Mathematatic Basics

Exponent

Definition: $a^x = b$
Theorem 1: $a^x \times a^y = a^{x+y}$
Theorem 2: $a^x / a^y = a^{x-y}$ $(a \neq 0)$
Theorem 3: $(a^x)^y = a^{xy}$
Theorem 4: $a^{xy} = (a^x)^y$

Logarithm

Definition: if $a^x = b$, then $x = \log(a, b)$
lg: $\lg(N) = \log_{10}(N)$
ln: $\ln(N) = \log_e(N)$, where $e \approx 2.71828$
Theorem 1: $\log(a, b \times c) = \log(a, b) + \log(a, c)$
Proof: $\log(a, b \times c)$
=: $\log(a, a\text{^}\log(a, b)) \times a\text{^}\log(a, c)))$
=: $\log(a, a\text{^}(\log(a, b) + \log(a, c)))$
=: $\log(a, b) + \log(a, c)$
Theorem 2: $\log(a, b/c) = \log(a, b) - \log(a, c)$
Theorem 3: $\log(a, b^c) = c \times \log(a, b)$. This is actually an extension of theorem 1.
Proof: $\log(a, b^c)$
=: $\log(a, (a\text{^}\log(a, b))^c)$
=: $\log(a, a\text{^}(c \times \log(a, b)))$
=: $c \times \log(a, b)$
Theorem 4: $\log(a, b) = \log(c, b) / \log(c, a)$, $c > 0$
Proof: $\log(a, b)$
=: $\log(\ c\text{^}\log(c, a)\ ,\ c\text{^}\log(c, b)\ )$
=: $\log(c, b) / \log(c, a)$

Geometric Series

Theorem 1: $\Sigma(i = 0 \rightarrow n, r^i) = (1 - r^{n+1}) / (1 - r)$

Arithmetic Series

Theorem 1: $\Sigma(i = 1 \rightarrow n, i) = n(n + 1) / 2 \approx n^2 / 2$
Theorem 2: $\Sigma(i = 1 \rightarrow n, i^k) \approx n^{k+1} / |k + 1|$, where $k \neq -1$
Theorem 3: $\Sigma(i = 1 \rightarrow n, 1/i) = Hn \approx \ln(n)$
Theorem 4: $\Sigma(i = c \rightarrow n, f(n)) = (n-c+1) \times f(n)$
Theorem 5: $\Sigma(i = c \rightarrow n, f(i)) = \Sigma(i = 1 \rightarrow n, f(i)) - \Sigma(i = 1 \rightarrow c - 1, f(i))$

Proof Techniques

Deduction:
Induction:
Counterexample:
Contradiction: Prove P is true by proving ~P is false. Prove A → B is true by proving A → ~B is false.
Contrapositive: Prove A → B is true by proving ~B → ~A is true

Analysis Basics

Definitions

Definition 1: $T(n) = O(f(n))$ if there are constants c and n_0 such that $T(n) \leq c \times f(n)$ when $n \geq n_0$

Theorem 1: $O(f) + O(g) = \max(O(f), O(g))$

Theorem 2: $O(f) \times O(g) = O(f \times g)$

Theorem 3: $O(f + g) = O(\max(f, g))$

Definition 2: $T(n) = \Omega(g(n))$ if there are constants c and n_0 such that $T(n) \geq c \times g(n)$ when $n \geq n_0$

Theorem 1: If $f(n) = O(g(n))$ then $g(n) = \Omega(f(n))$

Proof: If $f(n) = O(g(n))$

Then: $f(n) \leq c * g(n)$ when $n \geq n_0$

Then: $g(n) \geq 1/c * f(n)$ when $n \geq n_0$

Then: $g(n) = \Omega(f(n))$

Definition 3: $T(n) = \Theta(f(n))$ if $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$

Theorem 1: $\lim(n \to \infty, T(n) / f(n)) \to c$, where c is a constant

Definition 4: $T(n) = o(f(n))$ if $T(n) = O(f(n))$ and $T(n) \neq \Omega(f(n))$

Theorem 1: $\lim(n \to \infty, T(n) / f(n)) \to 0$

Growth Rates

O(c): constant

O(log n): logarithmic

Theorem: If $T(n) = T(n/2) + c$, then $T(n) = O(\log n)$

O($\log^k$ n):

