Algorithms Design and Analysis

Haeohreum Kim

I. Divide and Conquer

Divide and Conquer algorithms are defined by the splitting of problems into subproblems, such that the problems become simpler (usually, $\Theta(1)$).

A. Master Theorem

The master theorem provides an easy framework to analyse divide and conquer algorithms. For some $a > 0$ and $b > 1$, and some driving function $f(n)$, the master theorem is:

$$T(n) = a \cdot T(\frac{n}{b}) + f(n)$$

To figure the asymptotic complexity of the algorithm, we use the watershed function $n^{\log_b(a)}$ to consider three cases (informally); where $c^* = \log_b(a)$:

1) If $f(n) = O(n^{c^* - \epsilon})$ for some $\epsilon > 0$, then $T(n) = \Theta(n^{c^*})$.
2) If $f(n) = O(n^{c^*} \log^k n)$ for some $k \geq 0$, then $T(n) = \Theta(n^{c^* \log^{k+1} n})$
3) If $f(n) = \Omega(n^{c^* + \epsilon})$ for some $\epsilon > 0$, and for some $k < 1$ and some n_0,
$$af(\frac{n}{b}) \leq kf(n)$$
holds for all $n > n_0$ (the regularity condition), then $T(n) = \Theta(f(n))$.

B. Inversion Counting

Consider an array A, ranking a person A's preferences from $1 \to n$. The array B, ranks the person B's preferences using person A's ranks - that is, if person B likes person A's 5th preferred option, then the first array element would be 5.

The algorithm uses merge sort to count the inversions between three cases - the left subarray, the right subarray, as well as across the left and right subarray. In the above scenario, an inversion is whenever a large number comes before a small number. It runs in $\Theta(n \log n)$ time.

C. Karatsuba Multiplication

When trying to multiply large numbers using elementary techniques, we find that for some n bits, we have $\Theta(n^2)$ work to do. Rather, Karatsuba's algorithm splits some numbers n and m into half, say $n_{\text{hi}}, n_{\text{lo}}, m_{\text{hi}}, m_{\text{lo}}$. It should follow that:

$$n = 2^{\frac{n}{2}} \cdot n_{\text{hi}} + n_{\text{lo}}$$
$$m = 2^{\frac{n}{2}} \cdot m_{\text{hi}} + m_{\text{lo}}$$

Hence, now $n \cdot m$ becomes $2^n \cdot (n_{\text{hi}} \cdot n_{\text{lo}}) + 2^{\frac{n}{2}} \cdot (n_{\text{hi}} \cdot m_{\text{lo}} + m_{\text{hi}} \cdot n_{\text{lo}}) + n_{\text{lo}} \cdot m_{\text{lo}}$. The obvious next step is to then, also subdivide these multiplications into even smaller halves, which eventually become constant. The time complexity of Karatsuba's is $O(n^{1.6})$

D. Strassen's Matrix Multiplication Algorithm

Strassen's Algorithm splits square matrices into quadrants, which serve as their own matrices. The general premise is such that if we have a matrix equation:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae + bg & af + bh \\ ce + dg & cd + dh \end{bmatrix}$$

Then we can also have $a, b, c, \ldots$ be matrices themselves, and divide them until they become 2×2 matrices.

Note that multiplying will still be $\Theta(n^2)$, and splitting is $\Theta(1)$. The trick comes in making some expressions, such that we can reuse some of the values that were calculated. There are 7 overall products, instead of 8 products. Hence, the recursion function is:

$$7 \cdot T(\frac{n}{2}) + \Theta(n^2)$$

Therefore, our watershed function $n^{2.81}$, which is larger then the driving function, therefore, the algorithm's time complexity is $\Theta(n^{2.81})$.

