Design and Analysis of Algorithms

2018A7PS0193P

February 27, 2021

1 Fundamentals

Definition 1.1. An algorithm is a well defined computational procedure. It takes an input, does some computation and terminates with output

To check the correctness of an algorithm, we must check the following characteristics:

- Initialization: The algorithm is correct at the beginning
- Maintenance : The algorithm remains correct as it runs
- Termination : The algorithm terminates in finite time, correctly

For this entire course, we must always prove these characteristics when defining any algorithm.

Algorithms are generally defined by a complexity - the time taken to complete the computation on a given input size. There are three ways we could consider this - best case, worst case, or average case.

Complexity is discussed a lot in DSA, so I'm not going to rewrite it here. A quick roundup is:

- $O(g(n)) = \{f(n) : \text{there exists } c, n_0 : 0 \leq f(n) \leq c \cdot g(n) \forall n \geq n_0\}$
- $\Omega(g(n)) = \{f(n) : \text{there exists } c, n_0 : 0 \leq c \cdot g(n) \leq f(n) \forall n \geq n_0\}$
- $\Theta(g(n)) = \{f(n) : \text{there exists } c, n_0 : 0 \leq c_2 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \forall n \geq n_0\}$

To find complexities in the case of recurrences, we use the **master method**. Let the recurrence be given by:

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

Here, $a, b \geq 1$. Let ϵ be a constant. Then:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ then $T(n) = \Theta(n^{\log_b a})$
2. If $f(n) = O(n^{\log_b a})$ then $T(n) = \Theta(n^{\log_b a} \log n)$
3. If $f(n) = O(n^{\log_b a + \epsilon})$ then $T(n) = \Theta(f(n))$ provided if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n.

Here, we redo DSA despite it being a prerequisite of the course. This recap has lasted 3 lectures (so far). You should probably just read CLRS, this is a waste. The topics covered are:

- Quicksort (and it's average case analysis)
- The $\Omega(n \log n)$ lower bound of comparison sorting
- Non-comparison sorting like counting sort, radix sort, etc.
- Average case analysis of bucket sort

2 Matrix Multiplication

Naive Matrix multiplication is $\Theta(N^3)$, since we can express the result $A \cdot B = C$ as:

$$C_{ij} = \sum_{k=1}^{r} A_{ik} \times B_{kj}$$

We can improve this using a divide-and-conquer approach with **Strassen's Multiplication**. It has four steps:

1. Divide the input matrices A and B and the output matrix C into four $n/2 \times n/2$ submatrices. This takes $\Theta(1)$ time.
2. Create 10 matrices $S_1, S_2, ...S_10$, each of which is of size $n/2 \times n/2$ and is the sum or difference of two matrices created in step 1.
3. Using these submatrices, we can recursively compute seven matrix products $P_1, P_2, ...P_7$, each of which is $n/2$.

4. Compute the desired submatrices $C_{11}, C_{12}, C_{21}, C_{22}$ by adding and subtracting various combinations of the P_i matrices. We can compute all four in $\Theta(N^2)$ time.

The details of this can be seen on page 80 of CLRS, but the running time recurrence will be given by:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 7T(n/2) + \Theta(n^2) & \text{if } n > 1 \end{cases}$$

By master method, this is $T(n) = \Theta(n^{\log 7})$

3 Polynomial Multiplication

Polynomials are functions of the form:

$$f(x) = a_0 + a_1 x + a_x^2 + ...a_{n-1} x^{n-1}$$

One way to express this is as a vector of coefficients - this is called the **Coefficient form**. This form also allows us to evaluate $f(x)$ in $O(n)$ using Horner's rule, where we express the polynomial as:

$$a_0 + x(a_1 + x(a_2 + ...x(a_n - 1)...))$$

Another way to express this is using the **point value form** , where we express it as n point of the form $(x_i, f(x_i))$. This point value form uniquely identifies a polynomial. Generally this would take $\Theta(N^2)$ time , but with FFT we can do it in $O(N \log N)$.

