O(n log n) Time

O(n log n) time. Arises in divide-and-conquer algorithms.

also referred to as linearithmic time

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 copies of a file arriving at a server, what is the 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.

```
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.  \( O(n \log n) \) time possible
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?

$k$ is a constant

$O(n^k)$ 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 = $\binom{n}{k} = \frac{n(n-1)(n-2)\cdots(n-k+1)}{k(k-1)(k-2)\cdots(2)(1)} \leq \frac{n^k}{k!}$
- $O(k^2 \ n^k / k!) = O(n^k)$.

poly-time for $k=17$, but not practical
Exponential Time

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

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

```
S* ← φ
foreach subset S of nodes {
    check whether S in an independent set
    if (S is largest independent set seen so far)
        update S* ← S
}
```
**Sublinear Time**

**Binary search.** Given a sorted array $A$, check if a given number $p$ belongs to the array.

**$O(\log n)$ solution.** Probe particular entries in the array.

```python
Bsearch(A, p, lo, hi):
    if (hi < lo) return false;
    mid = (lo + hi)/2;
    if (A[mid] > p) return Bsearch(A, p, lo, mid - 1);
    else if (A[mid] < p) return Bsearch(A, p, mid + 1, hi);
    else return true;
```

**Caveat.** It takes $\Omega(n)$ time just to read the array.
So, only applicable in models where the input is “queried” rather than read directly.
Q3: Analyzing an algorithm

You have an array $A$ with integer entries $A[1], \ldots, A[n]$.

Here is an algorithm:

For $i=1, 2, \ldots, n$
    For $j = i+1, 2, \ldots, n$ {
        Add up entries $A[i]$ through $A[j]$
        Store result in $B[i,j]$
    }

Obtain an upper bound and a lower bound for the algorithm.
The lower bound is the more interesting one

Consider the times during the execution of the algorithm when \( i \leq n/4 \) and \( j \geq 3n/4 \).

In these cases, \( j - i + 1 \geq 3n/4 - n/4 + 1 > n/2 \). Therefore, adding up the array entries \( A[i] \) through \( A[j] \) would require at least \( n/2 \) operations, since there are more than \( n/2 \) terms to add up.

How many times during the execution of the given algorithm do we encounter such cases? There are \( (n/4)^2 \) pairs \((i, j)\) with \( i \leq n/4 \) and \( j \geq 3n/4 \). The given algorithm enumerates over all of them, and as shown above, it must perform at least \( n/2 \) operations for each such pair. Therefore, the algorithm must perform at least \( n/2 \cdot (n/4)^2 = n^3/32 \) operations. This is \( \Omega(n^3) \), as desired.