Algorithms, Design and Analysis

Big-Oh analysis, Brute Force, Divide and conquer intro

Types of formulas for basic operation count

- Exact formula
e.g., \( C(n) = \frac{n(n-1)}{2} \)

- Formula indicating order of growth with specific multiplicative constant
e.g., \( C(n) \sim 0.5 n^2 \)

- Formula indicating order of growth with unknown multiplicative constant
e.g., \( C(n) \sim cn^2 \)

Order of growth

- Most important: Order of growth within a constant multiple as \( n \to \infty \)

- Example:
  - How much faster will algorithm run on computer that is twice as fast?
  - How much longer does it take to solve problem of double input size?

- See table 2.1

Asymptotic growth rate

- A way of comparing functions that ignores constant factors and small input sizes

- \( O(g(n)) \): class of functions \( f(n) \) that grow no faster than \( g(n) \)

- \( \Theta(g(n)) \): class of functions \( f(n) \) that grow at same rate as \( g(n) \)

- \( \Omega(g(n)) \): class of functions \( f(n) \) that grow at least as fast as \( g(n) \)

see figures 2.1, 2.2, 2.3

Table 2.1

<table>
<thead>
<tr>
<th>( n )</th>
<th>( \log_2 n )</th>
<th>( n )</th>
<th>( n \log_2 n )</th>
<th>( n^2 )</th>
<th>( 2^n )</th>
<th>( n^3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 10^2 )</td>
<td>3.3</td>
<td>( 10^3 )</td>
<td>1.3 - 6.3 ( 10^2 )</td>
<td>( 10^2 )</td>
<td>( 10^4 )</td>
<td>3.6 ( 10^3 )</td>
</tr>
<tr>
<td>( 10^3 )</td>
<td>6.6</td>
<td>( 10^4 )</td>
<td>6.6 - 6.6 ( 10^2 )</td>
<td>( 10^4 )</td>
<td>( 10^6 )</td>
<td>3.3 ( 10^5 )</td>
</tr>
<tr>
<td>( 10^4 )</td>
<td>10</td>
<td>( 10^5 )</td>
<td>1.0 - 1.0 ( 10^4 )</td>
<td>( 10^6 )</td>
<td>( 10^8 )</td>
<td></td>
</tr>
<tr>
<td>( 10^5 )</td>
<td>13.3</td>
<td>( 10^6 )</td>
<td>1.3 - 1.3 ( 10^5 )</td>
<td>( 10^8 )</td>
<td>( 10^{10} )</td>
<td></td>
</tr>
<tr>
<td>( 10^6 )</td>
<td>17</td>
<td>( 10^7 )</td>
<td>1.7 - 1.7 ( 10^6 )</td>
<td>( 10^{10} )</td>
<td>( 10^{12} )</td>
<td></td>
</tr>
<tr>
<td>( 10^7 )</td>
<td>20</td>
<td>( 10^8 )</td>
<td>2.0 - 2.0 ( 10^7 )</td>
<td>( 10^{12} )</td>
<td>( 10^{14} )</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1 Values (some approximate) of several functions important for analysis of algorithms

Big-oh

Figure 2.1 Big-oh notation: \( f(n) \in O(g(n)) \)
Establishing rate of growth: Method 1 – using limits

\[
\lim_{n \to \infty} \frac{T(n)}{g(n)} = \begin{cases} 
0 & \text{order of growth of } T(n) \text{ is less than order of growth of } g(n) \\
8 & \text{order of growth of } T(n) \text{ is greater than order of growth of } g(n) \\
\end{cases}
\]

Examples:
- \(10^n\) vs. \(2n^3\)
- \(n(n+1)/2\) vs. \(n^2\)
- \(\log_2 n\) vs. \(\log_3 n\)

L'Hôpital's rule
- \(\lim_{n \to \infty} f(n) = \lim_{n \to \infty} g(n) = \infty\)
- The derivatives \(f', g'\) exist,

Then
\[
\lim_{n \to \infty} \frac{f(n)}{g(n)} = \frac{\lim_{n \to \infty} f'(n)}{\lim_{n \to \infty} g'(n)}
\]

