Design and Analysis of Algorithms

2018A7PS0193P

March 26, 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 Karatsuba Multiplication

Karatsuba multiplication is a divide and conquer method to speed up multiplication of large integers. Usually if multiplying two numbers x and y with n digits each takes $O(n^2)$ time, but Karatsuba multiplication does it in $O(n^{1.59})$.

Let us consider the strings in some base $B = 10$. Then, we can write

$$x = x_1 B^m + x_0$$

$$y = y_1 B^m + y_0$$

where $m = n/2$. The product will be given by:

$$xy = z_2 B^{2m} + z_1 B^m + z_0$$

where

$$z_2 = x_1 y_1$$

$$z_1 = x_1 y_0 + x_0 y_1$$

$$z_0 = x_0 y_0$$

This seems to need 4 multiplications, but it can in fact be done only in 3, by observing that:

$$z_1 = (x_1 + x_0)(y_1 + y_0) - z_2 - z_0$$

So, we get the algorithm:

Algorithm 1: KMul(x,y)

p = KMul($x_1 + x_0, y_1 + y_0$)
$x_1 y_1 = KMul(x_1, y_1)$
$x_0 y_0 = KMul(x_0, y_0)$
return $x_1 y_1 \times 10^n + (p - x_1 y_1 - x_0 y_0) \times 10^{n/2} + x_0 y_0$

The time complexity of this is $T(n) = 3T(n/2) + cn$, which gives the aforementioned complexity.

3 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 $T(n) = \Theta(n^{\log 7})$

4 Polynomial Multiplication

Polynomials are functions of the form:

$$f(x) = a_0 + a_1 x + a_2 x^2 + ...a_{n-1} x^{n-1}$$

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 finding the point value form from a coefficient form 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**.

4.1 Fourier Transform

4.1.1 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 $y_k = A(w_n^k)$.

In our case of polynomial multiplication, we will pad the polynomials with zero to the closest power of 2, such that it is at least double the size, and then compute DFT with this new size. This will help us since we need $2n$ point values to find the coefficient form of the product.

4.1.2 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 we can 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 first equation is clear, and comes directly from the formula. The proof for the second is as follows:

$$\begin{aligned} y_{k+n/2} &= A(w_n^{k+n/2}) \\ &= A_0(w_n^{2k+n}) + w_n^{k+n/2} A_1(w_n^{2k+n}) \\ &= A_0(w_n^{2k} w_n^n) + w_n^k w_n^{n/2} A_1(w_n^{2k} w_n^n) \\ &= A_0(w_n^{2k}) - w_n^k A_1(w_n^{2k}) \\ &= y_k^0 - w_n^k y_k^1 \end{aligned}$$

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

In short, the formula is:

$$a_k = \frac{1}{n} \sum_{j=0}^{n-1} y_j w_n^{-kj}$$

The coefficients can be found via the same divide and conquer algorithm as in the direct FFT, except we use w_n^{-k} instead of w_n^k.

I know I've done a poor job of explaining it, but I don't want to copy paste the entirety of the cp-algorithms post here. Check it out.

5 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.

5.1 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.

5.2 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.

5.3 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 2: 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 $O(n \log \log n)$

In a Huffman coding, the average bit length is given by:

$$\frac{\sum_{c \in A} |H_c| f_c}{\sum_{c \in S} f_c}$$

where A is the alphabet, H_c is the Huffman code for c, and f_c is the frequency of c.

6 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.

In more simple terms, the matroid gives a classification of each subset of S to be independent or dependent. The empty set is always independent and any subset of an independent set is independent. If an independent size A has smaller size than B, then there exists some element in B that can be added into A without loss of independency.

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. $A \cup \{x\} \in I$.

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

Theorem 6.2. All maximal independent subsets in a matroid have the same size.

This is obvious given that if a maximal independent subset had a smaller size, it could be extended using the exchange property.

A matroid $M = (S, I)$ is said to be **weighted** if it is associated with a strictly positive 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 3: 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

Notice, that this is basically Kruskal's algorithm for finding a minimum spanning tree. In graph theory terms, we are sorting all the edges in increasing order of edge weight, and choosing these edges one by one as long as they do not form a cycle (not in the independent set). The check for cycle can be done with a disjoint set union, in this case.

Lemma 6.3 (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 6.4. 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 6.4.1. 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.

These lemmas tell us that at any point, choosing the minimum is optimal, as long as it does not create a cycle.

Lemma 6.5 (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:

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

This lemma tells us that removing the edge x with minimum edge weight yields a new matroid for us to continue our greedy choices on.

From the above lemmas, we can be sure that our greedy solution is optimal.

7 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.

The transitions are:

$$M(i, w) = \max\{M(i-1, w), M(i-1, w-w_i) + p_i\}$$

8 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:

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

From this, we can design the TSP algorithm:

Algorithm 4: TSP(V, c_{ij})

for $i = 2$ *to* n **do**
  $g(i, \phi) = c_{i1}$
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 $\Theta(n2^n)$.