Theorem: $\log^k n = O(n)$ for any constant k

O(n): linear

Theorem: If $T(n) = T(n - k) + c$, then $T(n) = O(n)$

O(n log n): n log n

Theorem: If $T(n) = 2T(n/2) + cn$, then $T(n) = O(n \log n)$

Proof: $T(n) = 2T(n/2) + cn$

Then: $T(n) = 4T(n/4) + cn \times 2$

Then: $T(n) = nT(n/n) + cn \times \log n$

=: $cn + cn \times \log n$

=: $O(n) + O(n \log n)$

=: $O(n \log n)$

O(n^2): quadratic

O(n^3): cubic

O(a^n, n!): exponential

Maximum Subsequence Sum

Algorithm 1: look at all pairs and add up the numbers in between, saving the largest

T: $\Sigma(i = 1 \to n, \Sigma(j = i \to n, \Sigma(k = i \to j, 1)))$

=: $\Sigma(i = 1 \to n, \Sigma(j = i \to n, j - i + 1))$

=: $\Sigma(i = 1 \to n, \Sigma(x = 1 \to n - i + 1, x))$

=: $\Sigma(i = 1 \to n, (n - i + 1)(n - i + 2) / 2)$

=: $1/2 \times \Sigma(i = 1 \to n, (n - i + 1)(n - i + 2))$

=: $1/2 \times \Sigma(i = 1 \to n, n^2 - in + 2n - in + i^2 - 2i + n - i + 2)$

=: $1/2 \times \Sigma(i = 1 \to n, n^2 + 3n + 2 + i^2 + i(-2n - 3))$

=: $1/2 \times \Sigma(i = 1 \to n, n^2 + 3n + 2) + 1/2 \times \Sigma(i = 1 \to n, i^2) + 1/2 \times (-2n - 3) \times \Sigma(i = 1 \to n, i)$

=: $O(n^3) + O(n^3) + O(n^2)$

=: $O(n^3)$

Algorithm 2: refinement of algorithm 1

T: $\Sigma(i = 0 \rightarrow n - 1, \Sigma(j = i \rightarrow n - 1, 1))$

=: $\Sigma(i = 0 \rightarrow n - 1, n - i)$

=: $\Sigma(x = 1 \rightarrow n, x)$

=: $\Sigma(x = 1 \rightarrow n, x)$

=: $n(n + 1) / 2$

=: $O(n^2)$

Algorithm 3: the maximum subsequence either is in one half, in the other half, or spans the middle of the array. To find the maximum sum in the first and second halves use recursion. We can then see if there is a larger subsequence that spans the middle of the array.

T: O(n log n)

Algorithm 4: relies crucially on fact that we do not consider negative sums but instead return 0 if nothing positive.

T: O(n)

Search

Binary Search

Algorithm:

T: O(log n)

Exponentiation

Algorithm:

```
long int pow(const long int& x, const int n) {
    if (n == 0) return 1;
    if (n%2 == 0) return pow(x*x, n/2); // n is even
    else return pow(x*x, n/2) * x; // n is odd
}
```

T: O(log n)

Sort

Bucket Sort

Algorithm: To sort n integers in the range 0 to m−1, create an array arr of m buckets and initialize each bucket to 0. Then for each integer i, increment arr[i]. Finally print out i arr[i] times.

T: O(n + m) = O(max(n, m))

Problem: The storage and running time is a function of magnitude of integers so to sort n integers or strings in range 0 to $b^p - 1$ it is infeasible to allocate b^p buckets. This can be resolved by Radix Sort.

Radix Sort

Algorithm: To sort n integers or strings in range 0 to $b^p - 1$, use b buckets and make p passes from least significant radix to most significant radix. During each radix, run through the current sequence of integers or strings, put them into different buckets representing different values of that radix, and then reconstruct the sequence of integers or strings, from lexicographically larger bucket to smaller one (to make the values at the current radix position sorted), from bottom of bucket to top (to keep the values at the next less significant radix position remain sorted).

