Basics of Algorithm Analysis

• We measure running time as a function of \( n \), the size of the input (in bytes assuming a reasonable encoding).

• We work in the RAM model of computation. All “reasonable” operations take “1” unit of time. (e.g. +, *, -, /, array access, pointer following, writing a value, one byte of I/O...)

What is the running time of an algorithm

• Best case (seldom used)
• Average case (used if we understand the average)
• Worst case (used most often)

We measure as a function of \( n \), and ignore low order terms.

• \( 5n^3 + n - 6 \) becomes \( n^3 \)
• \( 8n \log n - 60n \) becomes \( n \log n \)
• \( 2^n + 3n^4 \) becomes \( 2^n \)
Asymptotic notation

big-O

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

Alternatively, we say

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

Informally, \( f(n) = O(g(n)) \) means that \( f(n) \) is asymptotically less than or equal to \( g(n) \).

big-Ω

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

Alternatively, we say

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

Informally, \( f(n) = \Omega(g(n)) \) means that \( f(n) \) is asymptotically greater than or equal to \( g(n) \).
\textbf{big-Θ}

\( f(n) = \Theta(g(n)) \) if and only if \( f(n) = O(g(n)) \) and \( f(n) = \Omega(g(n)) \).

Informally, \( f(n) = \Theta(g(n)) \) means that \( f(n) \) is asymptotically equal to \( g(n) \).

\textbf{INFORMAL summary}

- \( f(n) = O(g(n)) \) roughly means \( f(n) \leq g(n) \)
- \( f(n) = \Omega(g(n)) \) roughly means \( f(n) \geq g(n) \)
- \( f(n) = \Theta(g(n)) \) roughly means \( f(n) = g(n) \)
- \( f(n) = o(g(n)) \) roughly means \( f(n) < g(n) \)
- \( f(n) = \omega(g(n)) \) roughly means \( f(n) > g(n) \)

We use these to classify algorithms into classes, e.g. \( n, n^2, n \log n, 2^n \).

See chart for justification
Polynomial Time

The size of a problem instance typically is described by parameters such as:

- number of nodes $n$ or $V$
- number of edges $m$ or $E$
- largest capacity $U$
- largest cost (in absolute value) $C$

**Input size:** The size of the input, which consists of a list of nodes and edges and their capacities and costs is typically

\[ \Theta(n + m + m \log U + m \log C) \]

- A **polynomial algorithm** is one whose running time is polynomial in the input, i.e. is polynomial in $n$, $m$, $\log U$, $\log C$.
- A **strongly polynomial algorithm** is one whose running time is polynomial in the size of the graph and independent of the size of the numbers, i.e. is polynomial in $n$, $m$.
- A **pseudo-polynomial algorithm** is one whose running time is polynomial in the size of the graph and the magnitude of the numbers, i.e. is polynomial in $n$, $m$, $U$, $C$. 
Commentary (with trivial interpretations excluded)

• Strongly polynomial and polynomial algorithms are polynomial algorithms. Pseudo-polynomial algorithms are not polynomial algorithms.

• Strongly polynomial algorithms are mainly a theoretical concept and do not tend to get used in practice.

We will typically shoot for polynomial algorithms.
Some Graph terminology

- node, vertex
- edge, arc
- directed undirected
- head tail
- path
- cycle
- acyclic
- bipartite graph
- tree
- forest
- cut
- $s$-$t$ cut
- connectivity
- strong connectivity
- bipartite graph
Easily Solved Graph Problems

- Connectivity
- Strong Connectivity
- Spanning trees
- Bipartiteness
- Topological Search
- Depth-first Search
- Breadth-first Search
Basic Data Structures

- Arrays
- Linked Lists
- Stack - LIFO
- Queue - FIFO
- Binary tree
- Hash table
Dictionary Operations on ordered set

- Insert
- Delete
- Find
- Min, Max
- Successor, Predecessor
- IncreaseKey, DecreaseKey

Comments

- Some form of a balanced binary tree supports all dictionary operations in $O(\log n)$ time
- A hash table supports Insert, Delete and Find in $O(1)$ expected time
Graph Storage

- An adjacency matrix is an $n$ by $n$ matrix in which $A[i, j]$ stores values related to edge $(i, j)$.

- An adjacency list is a length $n$ array $L$ of linked lists, where entry $L[i]$ is a list of all edges adjacent to vertex $i$. 