9 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 dimen-

sions $p_{i-1} \times p_i$. Then we can use the following DP:

Algorithm 5: 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] = ∞
        for k=i to j-1 do
            q = m[i][k] + m[k+1][j] + p_{i-1} p_k p_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. First we realize that in any range $A_{i..j}$ we can split the range between A_k and A_{k+1}, in such a way that the parenthesization of the prefix $A_{i..k}$ is optimal. This is because if there was a less costly way to parenthesize $A_{i..k}$, we could replace it with that and reduce the total cost. From this, we can split a range into two parts - a prefix and a suffix, and solve recursively on it. To combine two solutions, we would use the equation:

$$m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$$

By memoizing these values, we can generate this DP.

10 Longest Common Subsequence

This is very common and standard, so I'm only writing the transitions here.

$$LCS(X_i, Y_j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ LCS(X_{i-1}, Y_{j-1}) + 1 & x_i = y_j \\ \max\{LCS(X_i, Y_{j-1}), LCS(X_{i-1}, Y_j)\} & x_i \neq y_j \end{cases}$$

This only gives lengths, but the exact LCS can be found by moving backwards on the DP table.

Interesting fact is that the longest palindromic subsequence in a string is the LCS of the string and it's reverse.

11 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:

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

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

Any non-leaf subtree of our BST must contain keys in a continuous range $k_i \cdots k_j$. Each subtree must be optimal, since if it were not, then we could replace it with the more optimal version and create a better tree. This creates our subproblems to divide into and DP on. 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 algorithm