E. DFT, FFT and Polynomials

The Discrete Fourier Transform (DFT) is used to transform a polynomial into a frequency domain. Ergo, the DFT tries to form some frequency pattern given some polynomial. The Fast Fourier Transform (FFT) is an algorithm that transforms polynomials into a complex number domain, using the DFT in a clever algorithm. The FFT uses complex numbers as a way to create a collapsible set of values. Here, instead of having to calculate every single value, we have to calculate half of the roots of unity then before for each recursive level.

```
n = len(P)
if n == 1:
  return P

w = exp((2 * Pi * i) / n)
P(even) = [even indices]
P(odd) = [odd indices]
y(even) = FFT(P(even))
y(odd) = FFT(P(odd))
y = [0] * n
for j in range(n / 2):
  y[j] = y(even)[j] + w^j * y(odd)[j]
  y[j+n/2] = y(even)[j] - w^j * y(odd)[j]

return y
```

Note that the $\frac{n}{2}$ notation denotes the mirrored roots of unity. Hence, in one step, we are able to calculate $y[j]$ and $y[j + \frac{n}{2}]$, which is the crux of this algorithm.

II. Greedy

Greedy algorithms take local optimal steps that build to a global optimal solution. It's clear that greedy algorithms don't apply to every scenario, as then every problem would be trivial. In 3121, you are expected to intuit the application of a greedy algorithm.

There are two main subgenres of greedy problems, which are **optimal selection** and **optimal ordering**. Optimal selection requires an algorithm that selects an item or a combination of items from an input, whereas an optimal ordering question requires the algorithm to reorder the given input to meet a certain standard.

A. Correctness of greedy algorithms

Greedy algorithms are hard to prove. They suffer from being myopic, and hence, it is often difficult to justify that the local step taken will always build to an optimal solution. There are two main ways to justify a greedy algorithm formally.

1) Exchange argument: The exchange argument is one in that for some alternative solution $B = \{b_0, b_1, \ldots, b_n\}$, that for the greedy solution $A = \{a_0, a_1, \ldots, a_m\}$, for any selection in B, you can exchange it with an appropiate selection in A, and it stay an optimal solution.

Consider the "Activity Selection" exercises proof below:
Suppose some greedy solution $G = \{g_1, \ldots, g_r\}$ and some alternative selection $A = \{a_1, \ldots, a_s\}$, each in ascending order of time. Suppose that $g_1 = a_1, g_2 = a_2, \ldots, g_{k-1} = a_{k-1}, g_k \neq a_k$.

Hence, we can define $A' = \{g_1, \ldots, g_k, a_{k+1}, \ldots, a_s\}$. There is certainly no conflicts $< k$, and since g_k finishes no later then a_k, there are no conflicts $\geq k$. Hence, A' is a valid selection. It can also be trivially said that the size of $A' = A$, and hence, the exchange maintains the optimal solution.

2) Greedy stays ahead: The greedy stays ahead argument is one that proves for every single local selection, that the greedy solution will be ahead (or equal to) any alternative solution. This is generally done akin to induction.

Consider the "cell towers" example from the lectures below:
For the base case of one tower, we place a tower 5km east of the first house encountered. If we place it any further east, it no longer covers the house, hence the base case holds true.

Suppose the claim is true for some $k - 1$, and consider the k-th tower. There is a house h such that:

$$g_{k-1} + 5 < h = g_k - 5$$

The alternatively paced towers, a_{k-1} and a_k are placed such that $a_{k-1} \leq g_{k-1}$, and hence, if $a_k > g_k$, the house h will not be covered.

B. Strongly connected components

Given a graph $G = (V, E)$ and a vertex v, the *strongly connected component* of G containing v consists of all vertices $u \in V$ such that there is a path in G from v to u and a path from u to v. We will denote it by C_v.

To find if u reaches v, construct another graph $G_{\text{rev}} = (V, E_{\text{rev}})$ consisting of the same set of vertices V, but with the set of edges E_{rev} obtained by reversing the direction of all edges E of G. If v can reach u in both G and G_{rev}, then u can also reach v.

1) Algorithm:

- BFS from the original graph G, to find $R_v \subseteq V$, where R denotes the reachable set of vertices.
- BFS from the reversed graph G_{rev}, to find $R'_v \subseteq V$.
- Hence, we have $O(V(V + E))$, with a BFS from each vertex.