The process of getting the coefficient form from the point value form is known as **interpolation**. We can do this in $O(n^3)$ using Gaussian Elimination, or in $O(n^2)$ with Lagrange Interpolation.

Generally, the multiplication of polynomials takes $\Theta(N^2)$. However, we can do this much faster using **Fast Fourier Transform**.

3.1 Fourier Transform

3.1.1 Discrete Fourier Transform

From now on, w_n^k will denote the k^{th} solution of $x^n = 1$, i.e. the n^{th} root of unity. w_n will denote the principal n^{th} root of unity. Remember the following properties:

1. $w_n^k = e^{\frac{2k\pi i}{n}}$
2. $w_{dn}^{dk} = w_n^k$
3. $w_n^{n/2} = w_2 = -1$
4. If $n > 0$ is even, then the squares of the n complex n^{th} roots of unity are the $n/2$ complex $n/2$th roots of unity. (Halving Lemma)
5. $\sum_{j=0}^{n-1} (w_n^k)^j = 0$ (Summation Property)

We call the vector $y = (y_0, y_1, ...y_{n-1})$ the **discrete Fourier Transform** of the polynomial A if $y_k = A(w_n^k)$.

3.1.2 Fast Fourier Transform

FFT consists of three parts:

1. Evaluation, where we find the DFT in $O(n \log n)$
2. Pointwise Multiplication of the two DFTs
3. Interpolation or the inverse FFT, where we find the coefficient form in $O(n \log n)$.

Consider the following polynomials:

$$A(x) = a_0x^0 + a_1x^1 + ... + a_{n-1}x^{n-1}$$
$$A_0(x) = a_0x^0 + a_2x^1 + ... + a_{n-2}x^{n/2-1}$$
$$A_1(x) = a_1x^0 + a_3x^1 + ... + a_{n-1}x^{n/2-1}$$

It is easy to see that:

$$A(x) = A_0(x^2) + xA_1(x^2)$$

These polynomials have only half as many coefficients as the polynomial A. So, if we can compute $DFT(A)$ from $DFT(A_1)$ and $DFT(A_0)$ in linear time, we would be able to do this in $O(n \log n)$ (direct from master method).

We find that do this with the equations:

$$y_k = y_k^0 + w_n^k y_k^1$$
$$y_{k+n/2} = y_k^0 - w_n^k y_k^1$$

Here, $k \in [0, n/2 - 1]$. The proof of this can be seen on cp-algorithms.

As such, we have found the DFT of the polynomial in $O(n \log n)$ time.

After performing the pointwise multiplication of the DFT of our polynomials A and B, we have to interpolate to find the coefficient form of our resulting polynomial C.

TODO: Add interpolation with Vandermonde matrix, can be seen on cp-algorithms

4 Greedy algorithms

Greedy algorithms involve making a sequence of choices where each looks best at the moment. It is making the locally optimal choice in the hope that it leads to a globally optimal solution. However, this may not always be the case. For this to work, we need the following properties:

- **Greedy choice property:** we can assemble a globally optimal solution by making locally optimal (greedy) choices.
- **Optimal Substructure** : A problem is said to have optimal substructure if an optimal solution to the problem contains within it optimal solutions to subproblems.

4.1 Activity Selection Problem

Consider a set $S = \{1, 2, 3, ..n\}$ of n activities that can happen one activity at a time. Activity i takes place during interval $[s_i, f_i)$. Activities i and j are compatible if $[s_i, f_i)$ and $[s_j, f_j)$ don't overlap. Our goal is to select the maximum size subset of mutually comparable activities.

To solve, we can assume that activities are in increasing order of their finishing time. If not, then sort it in $O(n \log n)$. Doing this, we can choose them greedily, picking an activity whenever we are free.

4.2 Fractional Knapsack Problem

A thief robbing a store finds n items. The i^{th} item is worth v_i dollars and weighs w_i pounds. The thief wants to take as valuable a load as possible, but he can carry at most W pounds in his knapsack. Which items should he take?

