11-711 Algorithms for NLP

Introduction to Analysis of Algorithms

Reading:
Cormen, Leiserson, and Rivest,
Introduction to Algorithms
Chapters 1, 2, 3.1., 3.2.

Analysis of Algorithms

- **Analyzing** an algorithm: predicting the computational resources that the algorithm requires
  - Most often interested in *runtime* of the algorithm
  - Also memory, communication bandwidth, etc.
- Computational model assumed: RAM
  - Instructions are executed sequentially
  - Each instruction takes constant time (different constants)
- The behavior of an algorithm may be different for different inputs. We want to summarize the behavior in general, easy-to-understand formulas, such as worst-case, average-case, etc.

Analysis of Algorithms

- Generally, we are interested in computation time as a function of *input size*.
  - Different measures of input size depending on the problem
  - Examples:

Problem	Input	Size
Sorting	List of numbers to sort	# of numbers to sort
Multiplication	Numbers to multiply	# of bits
Graph	Set of nodes and edges	# of nodes + # of edges

- **"Running time"** of an algorithm: number of primitive "steps" executed
  - Think of each "step" as a constant number of instructions, each of which takes constant time.
- Example: Insertion-Sort (CLR, p. 8)

Insertion Sort *(A)*

		cost	times
1.	**for** $j \leftarrow 2$ **to** length[A]	c_1	n
2.	**do** $key \leftarrow A[j]$	c_2	n-1
3.			
4.	$i \leftarrow j$ - 1	c_4	n-1
5.	**while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n}(t_j)$
6.	**do** $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
7.	$i \leftarrow i$-1	c_7	$\sum_{j=2}^{n}(t_j - 1)$
8.	$A[i+1] \leftarrow key$	c_8	n-1

Insertion-Sort

t_j: # of times the while loop in line 5 is executed for the value j

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

- In the best case, when the array is already sorted, the running time is a linear function of n.

$$T(n) = an + b$$

- In the worst case, when the array is opposite of sorted order, the running time is a quadratic function of n.

$$T(n) = an^2 + bn + c$$

Analysis of Algorithms

- We are mostly interested in *worst-case* performance because
  1. It is an upper bound on the running time for any input.
  2. For some algorithms it occurs fairly often.
  3. The average case is often roughly as bad as the worst case.
- We are interested in the rate of growth of the runtime as the input size increases. Thus, we only consider the *leading term*.
  – Example: Insertion-Sort has worst case $O(n^2)$
- We will normally look for algorithms with the lowest order of growth
- But not always:
  $\rightarrow$ for instance, a less efficient algorithm may be preferable if it's an *anytime* algorithm.

Analyzing Average Case of Expected Runtime

- **Non-experimental method:**
  1. Assume all inputs of length n are equally likely
  2. Analyze runtime for each particular input and calculate the average over all the inputs
- **Experimental method:**
  1. Select a sample of test cases (large enough!)
  2. Analyze runtime for each test case and calculate the average and standard deviation over all test cases

Recursive Algorithms (Divide and Conquer)

- General structure of recursive algorithms:
  1. *Divide* the problem into smaller subproblems.
  2. *Solve (Conquer)* each subproblem recursively.
  3. *Merge* the solutions of the subproblems into a solution for the full problem.
- Analyzing a recursive algorithm usually involves solving a **recurrence equation**.
- Example: Merge-Sort (CLR, p. 13)

Recursive Algorithms

- $D(n) =$ time to divide a problem of size n
- $C(n) =$ time to combine solutions of size n
- Assume we divide a problem into a subproblems, each $\frac{1}{b}$ the size of the original problem.
- The resulting recurrence equation is:

$$T(n) = \begin{cases} O(1) & if\ n \leq c \\ aT(\frac{n}{b}) + D(n) + C(n) & otherwise \end{cases}$$

Merge (A, p, q, r)**: Initialization**

	times
1. $n_1 \leftarrow q - p + 1$	1
2. $n_2 \leftarrow r - q$	1
3. create arrays $L[1..n_1 + 1]$ and $R[1..n_2 + 1]$	1
4. **for** $i \leftarrow i$ **to** n_1	n_1
5. **do** $L[i] \leftarrow A[p + i - 1]$	n_1
6. **for** $j \leftarrow i$ **to** n_2	$n - n_1$
7. **do** $R[j] \leftarrow A[q + j]$	$n - n_1$
8. $L[n_1 + 1] \leftarrow \infty$	1
9. $R[n_2 + 1] \leftarrow \infty$	1
10. $i \leftarrow 1$	1
11. $j \leftarrow 1$	1

Merge (A, p, q, r)**(cont.): Actual Merge**

	times
for $k \leftarrow p$ **to** r	n
do if $L[i] \leq R[j]$	n
then $A[k] \leftarrow L[i]$	n/b
$i \leftarrow i + 1$	n/b
else $A[k] \leftarrow R[j]$	$n/(n-b)$
$j \leftarrow j + 1$	$n/(n-b)$

Merge Sort (A,p,r)

		times
1.	**if** $p < r$	
2.	**then** $q \leftarrow \lfloor (p+r)/2 \rfloor$	
3.	Merge-Sort(A,p,q)	$T(n/2)$
4.	Merge-Sort($A,q+1,r$)	$T(n/2)$
5.	Merge(A,p,q,r)	$O(n)$

A note on Implementation

- Note that in the pseudocode of the Merge operation, the for-loop is *always* executed n times, no matter how many items are in each of the subarrays.
- This can be avoided in practical implementation.
- Take-home message: don't just blindly follow an algorithm.