However, a better time algorithm exists, using *Kosaraju's Algorithm* and *Tarjan's Algorithm*, which finds all strongly connected components in $O(V + E)$ time.

2) Condensation graph: The condensation graph is a graph that represents all of the strongly connected components as a set that connect to other strongly connected components. This may be useful for any problems that require the use of strongly connected components.

C. Topological sorting

Let $G = (V, E)$ be a directed graph, and let $n = |V|$. A *topological sort* of G is a linear ordering (enumeration) of it's vertices $\sigma : V \to \{1, \ldots, n\}$ such that if there exists an edge $(v, w) \in E$ then v precedes w in the ordering, i.e $\sigma(v) < \sigma(w)$.

A directed acyclic graph permits a topological sort of it's verticies. Topological sorts are not necessarily unique.

Can be computed in $O(V + E)$ time. Often useful to start with a topological sort, and then consider the problem for directed acyclic graph problems.

D. Djikstra's Algorithm

We know of the Djikstra's Algorithm from COMP2521. Let us prove some of the things that we took for granted in COMP2521. Note that d_v denotes the shortest path from s to v, where s is the source vertex. S represents the set of vertices where the shortest path has been found.

1) Why is it correct to always add the vertex outside S with the smallest d_v value?
2) When v is added to S, for which vertices z must we updated d_z, and how do we do these updates?
3) What data structure should be used to present the d_v values?
4) What is the time complexity of the algorithm?

1) Suppose there is some other shortest path from s to v, which leaves to some vertex y before reaching v. The alternate path, p', has a weight of atleast d_y. However, v was chosen to have the smallest d-value amongst vertices outside S. Therefore, we know that $d_v \leq d_y$, and hence, d_v is indeed the shortest path.

2) Consider the correct penultimate vertex v, and another alternative penultimate vertex, u, that both lead to z. Let us claim for contradiction that u leads to the shorter path. This means that within the shortest path from v to z, that it contains u.

   Since u was added before v was, we know that there is some shortest path from $s \rightarrow u$ that does not pass through v. Appending the edge from u to z produces a path through S from s to z which is no longer then p. Hence $p \geq d_v$, where d_v is the existing d_z value, and therefore, the changed d_z must have penultimate vertex v.

3) Since we need to delete and find the minimum d_v values, we should use a heap. However, we also need to update d_v values at times, we need to use an *augmented heap* that also has a lookup table for which index the values are contained at.

4) $O((n+m)\log n)$ with the augmented heap, but $O(m+n\log n)$ with a Fibonacci heap.

III. Flow networks

Flow networks are defined by directed graphs, where each edge $e = (u, v) \in E$ has a positive integer capacity $c(u, v) > 0$.

A flow in a network is a function $f : E \rightarrow [0, \infty), f(u, v) \geq 0$, which satifies:

1) A flow edge is less than or equal to the capacity edge
2) The outgoing flow is equal to the incoming flow

The value of a flow, is then defined as:

$$|f| = \sum_{(s,v)\in E} f(s, v) = \sum_{(v,t)\in E} f(v, t)$$

The *residual flow network* is a network that represents the leftover capacities for each flow edge - by representing it as the opposite directional edge. The two edges for each flow is:

1) an edge from v to w with capacity $c - f$, and
2) an edge from w to v with capacity f.

An **augmenting path** is a path in the residual flow network from s to t.

A. Single source and destination maximum flow

For flow networks, we generally want to find the maximum flow possible within the flow network. The *Ford-Fulkerson* algorithm works as such:

1) Keeping adding through new augmenting paths
2) When there are no more augmenting paths, we have achieved the largest possible flow.

Proof
The maximum flow of a flow network is found at a point where the flow over two partitions of the vertices V, is the same as the total capacity over those two partitions. We call these partitions *cuts*.

After the use of the Ford-Fulkerson Algorithm, partition the vertices into sets S and T, such that in S, all vertices are reachable from the source using the residual network, and where T are the remaining vertices. Hence, it follows that:

$$(s, t) : s \in S, t \in T : c(s, t) = f(s, t)$$

as $f(t, s) = 0$. Thus, at this point, the flow is maximal, and the cut is a minimal cut - the value of the flow is the maximum flow of the network.