If the thief can carry fractions of items, he can solve it greedily - this is called the fractional knapsack problem. Otherwise, if he can either take or leave an item (the 0-1 knapsack problem), then it needs to be solved by dynamic programming.

In the fractional knapsack problem, we can greedily choose the items with the largest value to weight ratio.

4.3 Huffman Coding

Huffman coding is a greedy algorithm that constructs an optimal prefix code.

Say we are given a text, along with the frequencies of each character in the text. Obviously, we want the most frequent character to take up the minimum number of bits, to minimize the total size. We can make the same arguments for the other characters in decreasing order of frequencies. For instance, if the character 'a' has the most frequency, we may represent it by the bit 0, and the character 'x' (which is next in the order of frequency) as 10 and so on. We have to design these so that there is no ambiguity when decoding the Huffman code. This means no code can be the prefix of any other code!

We can represent this encoding as a binary tree where going left corresponds to adding the character '0' to the code, and moving right corresponds to adding the character '1' to the code. Each character is a leaf in this binary tree, and they will definitely not be prefixes of one another.

The algorithm for Huffman Coding creates this tree. Say we are given a set C of characters, along with theire frequencies. The algorithm is as follows:

Algorithm 1: Huffman Coding

```
Result: A prefix tree for Huffman Codes
$n = |C|$
$Q = C$
for $i \leftarrow 1$ to $n - 1$ do
    Allocate new node $z$
    $z.left = x =$ EXTRACT_MIN(Q)
    $z.right = y =$ EXTRACT_MIN(Q)
    $z.freq = x.freq + y.freq$
    INSERT($Q.z$)
end
return EXTRACT_MIN(Q)
```

If we use a heap, we can do this in $O(n \log n)$. It can actually be faster using van Emde Boas Tree, which would make it $O(n \log \log n)$

5 Matroids

A **matroid** is an ordered pair $M = (S, I)$ satisfying the following conditions:

- S is a finite set
- I is a non empty family of subsets of S called the independent subsets of S, such that if $B \in I$ and $A \subseteq B$, then $A \in I$. This is the Hereditary Property.
- If $A \in I$, $B \in I$, and $|A| \leq |B|$ then there exists some element $x \in B - A$ such that $A \cup \{x\} \in I$. This is the exchange property.

The graphic matroid $M_G = (S_G, I_G)$ is defined as follows:

- The set S_G is the set of edges in the graph G
- If A is a subset of E (edges), then $A \in I_G$ if and only if A is acyclic. That is, a set of edges A is independent if and only if the subgraph $G_A = (V, A)$ forms a forest

Theorem 5.1. If $G = (V, E)$ is an undirected graph, then $M_G = (S_G, I_G)$ is a matroid.

Theorem 5.2. All maximal independent subsets in a matroid have the same size.

Given a matroid M, we call an element $x \notin A$ an **extension** of $A \in I$ if we can add x to A while preserving its independence, i.e. $A \cup \{x\} \in I$.

A matroid $M = (S, I)$ is said to be weighted if it is associated with weight function $w(x)$ for all $x \in S$. $w(A)$ is defined as :

$$w(A) = \sum_{x \in A} w(x)$$

The independent set with maximum $w(A)$ is called an optimal subset of a matroid. An optimal subset is always a maximal independent subset.

Let us consider the Minimum Spanning Tree problem, where we seek the subset of edges that connects all the vertices together and has minimum total length. This is like finding the optimal subset of a weighted matroid M_G where weight function $w'(e) = w_0 - w(e)$, where $w(e)$ is the weight of the edge and w_0 is some constant greater than all the weights. A greedy algorithm for a weighted matroid is:

Algorithm 2: Greedy(M,w)

```
$A = \phi$
sort $M.S$ into monotonically decreasing order of weight $w$
for each $x \in M.S$ do
    if $A \cup \{x\} \in M.I$ then
        $A = A \cup \{x\}$
    end
end
return A
```

Lemma 5.3 (Greedy Choice Property)**.** Consider $M = (S, I)$ with weight function w. Let S be sorted in decreasing order. Consider x, the first element of S such that $\{x\}$ is independent. If this exists then there exists an optimal subset A containing x.