Examples: \(\log n\) vs. \(n\)

Establishing rate of growth: Method 2 – using definition

- \(f(n)\) is \(O(g(n))\) if order of growth of \(f(n)\) = order of growth of \(g(n)\) (within constant multiple)
- There exist positive constant \(c\) and non-negative integer \(n_0\) such that
  \[
  f(n) = c g(n) \quad \text{for every } n = n_0
  \]

Examples:
- \(10n\) is \(O(2n^2)\)
- \(5n+20\) is \(O(10n)\)

Basic Asymptotic Efficiency classes

| \(n^1\) | constant |
| \(\log n\) | logarithmic |
| \(n\) | linear |
| \(n\log n\) | linear-log |
| \(n^2\) | quadratic |
| \(n^3\) | cubic |
| \(2^n\) | exponential |
| \(n!\) | factorial |
More Big-Oh Examples

- $7n - 2$
  $7n - 2$ is $O(n)$
  need $c > 0$ and $n_0 \geq 1$ such that $7n - 2 \leq c \cdot n$ for $n \geq n_0$
  this is true for $c = 7$ and $n_0 = 1$

- $3n^3 + 20n^2 + 5$
  $3n^3 + 20n^2 + 5$ is $O(n^3)$
  need $c > 0$ and $n_0 \geq 1$ such that $3n^3 + 20n^2 + 5 \leq c \cdot n^3$ for $n \geq n_0$
  this is true for $c = 4$ and $n_0 = 21$

- $3 \log n + \log \log n$
  $3 \log n + \log \log n$ is $O(\log n)$
  need $c > 0$ and $n_0 \geq 1$ such that $3 \log n + \log \log n \leq c \cdot \log n$ for $n \geq n_0$
  this is true for $c = 4$ and $n_0 = 2$

Big-Oh Rules

- If is $f(n)$ a polynomial of degree $d$, then $f(n)$ is $O(n^d)$, i.e.,
  1. Drop lower-order terms
  2. Drop constant factors

- Use the smallest possible class of functions
  - Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"

- Use the simplest expression of the class
  - Say "$3n + 5$ is $O(n)$" instead of "$3n + 5$ is $O(3n)$"

Intuition for Asymptotic Notation

Big-Oh
- $f(n)$ is $O(g(n))$ if $f(n)$ is asymptotically less than or equal to $g(n)$

Big-Omega
- $f(n)$ is $\Omega(g(n))$ if $f(n)$ is asymptotically greater than or equal to $g(n)$

Big-Theta
- $f(n)$ is $\Theta(g(n))$ if $f(n)$ is asymptotically equal to $g(n)$

Little-Oh
- $f(n)$ is $o(g(n))$ if $f(n)$ is asymptotically strictly less than $g(n)$

Little-Omega
- $f(n)$ is $\omega(g(n))$ if $f(n)$ is asymptotically strictly greater than $g(n)$

Brute Force

A straightforward approach usually based on problem statement and definitions
Examples:
1. Computing $a^n$ ($a > 0$, $n$ a nonnegative integer)
2. Computing $n!$
3. Selection sort
4. Sequential search

More brute force algorithm examples:

- Closest pair
  - Problem: find closest among $n$ points in $k$-dimensional space
  - Algorithm: Compute distance between each pair of points
  - Efficiency

- Convex hull
  - Problem: find smallest convex polygon enclosing $n$ points on the plane
  - Algorithm: For each pair of points $P_1$ and $P_2$, determine whether all other points lie to the same side of the straight line through $P_1$ and $P_2$
  - Efficiency

Brute force strengths and weaknesses

- Strengths:
  - wide applicability
  - simplicity
  - yields reasonable algorithms for some important problems
    - searching
    - string matching
    - matrix multiplication
  - yields standard algorithms for simple computational tasks
    - sum/product of $n$ numbers
    - finding max/min in a list

- Weaknesses:
  - rarely yields efficient algorithms
  - some brute force algorithms unacceptably slow
Divide and Conquer
The most well known algorithm design strategy:
1. Divide instance of problem into two or more smaller instances
2. Solve smaller instances recursively
3. Obtain solution to original (larger) instance by combining these solutions

