Asymptotic Notation

Consider the following theorems.

Thereom 1 – Big O Notation:

$$f(n) = O(g(n)) \text{ if } \exists\ C, N > 0 \text{ such that } 0 \leq f(n) \leq Cg(n)\ \forall\ n \geq N$$

Thereom 2 – Big Ω Notation:

$$f(n) = \Omega(g(n)) \text{ if } \exists\ c, N > 0 \text{ such that } 0 \leq cg(n) \leq f(n)\ \forall\ n \geq N$$

Thereom 3 – Θ Notation:

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

The general method to showing that a given $f(n)$ and $g(n)$ satisfies the above Theorems, is to rearrange the inequality with $f(n)/g(n)$ or $g(n)/f(n)$ in the middle of the inequality so it is bounded by zero on the left-hand side and some constant on the right-hand side. The Theorem satisfaction criteria are thus:

- If $f(n)/g(n)$ converges, then it will satisfy Theorem 1. It can be said that $f(n)$ does not grow faster than $g(n)$.
- If $f(n)/g(n)$ does not converge, then it is divergent and does not satisfy Theorem 1. There is no need to check the type of divergence.
- If $g(n)/f(n)$ converges, then it will satisfy Theorem 2. It can be said that $f(n)$ does not grow slower than $g(n)$.
- If $g(n)/f(n)$ does not converge, then it is divergent and does not satisfy Theorem 2. There is no need to check the type of divergence.

Recurrence

Using recurrence to solve a problem of size n reduces the problem to a many subproblems of a smaller size n/b. The overhead cost of reducing the problem and combining the solutions of the subproblems is $f(n)$.

The time complexity of the recurrence is given as:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Reducing the $T(n/b)$ term $log_b a$ times yields:

$$T(n) \approx n^{log_b a} T(1) + \sum_{i=0}^{\lfloor log_b n \rfloor - 1} a^i f\left(\frac{n}{b^i}\right)$$

Master Theorem

Master Theorem is used to find the time complexity of `typical' recurrence problems.

The critical polynomial is given as:

$$n^{c^*} = n^{log_b a}$$

1. If $f(n) = O(n^{c^*-\epsilon})$ for some $\epsilon > 0$, then $T(n) = \Theta(n^{c^*})$.

2. If $f(n) = \Theta(n^{c^*})$, then $T(n) = \Theta(n^{c^*} log(n))$.

3. If $f(n) = \Omega(n^{c^*+\epsilon})$ for some $\epsilon > 0$, and $af(n/b) \leq cf(n)$ holds for some $c < 1$ and holds $\forall$ $n > N$, then $T(n) = \Theta(f(n))$.

4. Otherwise, Master Theorem does not apply.

The general method to using the Master Theorem is to identify a, b, and $f(n)$; determine the critical polynomial; and setting ϵ to be some appropriate value. The Theorem satisfaction criteria are thus:

- If time complexity of $f(n)$ is less than time complexity of critical polynomial, then case 1 applies.

- If time complexity of $f(n)$ is equal to time complexity of critical polynomial, then case 2 applies.

- If time complexity of $f(n)$ is greater than time complexity of critical polynomial and satisfies the inequality, then case 3 applies.

- If none of the above applies, then the time complexity of the recurrence must be determined by hand (evaluating the recurrance equation to get an expanded series then simplified).

Fast Fourier Transform

Inverse discrete Fourier transform (IDFT) is a sequence of coefficients of a polynomial. Discrete Fourier transform (DFT) is a sequence of values of a polynomial at the root of unity of order m:

$$\langle A_0, A_1, ..., A_n, \underbrace{0, ..., 0}_{n} \rangle \xrightarrow{\text{DFT}} \langle \hat{A}_0, \hat{A}_1, ..., \hat{A}_{m-1} \rangle$$

$$\langle B_0, B_1, ..., B_n, \underbrace{0, ..., 0}_{n} \rangle \xrightarrow{\text{DFT}} \langle \hat{B}_0, \hat{B}_1, ..., \hat{B}_{m-1} \rangle$$

$$\langle C_0, C_1, ..., C_{m-1} \rangle \xleftarrow{\text{IDFT}} \langle \hat{C}_0, \hat{C}_1, ..., \hat{C}_{m-1} \rangle$$

Fast Fourier transform (FFT) is an algorithm used to compute the DFT or IDFT. It has a time complexity of $O(nlog(n))$.

Basically, to multiply a sequence of integers very quickly, use FFT.