Lemma 5.4. Let $M = (S, I)$ be any matroid. If x is an element of S that is an extension of some independent subset A of S, then x is also an extension of ϕ.

Corollary 5.4.1. Let $M = (S, I)$ be any matroid. If x is an element of S such that x is not an extension of ϕ, then x is not an extension of any independent set A of S.

Lemma 5.5 (Optimal substructure property)**.** Let x be the first element of S chosen by GREEDY for the weighted matroid $M = (S, I)$. We can reduce this problem to $M' = (S', I')$, such that:

- $s' = \{y \in S : \{x, y\} \in I\}$

- $I' = \{B \subseteq S - \{x\} : B \cup \{x\} \in I\}$

6 0-1 Knapsack

No notes for this, it's too simple. $O(N * W)$ algorithm. For general information, there are lots of faster algorithms if you add some extra constraints.

7 Travelling Salesman Problem

Consider we have a graph, where every edge between vertices i and j has some weight c_{ij}. Our goal is to find a path where we start from one city, visit every other city and return to the same one again, in the cheapest manner. This is like finding a Hamiltonian Cycle in the graph.

Let $g(i, S)$ be the length of the shortest path starting at vertex i, going through all the vertices in S and terminating at 1. Then the following equations are obvious:

$$g(1, V - \{1\}) = \min_{2 \leq k \leq n} \{c_{1k} + g(k, V - \{1, k\})\}$$
$$g(i, S) = \min_{j \in S} \{c_{ij} + g(j, S - \{j\})\}$$

From this, we can design the TSP algorithm:

Algorithm 3: TSP(V, c_{ij})

```
for $i = 2$ to $n$ do
    $g(i, \phi) = c_{j1}$
end
for $k = 1$ to $n - 2$ do
    for $i = 2$ to $n$ do
        for $S \subseteq V - \{i, 1\}$ with $|S| = k$ do
            $g(i, S) = \min_{j \in S}\{c_{ij} + g(j, S - \{j\})\}$
        end
    end
end
$g(1, V - \{1\}) = \min_{j \in S}\{c_{1i} + g(i, V - \{1, i\})\}$
return $g(1, V - \{1\})$
```

The time complexity of this is $T(n) = \Theta(n^2 \cdot 2^n)$ and space complexity $\Theta(n 2^n)$.

8 Matrix Chain Multiplication

If we are given a sequence of matrices, A, B, C of size $u \times v$, $v \times w$, $w \times z$ respectively. This gives us two ways to multiple the matrix :

- $(A \times B) \times C$: Takes $u \times v \times w + u \times w \times z$ steps
- $A \times (B \times C)$: Takes $u \times v \times z + v \times w \times z$ steps

Our goal is to find the order of multiplication that would take the minimum number of steps.

One way to do this could be brute force, where we try every order of multiplication. This problem is equivalent to finding the number of ways to parenthesize an expression of n matrices. This can be expression by the recursion:

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P(k) \times P(n-k) & \text{otherwise} \end{cases}$$

This is, in fact, the $n - 1$ Catalan number $C(n - 1)$, where:

$$C(n) = \frac{1}{n+1}\binom{2n}{n}$$

A more efficient approach to solve this is DP. Let us assume every matrix A_i has the dimensions $p_{i-1} \times p_i$. Then we can use the following DP:

Algorithm 4: Matrix-Chain-Order(p)

```
$n$ = length[$p$] - 1
for $i = 1$ to $n$ do
    $m[i][i] = 0$
end
for $l = 2$ to $n$ do
    for $i=1$ to $n-l+1$ do
        $j = i + l - 1$
        $m[i][j] = \infty$
        for $k=i$ to $j-1$ do
            $q = m[i][k] + m[k+1[j] + p_{i-1}p_kp_j$
            if $q < m[i][j]$ then
                $m[i][j] = q$
                $s[i][j] = k$
            end
        end
    end
end
```

This is a standard DP by length. The idea is, if we are given a range of matrices $[l..r]$, the best parenthesization will always involve splitting the range into two parts - a prefix and a suffix, and multiplying it. So, for every range, we store the best achievable cost for every subrange of a length l, and to find the cost of a current subrange, try every prefix to find the minimum cost.

