Algorithm runtime analysis and computational tractability

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

Time Complexity of an Algorithm

How do we measure the complexity (time, space requirements) of an algorithm.

The size of the problem: an integer n
- # inputs (for sorting problem)
- #digits of input (for the primality problem)
- sometimes more than one integer

Want to characterize the running time of an algorithm for increasing problem sizes by a function T(n)

Units of time

1 microsecond?
1 machine instruction?
# of code fragments that take constant time?

Units of time

1 microsecond?
no, too specific and machine dependent
1 machine instruction?
no, still too specific
# of code fragments that take constant time?
yes

unit of space

bit?
integer?

unit of space

bit?
very detailed but sometimes necessary
integer?
nicer, but dangerous: we can code a whole program or array (or disk) in one arbitrary integer, so we have to be careful with space analysis. Better to think in terms of machine words (ie fixed size collections of bits)
Worst-Case Analysis

Worst case running time.

A bound on largest possible running time of algorithm on inputs of size $n$.

- Generally captures efficiency in practice, but can be an overestimate.

Same for worst case space complexity

Average case

Average case running time. A bound on the average running time of algorithm on random inputs as a function of input size $n$. In other words: the expected number of steps an algorithm takes.

$$T(n) = \sum_{i \in I} t(i) \cdot p(i)$$

- $p(i)$: probability of input $i$
- $t(i)$: time complexity of input $i$
- $I$: all possible inputs of size $n$

- Hard to model real instances by random distributions.
- Algorithm tuned for a certain distribution may perform poorly on other inputs.
- Often hard to compute.

Why It Matters

A definition of tractability: Polynomial-Time

Brute force. For many 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 $n$.
- Unsatisfactory in practice.

Some poly-time algorithms do have high constants and/or exponents, and are useless in practice. Example: Coppersmith-Winograd algorithm: $O(n^{2.376})$.

Some exponential-time (or worse) algorithms are widely used because the worst-case instances seem to be rare. Example: simplex method for linear programming.

Comparing algorithm running times

Suppose that algorithm $A$ has a running time bounded by:

$$T(n) = 1.62 n^2 + 3.5 n + 8$$

- It is hard to get this kind of exact statement.
- More detail than is useful.
- Want to quantify running time in a way that will allow us to identify broad classes of algorithms.
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 $T(n) \leq c \cdot f(n)$.

Example: $T(n) = 32n^2 + 16n + 32$.
- $T(n)$ is $O(n^2)$, $O(n^3)$.

There are many possible upper bounds!

Expressing Lower Bounds

Big O doesn't always express what we want:

- 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.
- $T(n)$ 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 $T(n) \geq c \cdot f(n)$.

Example: $T(n) = 32n^2 + 16n + 32$.
- $T(n)$ is $\Omega(n^2)$, $\Omega(n)$.

Tight Bounds

$T(n)$ is $\Theta(f(n))$ if $T(n)$ is both $O(f(n))$ and $\Omega(f(n))$.

Example: $T(n) = 32n^2 + 17n + 32$.
- $T(n)$ is $\Theta(n^2)$.

If we show that the running time of an algorithm is $\Theta(f(n))$, we have found the "right" bound.
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 \( g = O(h) \) then \( f + g = \Theta(h) \).

Asymptotic Bounds for Some Common Functions

Polynomials: \( a_0 + a_1 n + \ldots + a_d n^d \) is \( O(n^d) \) if \( a_d > 0 \).

Polynomial time. Running time is \( O(n^d) \) for some constant \( d \).

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

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

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

Every exponential grows faster than every polynomial.

Running time of the GS algorithm

Initialize each person to be free.
while (some man is free and hasn’t proposed to every woman)
Choose such a man \( m \) and highest-ranked woman on \( m \)'s list to whom \( m \) has not yet proposed
if \( w \) is free)
\( (m, w) \) become engaged
else if \( w \) prefers \( m \) to her fiancé \( m' \)
\( (m, w) \) become engaged, \( m' \) becomes free
else
\( m \) remains free

We have shown that the algorithm takes at most \( n^2 \) iterations. Is the algorithm \( O(n^2) \)?

A Survey of Common Running Times

lower bounds on solving a problem

For a problem to have a lower bound \( \Omega(f(n)) \) means:
- Any algorithm solving this problem takes at least \( \Omega(f(n)) \) steps.
We can often show that an algorithm has to "touch" all elements of a data structure, or produce a certain sized output. This then gives rise to an easy lower bound.

