Complexity analysis

- Problem size $n$
  - Worst-case complexity: $\max$ # steps algorithm takes on any input of size $n$
  - Best-case complexity: $\min$ # steps algorithm takes on any input of size $n$
  - Average-case complexity: $\avg$ # steps algorithm takes on inputs of size $n$

The complexity of an algorithm associates a number $T(n)$, the best/worst/average-case time the algorithm takes, with each problem size $n$.

Mathematically,

- $T : \mathbb{N}^* \rightarrow \mathbb{R}^+$
- that is $T$ is a function that maps positive integers giving problem size to positive real numbers giving number of steps.

Why Worst-Case Analysis?

- Appropriate for time-critical applications, e.g. avionics
- Unlike Average Case, no debate about what the right definition is
- Analysis often easier
- Result is often representative of "typical" problem instances
- Of course there are exceptions…

Reading assignment

- Read Chapter 2 of *The ALGORITHM Design Manual*
O-notation etc

- Given two functions $f$ and $g: \mathbb{N} \to \mathbb{R}$
  - $f(n) = O(g(n))$ if there is a constant $c > 0$ so that $f(n)$ is eventually always $\leq c g(n)$
  - $f(n) = \Omega(g(n))$ if there is a constant $c > 0$ so that $f(n)$ is eventually always $\geq c g(n)$
  - $f(n) = \Theta(g(n))$ if there is a constant $c > 0$ so that $f(n)$ is eventually always $\leq c g(n)$ and $\geq c g(n)$

Examples

- $10n^2 - 16n + 100$ is $O(n^2)$ also $O(n^3)$
- $10n^2 - 16n + 100 \leq 11n^2$ for all $n \geq 10$
- $10n^2 - 16n + 100 = \Omega(n)$ also $\Omega(n)$
- $10n^2 - 16n + 100 \geq 9n^2$ for all $n \geq 16$
- Therefore also $10n^2 - 16n + 100$ is $\Omega(n^2)$

- $10n^2 - 16n + 100$ is not $O(n)$ also not $\Omega(n^2)$

Note: I don’t use notation $f(n) = O(g(n))$

Working with $O$, $\Omega$, $\Theta$ notation

- Claim: For any $a, b > 1$, $\log_a n$ is $\Theta(\log_b n)$
  - $\log_a n = \log_b n \cdot \log_b a$ so letting $c = \log_b a$ we get that $\log_a n \leq \log_b n \leq \log_a n$

- Claim: For any $a$ and $b > 0$, $(n+a)^b$ is $\Theta(n^b)$
  - $(n+a)^b \leq (2n)^b$ for $n \geq |a|$
  - $2^n = c \cdot 2^n$ for $c \cdot 2^n$ so $(n+a)^b$ is $O(n^b)$
  - $(n+a)^b = (n/2)^b$ for $n \geq 2|a|$
  - $2^n a^n = c' \cdot n$ for $c' \cdot 2^n$ so $(n+a)^b$ is $\Omega(n^b)$

Complexity

**Type of Complexity Analysis**

- Alg $A$
- Different running time for each input string

**Type of Bound**

- $T(n)$ grows like $n \log n$
- Function mapping input length to running time

Complexity analysis overview

We have looked at

- Type of complexity analysis
  - Worst-case, best-case, average-case
- Types of function bounds
  - $O$, $\Omega$, $\Theta$

These two considerations are orthogonal to each other

- One can do any type of function bound with any type of complexity analysis

Examples

- $10n^2 - 16n + 100$ is $O(n^2)$
- $10n^2 - 16n + 100 \leq 11n^2$ for all $n \geq 10$
- $10n^2 - 16n + 100 = \Omega(n)$
- $10n^2 - 16n + 100 \geq 9n^2$ for all $n \geq 16$
- Therefore also $10n^2 - 16n + 100$ is $\Omega(n^2)$