Efficiency
The efficiency of the algorithm can be described by $O(E|f|)$, as each edge could take the minimum increment 1, and then we would have to apply DFS for every single augmenting path from $1 \ldots |f|$.

This can be improved by using *Edmonds-Karp* algorithm, which extends the Ford-Fulkerson through the taking of the shortest paths possible. Then, the number of augmenting paths is $O(VE)$, and since each takes $O(E)$ to find, the time complexity is $O(VE^2)$. Note that the original bound still applies, and either one could be stricter.

B. Problem-solving with flow networks

To solve problems with flow networks, we must first learn to declare flow networks. Flow networks must be explicitly defined with vertices, sinks, sources and capacities. You can use a dotpoint style format.

If you see a time complexity that looks like $O(nm^2)$, then it's likely a flow network question.

C. Multiple sources and sinks

To deal with multiple sources and sinks, create a "super sink" and/or a "super source", that connects all sinks and/or sources to a single sink/source.

IV. Dynamic Programming

Dynamic programming is a problem solving paradigm that utilises *overlapping* subproblems. There are four main types that are present in COMP3121, which are:

- One parameter, constant recurrence
- One parameter, linear recurrence
- Two parameter, constant recurrence
- Two parameter, linear recurrence

However, three parameter questions can certainly be asked. Here are some examples for each problem type.

A. Shortest path in a directed acyclic graph

Due to topological ordering of DAG's, we are able to create a dynamic programming solution to shortest paths.
For all $t \in V$, let $P(t)$ be the problem of determining opt(t), the length of the shortest path from s to t.

$$\text{opt}(t) = \min\{\text{opt}(v) + w(v,t) | (v,t) \in E\}$$

We improve from using Djikstra's ($O(n \log n)$) to getting $O(n+m)$ performance.

B. Bellman-Ford algorithm

The Bellman-Ford algorithm finds the shortest path to all vertices from a single source. The algorithm requires $|V|-1$ iterations where each iteration consistently checks each edge, and checks if any paths can be shortened.
For all $0 \leq i \leq n-1$ and all $t \in V$, let $P(i,t)$ be the problem of determining opt(i,t), the length of a shortest path from $s \to t$ which contains at most i edges.

$$\text{opt}(i,t) = \min\{\text{opt}(i-1,v) + w(v,t) | (v,t) \in E\}$$

The theory is that at most, the shortest path must use $n-1$ edges, and hence, the answer is located at the list of values opt$(n-1,t)$. The overall time complexity is $O(nm)$. The shortest paths can of course, be reoconstructed by storing steps during the algorithm, and then using argmin notation.

C. Floyd-Warshall Algorithm

The Floyd-Warshall Algorithm finds the shortest path between *every single pair*.
For all $1 \leq i,j \leq n$ and $0 \leq k \leq n$, let $P(i,j,k)$ be the problem of determining opt(i,j,k), the weight of a shortest path from $v_i \to v_j$ using only $v_1, \ldots, v_k$ as intermediate vertices.

$$\text{opt}(i,j,k) = \min(\text{opt}(i,j,k-1), \text{opt}(i,k,k-1) + \text{opt}(k,j,k-1))$$

Base cases exist where if $i = j$ then 0, if an edge exists, then the edge weight, and if no edge exists, then ∞. The Floyd-Warshall algorithm takes $O(n^3)$ time.

D. Rabin-Karp Algorithm

The Rabin-Karp algorithm is a string-matching algorithm that utilises "polynomial rolling hashing" to be able to find substrings of equal hash value - and then compare them character by character. The hashing function is as follows:

$$h(B) = d^{m-1} \cdot b_1 + d^{m-2} \cdot b_2 + \ldots + b_m$$

where B is a string, d is the length of the alphabet and b_i are the characters of the string B. We can use Horner's rule to create repetitiveness. Indeed, there are overlapping terms for two strings, A_s and A_{s+1}, such that:

$$H(A_{s+1}) = d \cdot H(A_s) - d^m a_s + a_{s+m}$$

From this, we can calculate precompute the hash for the comparison string B in $O(m)$ - and at the worst case, every string in A matches to the comparison hash, which leads to $O(mn)$ performance.

