Run-Time Analysis: Some review, some new stuff.

Want to express running time of a program/algorithm.

Big-O:

For an algorithm A with input size n:

1. Express runtime of A as a function of n:
   \[ T(n) = \text{runtime of } A \] (might be a complicated function)
   \[ f(n) = \text{some "simple" function of } n. \]

2. Establish relative order \((\leq, =, \geq)\) \((0 \to \infty)\) between growth rates of \(T(n)\) and \(f(n)\).

---

\(1\) How to compare 2 functions? "Can't just say one is greater than the other."

Ex: \(T(n) = 1000n\) \(f(n) = n^2\)

\(T(n) > f(n)\) for \(n < 1000\)

\(T(n) < f(n)\) for \(n > 1000\)

\(\downarrow \downarrow\)

\(2\) So compare in terms of relative growth rate - growth rate - how fast a function grows asymptotically (i.e. as \(n \to \infty\))

\(\leq \leq \leq \leq\)

Fastest growing
*Typical growth rates*

**Runtime Notations**

1. \( T(n) = O(f(n)) \) \( T(n) \) grows at a rate \( \leq f(n) \)
   - \( f(n) \) is an upper bound on \( T(n) \).

   Ex: \( T(n) = O(n^2) \) \( \Rightarrow \) runtime is at most quadratic (won’t be worse)

2. \( T(n) = \Omega(f(n)) \) \( T(n) \) grows at a rate \( \geq f(n) \)
   - \( f(n) \) is a lower bound on \( T(n) \)

   Ex: \( T(n) = \Omega(\log(n)) \) runtime is at least logarithmic (can’t be better)

3. \( T(n) = \Theta(f(n)) \) \( T(n) \) grows at a rate \( = f(n) \)
   - \( f(n) \) is both upper and lower bound on \( T(n) \)

   Ex: \( T(n) = \Theta(n^2) \) \( \Rightarrow \) runtime is "exactly" quadratic

What’s the difference?

Ex: \( T(n) = 3n \)

- \( f(n) = n^2 \) \( T(n) = O(n^2) \) \( T(n) = \Omega(n^2) \) \( T(n) = \Theta(n^2) \)

- \( f(n) = n \)
  \( T(n) = O(n) \) \( T(n) = \Omega(n) \) \( T(n) = \Theta(n) \)

- \( f(n) = 1 \)
  \( T(n) = O(1) \) \( T(n) = \Omega(1) \) \( T(n) = \Theta(1) \)
Typical growth rates

- $c$ constant
- $\log(n)$ logarithmic
- $n$ linear
- $n \log n$ "efficient"
- $n^2$ quadratic
- $n^3$ cubic
- $n^k$ polynomial
- $2^n$ exponential

→ Go to run-time notations

"In this class, we'll typically use big-Oh but will express in tightest bound."

Tight bound - $f(n)$ expressed in lowest correct rate.

ex: $T(n) = 1 + 100n^2$ - Which is the tight bound?

- $T(n) = O(n)$ - not correct
- $T(n) = O(n^3)$ - correct but not tight
- $T(n) = O(n^3)$ - tight (so correct too)

In CS201 analyzed runtimes of many algorithms but approach is different for recursive algorithms

Today: analyze recursive Merge-Sort
Analyzing runtime of recursive algorithms
ex: Merge Sort

Quick Review
"Continuously split list in 1/2 until lists of size 1, merge these sublists together"

MergeSort (A) // A is list

// sublist has > 2 items
if (|A| > 1)
  leftA = getLeftHalf (A)
  rightA = getRightHalf (A)
  MergeSort (leftA)
  MergeSort (rightA)
  Merge (leftA, rightA)

Time? Θ(1) = c

"MergeSort Visual" (on slides)

Divide-and-Conquer (and -Combine)
- Divide: break problem into smaller subproblems
- Conquer: recursively solve each subproblem
- Combine: combine sub solutions

"For Merge Sort?"
recurrence - describes runtime of a recursive call in terms of input size n.

\[ T(n) = \text{runtime of Mergesort} \]

"Write as a recurrence. Since the function has 2 parts: base case + recursive, the recurrence has 2 parts"

\[ T(n) = \begin{cases} 
  c & \text{if } n = 1 \\
  2T(n/2) + cn + c & \text{if } n > 1 
\end{cases} \]

"What if we can write conquer time in terms of \( T(n) \)?"

"\( T(n) \) doesn't tell us much about runtime since it's in terms of \( T(n) \). Want it in terms of \( n \) only. To do so, we will solve the recurrence."

To solve a recurrence: (3 methods)

1) Recurrence Tree (visual) - today
2) Master's Theorem
3) Substitution Method

Recurrence Tree
Assume \( n \) is a power of 2: \( n = 2^k \).
Already saw recurrence tree for Change-Making. Change(n) — non-recursive part took \(O(k)\) + we made 3 recursive calls:

\[
\text{Change}(n-12) \quad \text{Change}(n-5) \quad \text{Change}(n-1)
\]

For Merge Sort:

\[
\text{MS}(n) \quad \text{MS}(n/2) \quad \text{MS}(n/4)
\]

Now, each call takes \(O(n) = cn\) time.

Each node represents the time for some call. Sum up the nodes to get total runtime.

Eventually? Last level of the tree?

All 0's: we recurse down to lists of size 1.

Time: \(cn\)
Let's find time at each level and multiply by number of levels.

Time at each level = $cn$

# levels = $O(\log n)$. (start from $n$, halve until 1)

Total time = $O(n \log n)$.

Note: Actually $\Theta(n \log n)$ not just $O(n \log n)$.

(i.e. not over/under-estimating anywhere).