- $10n^2 - 16n + 100$ is not $O(n)$ also not $\Omega(n^2)$

Note: I don’t use notation $f(n) = O(g(n))$
General algorithm design paradigm
- Find a way to reduce your problem to one or more smaller problems of the same type
- When problems are really small solve them directly

Example
- Mergesort
  - on a problem of size at least 2
    - Sort the first half of the numbers
    - Sort the second half of the numbers
    - Merge the two sorted lists
  - on a problem of size 1 do nothing

Cost of Merge
- Given two lists to merge size \( n \) and \( m \)
  - Maintain pointer to head of each list
  - Move smaller element to output and advance pointer
- \[ \begin{array}{c|c}
\hline
\text{n} & \text{m} \\
\hline
\end{array} \]
- Worst case \( n+m-1 \) comparisons
  - Best case \( \min(n,m) \) comparisons

Recurrence relation for Mergesort
- In total including other operations let's say each merge costs 3 per element output
  - \( T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + 3n \) for \( n \geq 2 \)
  - \( T(1) = 1 \)
- Can use this to figure out \( T \) for any value of \( n \)
- \( T(5) = T(2) + T(2) + 3 \times 5 = 15 + 15 + 15 = 45 \)
- \( T(n) = 3n \log_2 n \)

Insertion Sort
- For \( i = 2 \) to \( n \) do
  - \( j \leftarrow i \)
  - while \( j > 1 \& X[j] > X[j-1] \) do
    - swap \( X[j] \) and \( X[j-1] \)
- i.e., For \( i = 2 \) to \( n \) do
  - Insert \( X[i] \) in the sorted list \( X[1], \ldots, X[i-1] \)

Recurrence relation for Insertion Sort
- Let \( T_a(i) \) be the worst case cost of creating list that has first \( i \) elements sorted out of \( n \).
- We want to know \( T_a(n) \)
  - The insertion of \( X[i] \) makes up to \( i-1 \) comparisons in the worst case
  - \( T_a(i) = T_a(i-1) + i-1 \) for \( i > 1 \)
  - \( T_a(1) = 0 \) since a list of length 1 is always sorted
  - Therefore \( T_a(n) = n(n-1)/2 \)
Solving recurrence relations
- e.g. \( T(n) = T(n-1) + f(n) \) for \( n \geq 1 \)
  \( T(0) = 0 \)
- solution is \( T(n) = \sum_{i=1}^{n} f(i) \)
- Insertion sort: \( T_n(i) = T_n(i-1) + i - 1 \)
- so \( T_n(n) = \sum_{i=1}^{n} (i - 1) = n(n-1)/2 \)

Arithmetic Series
- \( S = 1 + 2 + 3 + ... + (n-1) \)
- \( S = \frac{(n-1)n}{2} \)
- so \( S = n(n-1)/2 \)
- Works generally when \( f(i) = ai + b \) for all \( i \)
- Sum = average term size \( \times \) # of terms

Quick sort
- \( \text{QuickSort}(X, \text{left}, \text{right}) \)
  if \( \text{left} < \text{right} \)
  \( \text{split} = \text{Partition}(X, \text{left}, \text{right}) \)
  \( \text{QuickSort}(X, \text{left}, \text{split}-1) \)
  \( \text{QuickSort}(X, \text{split}+1, \text{right}) \)

Partition - two finger algorithm
- \( \text{Partition}(X, \text{left}, \text{right}) \)
  choose a random element to be a pivot and pull it out of the array, say at left end
  maintain two fingers starting at each end of the array
  slide them towards each other until you get a pair of elements where right finger has a smaller element and left finger has a bigger one (when compared to pivot)
  swap them and repeat until fingers meet
  put the pivot element where they meet