E. Knuth-Morris-Pratt Algorithm

The Knuth-Morris-Pratt (KMP) algorithm utilises a notion of a prefix-suffix to compare two strings in linear time. For example, for some pattern `xyxyxzx`, let us say we get to some pattern `xyxyxy`. We don't like this, as we were expecting a `z`. However, this doesn't mean we have to restart from the second letter of the comparator. We can simply identify that `xyxy` is a prefix-suffix in the comparator string that is contains in our pattern.

This notion of a prefix-suffix is controlled by the *failure function*, defined by $\pi(k)$, which is the length of the longest prefix-suffix for a string B_k. Thus, if we get to some width of the string w, we know that if we face a unmatched character, we can reduce to $\pi(w)$, to find the best partial match we can.

Consider the following algorithm.

- Maintain pointers $l = 1$ and $r = 0$, which record the left and right boundaries of our current partial match.
- We'll use $w = r - l + 1$ as shorthand for the length of our partial match.
- Compare the next character a_{r+1} to b_{w+1}. If they agree, we can extend the partial match.
- Otherwise, try to extend from $\pi(w) \to \pi(w) + 1$, by increasing l by the appropiate amount, and repeat previous step.
- If the characters ever agree, increase r by one.
- If a match of length 0 can't be extended, increase both l and r by one and move on

Two pointer $O(n+m)$ solution, where $O(m)$ is to precompute the failure function values.

Let $\pi(k)$ be the length of the longest prefix-suffix of B_k. The following recurrence is used to compute the failure function:

$$\pi(k+1) = \begin{cases} \pi(k)+1 & \text{if } b_{k+1} = b_{\pi(k)+1} \\ \pi(\pi(k))+1 & \text{if } b_{k+1} = b_{\pi(\pi(k))+1} \\ \dots \end{cases}$$

The base case is $\pi(1) = 0$. The algorithm works by finding matching characters, and when an unmatching character is found - by iterating backwards to find a smaller matching segment (a smaller prefix-suffix).

F. KMP represented as a finite automaton

What is a finite automaton? Hell if I know. But in general, we have some "transition" function, which shows the different changes in state of our "search" that may occur at each point of our search. For example, consider the below diagram:

k	matched	x	y	z
0		**1**	0	0
1	x	1	**2**	0
2	xy	**3**	0	0
3	xyx	1	**4**	0
4	xyxy	**5**	0	0
5	xyxyx	1	4	**6**
6	xyxyxz	**7**	0	0
7	xyxyxzx	1	2	0

Here, at each stage, the corresponding number in the alphabet shows if that letter is encountered next, what state the matching would go into next. For example, at state $k = 5$, if we saw a x again, we'd have to start all the way from state 1, as we can't reuse any of our letters. Thus, this concept is highly linked to our failure function.

V. Intractable problems

Intractable problems are problems that are very hard to solve.

A. Analysing time complexity

Time complexity should be calculated with respects to the length of the input. This means that things that looked polynomial before (such as Knapsack), are actually non-deterministic.

We can define an integer x to take up $\log_2(x)$ amount of space using binary representation. Similarly, we can say that a graph takes up $E \cdot \log_2(W)$ space, where W is the largest weight of an edge. An n length array, could then have an input length of $n \log_2(M)$, where M is the largest value of the array.

Consider the Knapsack problem. We have the time complexity $O(nC)$. Our length of input is $n \log_2(C)$, but $C = 2^{\log_2(C)}$, therefore, we are actually non-deterministic.