Divide and Conquer Examples
- Sorting: mergesort and quicksort
- Tree traversals
- Binary search
- Matrix multiplication-Strassen’s algorithm
- Convex hull-QuickHull algorithm

General Divide and Conquer recurrence:
\[ T(n) = aT(n/b) + f(n) \quad \text{where } f(n) \leq T(n^c) \]
1. \( a < b^k \) \( T(n) \leq T(n^k) \)
2. \( a = b^k \) \( T(n) \leq T(n^k \log n) \)
3. \( a > b^k \) \( T(n) \leq T(n^{\log_a b}) \)

Note: the same results hold with \( O \) instead of \( T \).

Mergesort
Algorithm:
- Split array \( A[1..n] \) in two and make copies of each half in arrays \( B[1..n/2] \) and \( C[1..n/2] \)
- Sort arrays \( B \) and \( C \)
- Merge sorted arrays \( B \) and \( C \) into array \( A \) as follows:
  - Repeat the following until no elements remain in one of the arrays:
    - compare the first elements in the remaining unprocessed portions of the arrays
    - copy the smaller of the two into \( A \), while incrementing the index indicating the unprocessed portion of that array
  - Once all elements in one of the arrays are

Mergesort Example
7 2 1 6 4
Efficiency of mergesort

• All cases have same efficiency: $T(n \log n)$

• Number of comparisons is close to theoretical minimum for comparison-based sorting:
  $\log n! \approx n \lg n \approx 1.44n$

• Space requirement: $T(n)$ (NOT in-place)

• Can be implemented without recursion (bottom-up)

Efficiency of quicksort

• **Best case**: split in the middle — $T(n \log n)$

• **Worst case**: sorted array! — $T(n^2)$

• **Average case**: random arrays — $T(n \log n)$

• Improvements:
  – better pivot selection: median of three partitioning avoids worst case in sorted files
  – switch to insertion sort on small subfiles
  – elimination of recursion
  these combine to 20-25% improvement

  • Considered the method of choice for internal sorting for large files ($n = 10000$)

Quicksort

• Select a pivot (partitioning element)

• Rearrange the list so that all the elements in the positions before the pivot are smaller than or equal to the pivot and those after the pivot are larger than the pivot (See algorithm Partition in section 4.2)

• Exchange the pivot with the last element in the partition

• Sort the two sublists

QuickHull Algorithm

Inspired by QuickSort compute Convex Hull:

• Assume points are sorted by x-coordinate values

• Identify extreme points $P_1$ and $P_2$ (part of hull)

• Compute upper hull:
  – find point $P_{max}$ that is farthest away from line $P_1P_2$
  – compute the hull of the points to the left of line $P_1P_{max}$

• Compute lower hull in a similar manner

The partition algorithm

```
Algorithm Partition(A[], p)
// Partition a subarray A[p..r] into two subarrays A[0..r]
// Indicate the position after the partition returned
// by returning the pivot.
\begin{verbatim}
\text{p = A[p]} \\
\text{i = i + 1} \\
\text{repeat} \\
\text{repeat until (i < j)} \\
\text{\text{if A[i] > A[p]} then} \\
\text{i = i + 1} \\
\text{\text{else}} \\
\text{\text{if A[j] < A[p]} then} \\
\text{j = j + 1} \\
\text{\text{else}} \\
\text{swap(A[i], A[j])} \\
\text{\text{swap(A[p], A[j])}} \\
\text{return j}
\end{verbatim}
```

Quicksort Example

15 22 13 27 12 10 20 25
Efficiency of QuickHull algorithm

- Finding point farthest away from line $P_1P_2$ can be done in linear time
- This gives same efficiency as quicksort:
  - Worst case: $T(n)$
  - Average case: $T(n \log n)$

- If points are not initially sorted by x-coordinate value, this can be accomplished in $T(n \log n)$ — no increase in asymptotic efficiency class
- Other algorithms for convex hull:
  - Graham's scan
  - DCHull