Partition - two finger algorithm
- \( \text{Partition}(X, \text{left}, \text{right}) \)
  swap \( X[\text{left}], X[\text{random(left, right)}] \)
  pivot \( \leftarrow X[\text{left}] \); \( L \leftarrow \text{left}; R \leftarrow \text{right} \)
  while \( L < R \) do
    while \( (X[L] \leq \text{pivot} \&\& L < \text{right}) \) do
      \( L \leftarrow L+1 \)
    while \( (X[R] > \text{pivot} \&\& R > \text{left}) \) do
      \( R \leftarrow R-1 \)
    if \( L > R \) then swap \( X[L], X[R] \)
  swap \( X[\text{left}], X[\text{right}] \)
  return \( R \)

In practice
- often choose pivot in fixed way as
  - middle element for small arrays
  - median of 1st, middle, and last for larger arrays
  - median of 3 medians of 3 (9 elements in all) for largest arrays

  four finger algorithm is better
  - also maintain two groups at each end of elements equal to the pivot
  - swap them all into middle at the end of Partition
  - equal elements are bad cases for two fingers
Quicksort Analysis

- Partition does n-1 comparisons on a list of length n
  - pivot is compared to each other element
- If pivot is ith largest then two sub-problems are of size i-1 and n-1
  - If pivot is always in the middle get
    \[ T(n) = \frac{n}{2}\log_2 n + \text{better than Mergesort} \]
  - If pivot is always at the end get
    \[ T(n) = T(n-1) + n-1 \] comparisons
  - \[ T(n) = (n-1)/2 \] like Insertion Sort

Quicksort Analysis Average Case

- Recall
  - Partition does n-1 comparisons on a list of length n
- If pivot is ith largest then two sub-problems are of size i-1 and n-1
  - Pivot is equally likely to be any one of 1st through nth largest
    \[ T(n) = n-1 + \frac{1}{n} \sum_{i=1}^{n} (T(i-1) + T(n-i)) \]

Quicksort analysis

\[ T(n) = n-1 + \frac{1}{n} \sum_{i=1}^{n} (T(i-1) + T(n-i)) \]
\[ = n-1 + \frac{2}{n} \sum_{i=1}^{n} T(i) + 2T(2) + \ldots + 2T(n-1) \]
\[ = nT(n) = n(n-1) + 2T(1) + 2T(2) + \ldots + 2T(n-1) \]
\[ = (n+1)T(n+1) = (n+1)n + 2T(1) + 2T(2) + \ldots + 2T(n) \]
\[ \therefore (n+1)T(n+1) - nT(n) = 2T(n) + 2n \]
\[ (n+1)T(n+1) = (n+2)T(n) + 2n \]
\[ T(n+1) = T(n) + \frac{2n}{n+1} \]
\[ \therefore T(n) = 1.38n \log_2 n \]

“Gestalt” Analysis of Quicksort

- Look at elements that ended up in positions j < k of the final sorted array
- The expected # of comparisons in Qsort = the expected # of j < k such that the jth and kth elements were compared
- = \[ \sum_{j<k} Pr[j\text{th and } k\text{th elts were compared}] \]

Quicksort execution
“Gestalt” Analysis of Quicksort

- Look at elements that end up in positions \( j < k \) of the final sorted array
- What is the chance that they were compared to each other during the course of the algorithm?
  - They started off together in the same sub-problem
  - They ended up in different sub-problems
  - The only time they might have been compared to each is when they were split into separate sub-problems

- The only time they might have been compared to each is when they were split into separate sub-problems
  - The pivot could be \( j^{th} \) or \( k^{th} \)
  - Those are the only cases when they are compared
  - Chances of that happening is 2 out of \((k - j + 1)\) equally likely possibilities

Total cost of Quicksort

- Total expected cost
  \[
  \sum_{k=j}^{2} \frac{2}{k - j + 1}
  \]
  - The contribution for each \( j \) is at most
    \[
    \frac{1}{2}, \frac{1}{3} + \frac{1}{4} + \ldots + \frac{1}{n} \leq 2 \log n
    \]
  - Total \( 2n \log n \) = 1.38 \( n \log n \)