Greedy Algorithm

Greedy algorithm solves a problem by dividing it into stages and considering the choice that is best at the time. It may not be clear if the locally optimal choice leads to a globally optimal solution. Therefore to prove correctness of our solution, consider two correctness proofs:

Greedy Stays Ahead

Prove at every stage, there is no local solution better than the greedy algorithm's local solution.

1. Label our algorithm's partial solution, $A(1..k)$, and a general partial solution, $O(1..m)$.

2. Assume O is not the same as A e.g. A is naturally ordered by greedy algorithm; but O is optimally ordered (which optimises the measure).

3. Define a measure, $f(a_1..a_j)$ and $f(o_1..o_j)$, where greedy algorithm is ahead of O e.g. f is last completion of first j elements; total weight of first j elements; or largest index of first j elements.

4. Define a goal to compare measures e.g. $f(a_1..a_j) \leq f(o_1..o_j)$; or $f(a_1..a_j) \geq f(o_1..o_j)$ $\forall$ $j \leq k$.

5. Prove the above goal by induction.

6. $A(1..j)$ stays ahead of $O(1..j)$ with respect to $f(1..j)$, therefore $A(1..j)$ is optimal.

Exchange Argument

Consider a correct algorithm to the solution and morph it into the greedy solution without making any worse.

1. Label our algorithm's solution, A, and general solution, O.

2. Assume O is not the same as A e.g. there is an element of O not in A and an element of A not in O; or there are 2 consecutive elements in O in different order than in A (inversion).

3. Explain how A and O can be different.

4. Swap elements in O and argue that O is no worse than A.

5. Argue that we can continue to swap and eliminate all differences between O and A without worsening O.

6. Therefore A is just as good as O and is therefore optimal itself.

Kruskal's Algorithm

Outputs a minimum spanning tree (MST) for a time complexity of $O(|E|log(|V|))$.

Dynamic Programming

Dynamic programming solves a problem by recursively solving overlapping subproblems with optimal subsolutions which are combined into an optimal solution for the full problem. Dynamic programming problems consist of defining the subproblem, recurrence relation, and base cases. The general method to solving these problems is:

1. For $1 \leq i \leq n$, let $P(i)$ be the problem of determining $opt(i)$.

2. Define $opt(i)$ in context to the problem.

3. Define the recurrence relation: $opt(i) = min/max(opt(?), opt(?))$.

4. Define base cases e.g. $opt(1) = ?$.

5. Identify the order of which subproblems should be solved if appropriate e.g. i must be solved in descending order.

6. State the final solution e.g. $max_{1 \leq i \leq n}(opt(i))$.

7. Determine the time complexity e.g. there are $O(n)$ subproblems, each solved in $O(n)$ time, therefore the overall time complexity is $O(n^2)$.

It is possible to have multiple iterators, if more information needs to be passed recursively.

Max Flow

Max flow problems consist of a flow network construction then application of some max flow min cut algorithm.

Flow Network Construction

Flow networks are directed graphs which have *capacity constraint* and *flow conservation.* To fully construct a flow network, do the following:

1. Let there be a flow network represented as a directed graph.

2. Define the flow.

3. Define the nodes.

4. Define the node capacities.

5. Define the edges.

6. Define the edge capacities.

7. Define the source.

8. Define the sink.

Max Flow Min Cut

The max flow of a flow network is equal to the min cut's capacity (cutting across a set of edges which bisects the flow network). To find the max flow, do the following:

1. Define the max flow, f.

2. Apply Ford-Fulkerson, $O(|E|f)$, or Edmonds-Karp, $O(min(|V||E|^2, |E|f))$, algorithm to compute f.

3. Explain how f is used.

Max Bipartite Matching

Max bipartite matching problem can be identified from a flow network problem with two disjoint sets of nodes. Convert the max bipartite matching problem to a flow network problem by modifying the flow network construction with the following:

1. Replace the source with a super-source which is an arbitrary coordinate to a set of nodes.

2. Define the capacity of edges to super-source.

3. Replace the sink with a super-sink which is an arbitrary coordinate to a set of nodes.

4. Define the capacity of edges to super-sink.

Typically, capacities of edges to super-sources and super-sinks are 1.

The time complexities for both Ford-Fulkerson and Edmonds-Karp are $O(|E||V|)$.

String Matching

String matching is determining if a given string, A, has a contiguous given substring/pattern, B.

Rabin-Karp Algorithm

Rabin-Karp algorithm uses hashing and recursion to efficiently find matching strings in average $O(n+m)$ but is limited by unlikely possibility of hash collisions, therefore cannot guarantee worst case performance, $O(nm)$.