T: O((n + b) * p)

Merge Sort

Algorithm: sort the first half of the array, sort the second half of the array, merge the two halves of the array

T: 2T(n/2) + cn = O(n log n)

Quicksort

Algorithm:

- **Tb**: O(n log n)
- **Ta**: O(n log n)
- **Tw**: O(n^2)

Finding the k-th largest element:

- **Tb**: O(n)
- **Ta**: O(n)

Shellsort

Algorithm: k-sort means to sort the array so that a[i] ≤ a[i+k] for any valid i

Shell: k0 = n/2, k = k/2

- **Tw**: Θ(n^2)

Hibbard: k0 ≤ n/2, k = 2^i - 1 where i = 1, 2, ...

- **Tw**: Θ(n^1.5)

Theorems

Two-way comparison sorting algorithms: T(n) = Ω(n log n)

General Solutions to Divide and Conquer: T(1) = b, T(n) = aT(n/c) + bn

- **a < c**: T(n) = Θ(n)
- **a = c**: T(n) = Θ(n log(c, n))
- **a > c**: T(n) = Θ(n^log(c, a))

Graph

Graph

Graph:

- k-connected: there are k vertex-disjoint paths between every pair of nodes

Directed graph:

- connected: there is some path from every vertex to every other vertex

Undirected graph:

- strongly connected: there is some path from every vertex to every other vertex
- weakly connected: not strongly connected but the underlying undirected graph is connected

Complete graph: every pair of vertices has an edge between them.

- **|E|**: If directed |E| = n(n - 1). If undirected, |E| = nC2

Representing a graph:

- Adjacency matrix: Θ(|V|^2)
- Adjacency list: O(|V| + |E|)

Path

Loop / Self-edge: e = (v, v)

Path: v_1, v_2, ..., v_n such that (vi, vi+1) ∈ E

- length: the number of edges

Simple Path: a path such that v_2, ..., v_n are distinct

Cycle: a path such that $v_1 = v_n$

Simple Cycle: a simple path such that $v_1 = v_n$

DAG TS (Topological Sort)

Topological Sort: Find a node with 0 indegree; Record this node; Remove it and any of its edges; Repeat |V | times

T: O(n^2)

T: O(|E | + |V|), for Modified TS

Unweighted Shortest Path

Breadth-First Search (BFS): Spread out from the vertex, one level at a time.

T: O(|E | + |V|)

Weighted Shortest Path

Dijkstra's Algorithm: no negative cost

T: O(|E| log |V|)

Brute Force Algorithm: no cycles of negative cost

T: O(|E||V|)

Biconnectivity

The removal of any single vertex maintains the connectedness of the graph

Articulation point / Cut vertex: the node that prevents a graph from being biconnected

Proof of 1: if the node is the root and it has more than one child, then must be no path from the lexicographically-least node to any other children of the root.

Proof of 2: for a node not to be an AP there must be a back edge from a node later to a node earlier.

Biconnected components: APs partition a graph into bi-connected components. Vertices u and v are in the same bi-connected component if there are two or more vertex disjoint paths from u to v

Depth First Search: follows a path as deeply into the graph as it can

1 iff node v is the root and it has more than one child

2 or node v is any other node and it has some child w such that low(w) ≥ num(v)

num(v) is computed by numbering the vertices as they appear in a DFS (preorder)

low(v) is therefore the minimum of

1. num(v) (Rule 1)
2. the lowest num(w) among all back edges (Rule 2)
3. the lowest low(w) among all tree edges (Rule 3)

T: O(|E | + |V|)

Minimum Spanning Trees

Kruskal's Algorithm: Sort edges by weight; Add to tree if it does not create a cycle; Stop when |V | − 1 edges added

T: O(|E| log |E|)

Prim's Algorithm: Add to tree the smallest edge connecting one vertex in the tree and one vertex not in

T: O(|E| log |V|)