Merge-Sort

- $D(n) = O(1)$
- $C(n) = O(n)$
- According to the Merge-Sort algorithm, we divide the problem into 2 subproblems of size $\frac{n}{2}$.
- The resulting recurrence equation is:

$$T(n) = \begin{cases} O(1) & if\ n = 1 \\ 2T(\frac{n}{2}) + O(n) & if\ n > 1 \end{cases}$$

- By solving the recurrence equation we get $T(n) = O(n \log_2 n)$, much better than $O(n^2)$ for Insertion-Sort.
- There are several ways of solving this recurrence equation. The simplest one uses a recurrence tree, where each node in the tree indicates what the cost of solving it is. Then inspecting the tree can give us the solution.

- In this case, we get $log_2 n$, because we divide the problem in half in each recursive step.

Growth of Functions

- We need to define a precise notation and meaning for *order of growth* of functions.
- We usually don't look at exact running times because they are too complex to calculate and not interesting enough.
- Instead, we look at inputs of large enough size to make only the order of growth relevant **- asymptotic behavior**.
- Generally, algorithms that behave better asymptotically will be the best choice (except possibly for very small inputs).

Θ-notation ("Theta" notation)

- **Asymptotically <u>tight</u> bound**
- Idea: $f(n) = \Theta(g(n))$ if both functions grow "equally" fast
- Formally:

  $\Theta(g(n)) = \{f(n) \mid$ there exist positive constants c_1, c_2, and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0\}$
- $\Theta(1)$ means a *constant* function or a constant

O-notation ("Big-O" notation)

- **Asymptotic upper bound**
- Idea: $f(n) = O(g(n))$ if $g(n)$ grows faster than $f(n)$, possibly *much* faster
- Formally:

$$O(g(n)) = \{f(n) \mid \text{there exist positive constants } c \text{ and } n_0 \\ \text{such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$$

- Note: $f(n) = \Theta(g(n)) \Longrightarrow f(n) = O(g(n))$
- "Big-O" notation often allows us to describe runtime by just inspecting the general structure of the algorithm.
  - Example: The double loop in Insertion-Sort leads us to conclude that the runtime of the algorithm is $O(n^2)$.
- We use "Big-O" analysis for worst-case complexity since that guarantees an upper bound on *any* case.

Ω-notation ("Big-Omega" notation)

- **Asymptotic <u>lower</u> bound**
- Idea: $f(n) = \Omega(g(n))$ if $g(n)$ grows slower than $f(n)$, possibly *much* slower
- Formally:

$$\Omega(g(n)) = \{f(n) \mid \text{there exist positive constants } c \text{ and } n_0 \\ \text{such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}$$

- **Theorem:**

  For any two functions $f(n)$ and $g(n)$,

  $f(n) = \Theta(g(n))$ iff $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

o-notation ("Little-o" notation)

- **Upper bound that is not asymptotically tight**
- Idea: $f(n) = o(g(n))$ if $g(n)$ grows much faster than $f(n)$
- Formally:

$$o(g(n)) = \{f(n) \mid \text{for any positive constant } c > 0\text{, there exists a constant } n_0 > 0 \text{ such that } 0 \leq f(n) < cg(n) \text{ for all } n \geq n_0\}$$

- We will normally use "Big-O" notation since we won't want to make hard claims about tightness.

ω-notation ("Little-omega" notation)

- **<u>Lower</u> bound that is not asymptotically tight**
- Idea: $f(n) = \omega(g(n))$ if $g(n)$ grows much slower than $f(n)$
- Note: $f(n) \in \omega(g(n))$ if and only if $g(n) \in o(f(n))$
- Formally:

$$\omega(g(n)) = \{f(n) \mid \text{for any positive constant } c > 0 \text{, there exists a constant } n_0 > 0 \text{ such that } 0 \leq cg(n) < f(n) \text{ for all } n \geq n_0\}$$

Comparison of Functions

- The Θ, O, Ω, o, and ω relations are transitive.
  - Example:
    $f(n) = O(g(n)) \wedge g(n) = O(h(n)) \Longrightarrow f(n) = O(h(n))$
- The Θ, O, and Ω relations are reflexive.
  - Example: $f(n) = O(f(n))$
- Note the analogy to the comparison of two real numbers.
  e.g. $f(n) = O(g(n)) \approx a \leq b$
  e.g. $f(n) = \omega(g(n)) \approx a > b$

Comparison of Growth of Functions

- Transitivity: all
- Reflexivity:
  $f(n) = \Theta(f(n))$
  $f(n) = O(f(n))$
  $f(n) = \Omega(f(n))$
- Symmetry:
  $f(n) = \Theta(g(n))$ iff $g(n) = \Theta(f(n))$
- Transpose symmetry:
  $f(n) = O(g(n))$ iff $g(n) = \Omega(f(n))$
  $f(n) = o(g(n))$ iff $g(n) = \omega(f(n))$
- Lack of Trichotomy:
  Some $f(n)$ are neither $O(f(n))$ nor $\Omega(f(n))$