9 Longest Common Subsequence

This is also very common and standard, read GFG or something. Interesting fact is that the longest palindromic subsequence in a string is the LCS of the string and it's reverse.

10 Optimal Binary Search Trees

Suppose that we are designing a program to translate text from English to French. For each occurrence of English word in the text, we need to look up it's French equivalent. This can be done using a binary tree, and could ensure $O(\log n)$ time. However, words occur at different frequencies, so there could be a different total cost of search given a text. So, we want a optimal binary search tree.

Formally, we are given n keys $K = k_1, k_2, ..., k_n$ in sorted order and wish to build a BST on these keys. For each k_i, we have a p_i probability that the search will be for k_i. Some searches may be for values not in K, so we also have $n+1$ dummy keys $d_0, d_1, ...d_n$. In particular, d_i represents values between k_i and k_{i+1}. Each of these have a probability q_i. Of course,

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$$

The expected cost in a tree T is given by:

$$E[T] = \sum_{i=1}^{n} (depth_T(k_i) + 1) \times p_i + \sum_{i=0}^{n} (depth_T(d_i) + 1) \times q_i$$

Any non-leaf subtree of our BST must contain keys in a continuous range $k_i \cdots k_j$. Each subtree must be optimal. Let $e[i,j]$ be the expected cost for an optimal BST of keys $k_i, \cdots, k_j$, and let $w[i,j]$ be such that:

$$w[i,j] = \sum_{v=i}^{j} p_v + \sum_{v=i-1}^{j} q_v$$

Then if k_r is root,

$$\begin{aligned} e[i,j] &= p_r + e[i,r-1] + w[i,r-1] + e[r+1,j] + w[r+1,j] \\ &= e[i,r-1] + e[r+1,j] + w[i,j] \end{aligned}$$

Hence our goal becomes choosing r such that it minimizes $e[i,j]$. This gives us the following algo

Algorithm 5: Optimal-BST(p,q,n)

```
for $i$ = 1 to n+1 do
    $e[i, i-1] = w[i, i-1] = q_i - 1$
end
for $l$ = 1 to $n$ do
    for $i$ = 1 to n-l+1 do
        $j = i - l + 1$
        $e[i, j] = \infty$
        $w[i, j] = w[i, j-1] + p_j + q_j$
        for $r$ = $i$ to $j$ do
            $t = e[i, r-1] + e[r+1, j] + w[i, j]$
            if $t < e[i, j]$ then
                $e[i, j] = t$
                $root[i, j] = r$
            end
        end
    end
end
```

11 Flow Shop Scheduling

Consider n jobs each having m tasks $T_{1i}, T_{2i}, ..., T_{mi}$ for $1 \leq i \leq n$ where T_{1i} can be executed on processor p_i only. A processor cannot execute two tasks at a time, and T_{2i} cannot be executed before T_{1i}.

Say that according to a schedule S, we have a job finish at $f_i(S)$. Then the finish time of the schedule is:

$$F(S) = \max_{1 \leq i \leq n} f_i(S)$$

Let us first solve for $m = 2$. Let $a_i = t_{1i}$ and $b_i = t_{2i}$.

12 Flows

A **flow network** is a directed graph $G = (V, E)$ such that:

- Every edge $(u, v) \in E$ has a non-negative capacity $c(u, v)$.
- There are two distinguished vertices, a source s and a sink t.