B. Classifications

- **P(olynomial)**
  A decision problem $A(x)$ is in class **P** if there exists a polynomial time algorithm which solves it.
- **N(on-deterministic) P(olynomial)**
  A decision problem $A(x)$ is in class NP if there is a problem $B(x, y)$ such that:
  1) For every input x, $A(x)$ is true if and only if there is some y for which $B(x, y)$ is true, and
  2) The truth of $B(x, y)$ can be verified by an algorithm running in polynomial time in the length of x only
  3) We call B the certifier, and y the certificate.
- **NP-hard**
  A decision problem V is NP-*hard* if every other NP problem is reducible to V. This means that NP-*hard* problems are at least as hard as everything in NP.
- **NP-complete**
  A decision problem is NP-*complete* if it is in class **NP** and class **NP-H**. NP-complete problems are in a sense universal. If we had an algorithm which solves any problem V, then we could also solve every other NP problem U by reduction.

C. Polynomial reductions

Polynomial reductions is a problem-solving method in intractable problems, by reducing a problem into a similar situation. Importantly, if there is a polynomial reduction from U to V, we can conclude that U is no harder than V.

A reduction does not need to be *surjective*, that is, we might only map to specific kinds of instances of V. A useful result is to use the contrapositive. If U can be reduced to V, then if you could solve problem V in polynomial time, the problem U would also have a polynomial time solution. The contrapositive is also true, if there is no known polynomial time algorithm for U, then there also can't be a polynomial time algorithm for V.

1) Cook's Theorem: Every NP problem is polynomially reducible to the SAT problem.

To show that a transformation from problem $U \rightarrow V$ $f(x)$ is a polynomial reduction, you must generally prove:

1) If x is YES, then $f(x)$ is YES.
2) If $f(x)$ is YES, then it maps from a YES x instance (or that if x is NO, then $f(x)$ is also NO).
3) That the reduction can be performed in polynomial time.

D. Solving reductions

Many problems are prefaced with an *at most* or an *at least* - if this is the case, just try to prove the extreme equality case. For example, if the question wants you to prove the cases ≤ 4000, then try to just prove $= 4000$. in terms of a decision problem, this is still true.

E. Linear Programming

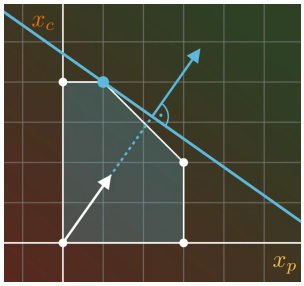
Linear programming is getting a set of variables, their linear constraints, and some linear *objective function*, and then solving an optimisation problem based on them.

F. How it works through an example

A farmer has 3 tons of potato seeds, 4 tons of carrot seeds and 5 tons of fertiliser (used in a $1:1$ ratio). The profit is \$1.2/kg for potatoes, and \$1.7/kg for carrots. Then, we can set up our problem like this:

- Declare two variables, p and c, for the amount of potatoes and carrots we plant.
- We have some constraints, namely:
  - $p, c \geq 0$
  - $p \leq 3000$
  - $c \leq 4000$
  - $p + c \leq 5000$
- We want to declare our objective, which is to maximise our profit. Hence, this becomes: $1.2p + 1.7c$

We first visualise all the constraints onto a plane. We then find the convex polygon that is the intersection of all of the inequalities. From here, distinct corners are found. To find the optimal solution, we travel in the direction of the objective function until we find the last point.



G. Integers vs Real Numbers

Real numbers create nice convex polygons, that we can utilise to find the ”points” shown above. A lot of the time though, we need to find only integer solutions. This actually becomes *incractable*.

The biggest issue with integer linear programming comes to the fact that it no longer follows the ”corner” method that we saw before, since we cannot guarantee that the polygons form corners at integer values. Hence, with this integer restrictions, comes an egregious amount of new points to test, such that it becomes intractable.

H. Super-polynomial algorithms

Even if our problems are intractable, we can create exponential tiem algorithms and try to make them more efficient. Many of these problems include finding *subsets*, and then filtering through some form of objects, and hence, and often seen time complexiy is $O(2^n n^k)$.

For example, subset sum takes $O(2^{\frac{n}{2}} n)$ and the Travelling Salesman problem takes $O(2^n m)$, due to the way they consider every single subset.

Developing algorithms for these problems is the same as any other - and we *very often* use dynamic programming to make things more efficient.