Sometimes we can prove better (higher) lower bounds (eg Sorting).

Closed / open problems

Problems have lower bounds, algorithms have upper bounds. A closed problem has a lower bound \( \Omega(f(n)) \) and at least one algorithm with upper bound \( O(f(n)) \).

Example: sorting is \( \Omega(n \log n) \) and there are \( O(n \log n) \) sorting algorithms.

To show this, we need to reason about lower bounds of problems (cs420).
An open problem has lower bound + upper bound
- Example: matrix multiplication (multiply two \( n \times n \) matrices).
  - Takes \( \Omega(n^3) \) why?
  - Naïve algorithm: \( O(n^3) \)
  - Coppersmith-Winograd algorithm: \( O(n^{2.376}) \)
Constant time: $O(1)$

A single line of code that involves "simple" expressions, e.g.:
- Arithmetical operations (+, -, *, /) for fixed size inputs
- Assignments ($x = \text{simple expression}$)
- Conditionals with simple sub-expressions
- Function calls (excluding the time spent in the called function)

Logarithmic time

Example of a problem with $O(\log(n))$ bound: binary search

How did we get that bound?

Logarithms and their properties

definition:
- $b\log_a(x) = \log_b(x) \Rightarrow \log_b(a) = \frac{1}{\log_a(b)}$
- $\log(xy) = \log(x) + \log(y)$
- $\log\left(\frac{x}{y}\right) = \log(x) - \log(y)$
- $\log(x^y) = y\log(x)$

Linear Time: $O(n)$

Linear time. Running time is proportional to the size of the input.

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

```plaintext
maximum \leftarrow a_1
for i = 2 to n {
    if $(a_i > \text{maximum})$
        maximum \leftarrow a_i
}
```

Logarithms and algorithms

When in each step of an algorithm we halve the size of the problem then it takes $\log_2(n)$ steps to get to the base case.

We often use $\log(n)$ when we should use $\lfloor \log(n) \rfloor$. That’s OK since $\lfloor \log(n) \rfloor = \Theta(\log(n))$.

Similarly, if we divide a problem into $k$ parts the number of steps is $\log_k(n)$. For the purposes of big-$O$ analysis it doesn’t matter since $\log_a(n)$ is $O(\log_b(n))$.

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 a single sorted list.

Claim. Merging two lists of size $n$ takes $O(n)$ time.
### Linear Time: $O(n)$

**Polynomial evaluation.** Given

$$A(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0 \quad (a_n \neq 0)$$

Evaluate $A(x)$

**How not to do it:**

$$a_n \exp(x,n) + a_{n-1} \exp(x,n-1) + \ldots + a_1 \exp(x,1) + a_0$$

$$y = 0$$

for $i$ in range(n + 1):

$$y += a[i] \exp(x, i)$$

**Why not?**

### Polynomial evaluation using Horner: complexity

**Lower bound:** $\Omega(n)$ because we need to access each $a[i]$ at least once

**Upper bound:** $O(n)$

Closed problem!

But what if $A(x) = x^n$?

### A glass-dropping experiment

You are testing a model of glass jars, and want to know from what height you can drop a jar without its breaking. You can drop the jar from heights of 1, 2, ..., $n$ foot heights.

- **If you have a single jar:** do linear search ($O(n)$ work).
- **If you have an unlimited number of jars:** do binary search ($O(\log n)$ work).
- **If you have an unlimited number of jars:** do binary search ($O(\log n)$ work)

Can you design a strategy for the case you have 2 jars, resulting in a bound that is strictly less than $O(n)$? 

---

**Recurrence:**

$$x^{2n} = x^n \cdot x^n$$

$$x^{2n} = x^n \cdot x^{2n}$$

```python
def pwr(x, n):
    if (n==0) : return 1
    if odd(n) : return x * pwr(x, n-1)
    else :
        a = pwr(x, n/2)
        return a * a
```

**Complexity?**

**A glass-dropping experiment**

You are testing a model of glass jars, and want to know from what height you can drop a jar without its breaking. You can drop the jar from heights of 1, 2, ..., $n$ foot heights.