- For each vertex $v \in V$ there exists a path from s to v to t.
- Self loops are not allowed.
- No reverse edges
- If $(u, v) \notin E$, $c(u, v) = 0$
- The graph is connected.

The **flow** in a graph is a real valued function $f : V \times V \to \mathbb{R}$ such that:

- $\forall u, v \in V, 0 \le f(u, v) \le c(u, v)$ (Capacity constraint)
- $\forall y \in V - \{s, t\}, \sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v)$ (Flow conservation)

The **network flow** is defined as:

$$|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$$

Typically, no edge enters the source so $\sum_{v \in V} f(v, s) = 0$. However, this definition will be more important when discussing residual networks.

When solving problems with flow, we need to design a flow network. Doing this is at times trivial (see e.g. in CLRS) and at times not (see any Codeforces flow problem). For one, sometimes it may be natural to have antiparallel edges in a graph while modelling - however this is not allowed in flow networks. Remove it by breaking any one of the edges (u, v) into two - $(u, v'), (v', v)$. Both of these new edges will have the same capacity.

Another issue is when handling a network that could have multiple sources or multiple sinks. In that case, we create supersources and supersinks, nodes which have edges with infinite capacity to other sources or from other sinks.

The **maximum flow problem** requires us to find a flow of maximum value in the graph. Before looking at the algorithm for this, we need to look at some preliminary concepts.

The **residual graph** G_f represents the flow of f on G as well as how we can change this flow. The edges of G that are in G_f are those that can admit more flow, and have a residual capacity given by:

$$c_f(u, v) = c(u, v) - f(u, v)$$

The residual network also has extra edges. In order to represent a possible decrease in the flow $f(u, v)$ we place an edge (v, u) in G_f with residual capacity $c_f(v, u) = f(u, v)$. It will admit flow in the opposite direction to (u, v), allowing the flow on an edge to be decreased.

If f is a flow in G and f' is a flow in the corresponding residual network G_f, we define $f \uparrow f'$ to be the **augmentation of flow** f by f', given by:

$$(f \uparrow f')(u,v) = \begin{cases} f(u,v) + f'(u,v) - f'(v,u) & (u,v) \in E \\ 0 & \text{otherwise} \end{cases}$$

We can prove that $|f \uparrow f'| = |f| + |f'|$.

An **augmenting path** is a simple path from s to t in the residual network G_f. We may increase the flow on an edge (u,v) of an augmenting path by up to $c_f(u,v)$. Hence, the amount by which we can increase the flow of each edge in an augmenting path p, called the residual capacity of p, is the minimum capacity $c_f(u,v)$ in the augmenting path. If we augment the flow f by this amount, we will get another flow, whose value is closer to the maximum.

Now we can define the algorithm to find the maximum flow in a path, called the **Ford Fulkerson Method** .

1. Initialize the flow f to 0
2. While an augmenting path p exists in the residual network G_f, augment the flow f along that path p.

The Ford Fulkerson method in fact refers to a class of algorithms, which have the same basic approach but find the augmenting path in different ways. It's complexity is $O(EF)$, where F is the maximal flow in the network. For more specifics, look into algorithms like Dinic, Edmonds-Karp, etc. Remember that is can be difficult to design a worst case for many flow algorithms.

A **cut** (S,T) of a graph is a partition of V into two disjoint sets S and T such that $s \in S$ and $t \in T$. The net flow across a cut is given by:

$$f(S,T) = \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{u \in S} \sum_{v \in T} f(v,u)$$

The capacity of the cut is:

$$c(S,T) = \sum_{u \in S} \sum_{v \in T} c(u,v)$$

The minimum cut of a network is the cut whose capacity is minimum across all cuts.

Theorem 12.1 (Max Flow Min Cut Theorem)**.** If f is a flow in a flow network $G = (V,E)$ with source s and sink t, then the following conditions are equivalent:

- f is a maximum flow in G
- The residual network G_f contains no augmenting paths
- $|f| = c(S, T)$ for some cut (S, T) in G