1. Map symbol $s_i \rightarrow i$: $S = \{s_0, s_1, ..., s_{d-1}\} \rightarrow \{0, 1, ..., d-1\}$.

2. Construct and evaluate hash integer using Horner's rule: $H(B) = h(b_0 b_1 ... b_m) = d^{m-1} b_0 + d^{m-2} b_1 + ... + d b_{m-2} + b_{m-1}$.

3. Choose large prime number: p.

4. Let $H(B) = h(B) \,\%\, p$.

5. Compute hash, $H(A_s)$, for each $A_s = a_s a_{s+1} ... a_{s+m-1}$ of A. Using recursion to compute hash: $H(A_{s+1}) = dH(A_s) - (d^m a_s + a_{s+m}) \,\%\, p$.

6. Compare hash values of $H(B)$ and $H(A_s)$. Do symbol-by-symbol matching $\iff$ $H(B) = H(A_s)$.

Finite Automata

Pattern B can be converted into a finite automata (simple state machine). The method (by-hand) to get the state transition table from B is:

1. Draw out a table with rows for each state (i.e. each index in B); and columns for each transition (i.e. each symbol in B).

2. Input the ???.

The method (by-hand) to get the state transition diagram from the table is:

1. Draw nodes for each row.

2. Draw directed arrows from the row's index to the index specified in the transition column and label it with the transition symbol.

Knuth-Morris-Pratt Algorithm

Knuth-Morris-Pratt algorithm uses finite automata and recursion to efficiently find matching strings in $O(n)$.

1. Suppose $B_s = b_0..b_{s-1}$ is longer prefix of B which is suffix of $A_i = a_0..a_{i-1}$.

2. If $a_i = b_s$ then let $s = s+1$; else, recursively check condition by letting $s = \pi(s)$.

3. If $s = m$ then string match found.

Linear Programming

Linear programming optimises linear functions subject to various conditions. Standard form of the primal linear program, P, is: maximise $z(\mathbf{x}) = \mathbf{c}^T\mathbf{x}$, subject to $A\mathbf{x} \leq \mathbf{b}$ and $\mathbf{x} \geq \mathbf{0}$.

The standard form of the dual linear program, P^*, is: minimise $z^*(\mathbf{y}) = \mathbf{b}^T\mathbf{y}$ subject to $A\mathbf{y} \geq \mathbf{c}$ and $\mathbf{y} \geq \mathbf{0}$.

Weak Duality Theorem

If $x = \langle x_1, ..., x_n \rangle$ is any feasible solution for P and $y = \langle y_1, ..., y_m \rangle$ is any feasible solution for P^* then:

$$z(x) = \sum_{j=1}^{n} c_j x_j \leq \sum_{i=1}^{n} b_i y_i = z^*(y)$$

Any x is a lower bound for P^* and any y is an upper bound for P, so if $x = y$ then optimal solution has been found.

Integer Linear Programming

Integer linear programming restricts some or all variables to be integers. Standard form is: maximise $\mathbf{c}^T\mathbf{x}$ subject to $A\mathbf{x} + \mathbf{s} = \mathbf{b}$, $\mathbf{s} \geq \mathbf{0}$, and $\mathbf{x} \geq \mathbf{0} \in \mathbb{Z}^n$.

Intractability

Consider the following classes of decision (true/false) problems:

- $A(x) \in \mathbf{P}$ (polynomial time) if $\exists$ a polynomial time algorithm which solves A.
- $A(x) \in \mathbf{NP}$ (non-deterministic polynomial time) if $\exists$ a problem $B(x, y)$ such that:
  - $A(x) = \text{true} \ \forall \ x \iff \exists$ some y for $B(x, y) = \text{true}$; and
  - $B(x, y)$ can be verified by a polynomial algorithm in length of x only.
- $A(x) \in \mathbf{NP} - \mathbf{C}$ (non-deterministic polynomial complete time) if every other **NP** problem is polynomially reducible to A i.e. U is polynomially reducible to $V \iff \exists \ f(x)$ such that:
  - $f(U) \rightarrow V \bigcap f(U) = V$
  - $f(x)$ is computable by a polynomial time algorithm.
- $A(x) \in \mathbf{NP} - \mathbf{H}$ (non-deterministic polynomial hard time) if every **NP** problem is polynomial time in A.

Cook's Theorem

Every **NP** problem is polynomially reducible to the boolean satisfiability (SAT) problem.