- **If you have a single jar:** do linear search ($O(n)$ work).
- **If you have an unlimited number of jars:** do binary search ($O(\log n)$ work).
- **If you have an unlimited number of jars:** do binary search ($O(\log n)$ work)

Can you design a strategy for the case you have 2 jars, resulting in a bound that is strictly less than $O(n)$?

---

**How to do it: Horner’s rule**

$$a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0 =$$

$$\left(a_n x^{n-1} + a_{n-1} x^{n-2} + \ldots + a_1 \right) x + a_0 =$$

$$\left(\ldots \left(a_n x + a_{n-1}\right) x + a_{n-2}\right) x + \ldots + a_1 \right) x + a_0$$

$$y[a[n]]$$

for $i$ in range(n-1, -1, -1) :

$$y = y \times x + a[i]$$

---

**O($n \log n$) Time**

Often arises in divide-and-conquer algorithms like mergesort.

```python
mergesort(A) :
    if len(A) <= 1 return A
    else return merge(mergesort(left half(A)), mergesort(right half(A)))
```
Merge Sort - Divide

{7,3,2,9,1,6,4,5}

{7,3,2,9}

{1,6,4,5}

{7,3}

{2,9}

{1,6}

{4,5}

Merge Sort - Merge

{7,3,2,9,1,6,4,5}

{1,6,4,5}

{7,3,2,9}

{1,6,4,5}

{7,3}

{2,9}

{1,6}

{4,5}

{1,2,3,4,5,6,7,9}

{2,3,7,9}

{1,4,5,6}

{3,7}

{2,9}

{1,6}

{4,5}

O(n log n)

mergesort(A) :

if len(A) <= 1 return A
else return merge(mergesort(left half(A)), mergesort(right half(A)))

At depth i
- work done
- split
- merge
- total work?
Total depth?
Total work?

Quadratic Time: O(n^2)

Examples:

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), …, (x_n, y_n), find the pair that is closest.

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

\[
\text{min} \leftarrow (x_1 - x_2)^2 + (y_1 - y_2)^2
\]

for i = 1 to n - 1 {
    for j = i+1 to n {
        \[d \leftarrow (x_i - x_j)^2 + (y_i - y_j)^2\]
        if \(d < \text{min}\) {
            \[\text{min} \leftarrow d\]
        }
    }
}

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

Largest interval sum

Given an array A[0],…,A[n-1], find indices i,j such that the sum \(A[i] + \ldots + A[j]\) is maximized.

Naïve algorithm :

\[
\text{max_sum} = -\text{infinity}
\]

for i in range(n - 1) :
    for j in range(i + 1, n) :
        \[\text{current_sum} = A[i] + \ldots + A[j]\]
        if \(\text{current_sum} > \text{max_sum}\)
            \[\text{max_sum} \leftarrow \text{current_sum}\]

Example:

A = [2, -3, 4, 2, 5, 7, -10, -8, 12]

big O bound?
Can we do better?
Exponential Time

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

solution: Enumerate all subsets.

\[
S^* \leftarrow \phi
\]

foreach subset \( S \) of nodes {
check whether \( S \) is an independent set
if \( S \) is largest independent set seen so far
update \( S^* \leftarrow S \)
}

big \( O \) bound?

Exponential Time

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

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

\[
S^* \leftarrow \phi
\]

foreach subset \( S \) of nodes {
check whether \( S \) is an independent set
if \( S \) is largest independent set seen so far
update \( S^* \leftarrow S \)
}

No polynomial time solution to the problem is known. But it hasn’t been proven that exponential time is required.

Polynomial, NP, Exponential

Some problems (such as searching a sorted array) have a polynomial solution: an \( O(n) \) time algorithm solving them. \((p \text{ constant})\)

Some problems (such as Hanoi) take an exponential time to solve: \( O(p^n) \) \((p \text{ constant})\)

For some problems (independent set) we only have an exponential solution, but we don’t know if there exists a polynomial solution.

Some NP problems

TSP: Travelling Salesman
given cities \( c_1, c_2, \ldots, c_n \) and distances between all of these, find a minimal distance tour connecting all cities.

SAT: Satisfiability
given a Boolean expression \( E \) with variables \( x_1, x_2, \ldots, x_n \) determine a truth assignment to all \( x \), making \( E \) true

Coping with intractability

NP problems become intractable quickly
Good to know you are dealing with such a problem!

Coping with intractability:
- Approximation: Find a nearly optimal solution
- Randomization: Use a probabilistic algorithm using “coin tosses”