Algorithm 6: 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] = ∞
        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
```

12 Flow Shop Scheduling

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

This is essentially a scheduling problem. As is common in scheduling, we have two variants - preemptive and non-preemptive.

For a given schedule, $f_i(S)$ is the time taken to complete a job i. Then the finish time of a schedule S is given by:

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

Our goal is to get the optimal non-preemptive schedule, which would have minimum $F(S)$.

This is difficult to solve for $m > 2$, so let us solve for $m = 2$. Let us simplify the notation by using a_i for t_{1i} and b_i for t_{2i}. This can be represented by the matrix

$$\begin{bmatrix} a_1 & a_2 & a_3 & a_4 & a_5 \\ b_1 & b_2 & b_3 & b_4 & b_5 \end{bmatrix}$$

When $m = 2$, we find that there is nothing to gain by using different schedules for A and B. This is because before executing b_i, we need to execute a_i. So, after finishing the first task for A, we would want to immediately start the task for B, otherwise we would be wasting time.

Hence, we want a single optimal permutation. We find that in the optimal permutation, given the first job in the permutation, the remaining permutation is optimal. This is because if it were not, we could rearrange it to get the optimal permutation and only improve.

Let $g(S, t)$ be the length of the optimal schedule for the subset of jobs S under the assumption that the processor 2 is not available until time t. We want $g(\{1, 2, 3, \cdots, n\}, 0)$.

$$g(\{1, 2, 3, \cdots, n\}, 0) = \min_{1 \leq i \leq n} \{a_i + g(\{1, 2, 3, \cdots, n\} - \{i\}, b_i)\}$$

Here, we assume that $g(\phi, t) = t$ and $a_i \neq 0$.

Generalizing this,

$$g(S, t) = \min_{i \in S} \{a_i + g(S - \{i\}, b_i + \max\{t - a_i, 0\})\}$$

If i and j are the first two jobs of the schedule and B is not available for time t,

$$\begin{aligned} g(S, t) &= a_i + g(S - \{i\}, b_i + \max\{t - a_i, 0\}) \\ &= a_i + a_j + g(S - \{i, j\}, b_j + \max\{b_i + \max\{t - a_i, 0\} - a_j, 0\}) \end{aligned}$$

Now, let

$$\begin{aligned} t_{ij} &= b_j + \max\{b_i + \max\{t - a_i, 0\} - a_j, 0\} \\ &= b_j + b_i - a_j + \max\{\max\{t - a_i, 0\}, a_j - b_i\} \\ &= b_j + b_i - a_j + \max\{t - a_i, 0, a_j - b_i\} \\ &= b_j + b_i - a_j - a_i + \max\{t, a_i, a_j + a_i - b_i\} \end{aligned}$$

From the above discussion,

$$g(S,t) = a_i + a_j + g(S - \{i,j\}, t_{ij})$$

If i and j are interchanged, we get

$$g'(S,t) = a_j + a_i + g(S - \{j,i\}, t_{ji})$$

Hence, we can see that

$$g(S,t) < g'(S,t) \Leftrightarrow t_{ij} \leq t_{ji}$$

This should hold for all t, so after some reduction,

$$\min\{a_i, b_i\} \geq \min\{a_j, b_j\}$$

So, if $\min\{a_1, a_2, ..., a_n, b_1, b_2, ..., b_n\}$ is a_i, then i should be the first job. If $\min\{a_1, a_2, ..., a_n, b_1, b_2, ..., b_n\} = b_j$ then j should be the last job.

13 Flows

13.1 Introduction

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:

- $\forall u, v \in V, 0 \leq f(u, v) \leq c(u, v)$ (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.

13.2 Maximum Flow

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 13.1 (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

This means that the value of the maximum flow is equal to the capacity of the minimum cut.

13.3 Maximum Bipartite Matching

A **matching** is a subset of edges $M \subseteq E$ such that for all vertices $v \in V$, at most one edge in M is incident on v.

A matching is said to be **complete** when the number of edges is the same as the number of nodes in one side. A matching is **maximal** if it is impossible to add any more edges.

In our problem, we seek to find the **maximum matching**, i.e., the matching of maximum size in a bipartite graph.

To solve this, we can reduce it to a flow network. Define new nodes s and t. Add directed edges from s to all $u \in L$, and from all $v \in R$ to t. Here L, R are the two sets of the bipartite graph. This creates a new graph $G'(V', E')$, which will be our flow network. In this flow network, all the edges will have a capacity of 1.

If this is our flow network, then the cardinality of the maximal matching is equal to the max flow in the graph.

13.4 Edge Disjoint Paths

A set of paths is edge disjoint if their edge set is different. Our goal is to find the set of edge disjoint paths from s to t of maximum size.

In $G(V, E)$ with it's two distinguished nodes s and t define a flow network with capacity of each edge being 1. Then, the maximum flow in this network is the number of disjoint edge paths in the graph. The paths would actually be the augmenting paths in the graph.

It is important to mention **Menger's Theorem** - the maximum number of edge-disjoint $s - t$ paths equals the minimum number of edges whose removal separates s from t.

TODO: Cycles and undirected paths

14 The Complexity Class P

14.1 Prerequisites

Before we can talk about this complexity class, we need to revise some definitions.

Alphabet is a finite set of symbols. **Strings** are a concatenation of zero or more symbols from the alphabet. A set of string is called a **Language**.

Also see TOC to learn about Turing Machines. A Turing Machine can be deterministic (DTM) or non-deterministic (NTM).

14.2 Turing Machines and Time Complexity

DTMs encode some algorithm, and hence have a time complexity. Consider an example DTM that accepts the language $L = \{0, 1, 10, 11, ...\}$, i.e., the set of all binary strings. Such a DTM would have a $\Theta(n)$ complexity, to read the entire input. The time complexity of a DTM is independent of an input, and only depends on the size n.

14.3 Deterministic Time and the class P

A **complexity class** is a set of functions that can be computed with a given resource. In this course we will most pay attention to the classification of **decision problems**, which are Boolean functions that give a "yes" or "no" answer to a question. These can also be expressed as languages. A Boolean function f can be expressed as a language $L_f = \{x : f(x) = 1\}$. Let us now look at our first class.

Let $T : \mathbb{N} \to \mathbb{N}$ be some function. Then $\mathbf{DTIME}(T(N))$ is the set of all Boolean functions that are computable in $c \cdot T(n)$ time for some constant $c > 0$, i.e. $O(T(n))$. By "computable", we mean that there exists some DTM that is capable of recognizing the desired language L in $O(T(n))$ time.

The **DTIME** class serves as the base upon which we can define the class **P**. The class **P** is defined as:

$$\mathbf{P} =_{c \geq 1} \mathbf{DTIME}(n^c)$$

Now it's clear that **P** defines the class of problems that can be solved in polynomial time.

A simple example of a problem in P is the problem of multiplying two integers x and y of n bits each. We can multiply them in $O(n^2)$ time on a RAM Turing machine (a Turing Machine with RAM). A $T(n)$ time RAM TM can be simulated in $T(n^2)$ by a multitape DTM. Since this will not change the time complexity class **P** , we can prove that a given language is in **P** by checking if it has a polynomial time algorithm using a RAM Turing Machine.

The multiplication problem can be defined as a decision problem with the language $L_{mult} = \{(x, y, z) | x, y, z \text{ are binary integers such that } z = xy\}$. A RAM TM for accepting L_{mult} would calculate $z' = xy$ in time $O(n^2)$ and check whether $z' = z$ in time $O(n)$, allowing for a total time complexity of $O(n^2)$. Hence, $L_{mult} \in \mathbf{P}$.

An interesting property of **P** is that it is closed under complementation. We define the time complexity class $Co - P$ as the set such that $L \in Co - P$ if $\overline{L} \in P$. Notice that $Co - P$ is not actually the complement of P. We can prove that $Co - P = P$.

15 The Complexity Class NP

15.1 Non-Deterministic Turing Machine

Non-deterministic Turing Machines (NTM) are generalizations of DTM in which at each step, the Turing Machine may have more than one possible choice of moves. Since there is more than one applicable transitions at a state, the NTM will make a guess as to which transition to choose. Importantly, NTMs will always make the choice that would make it possible to accept the input. We assume it is capable of doing this without any lookahead of future moves.

We say that a given NTM M runs in $T(n)$ time if for every input $x \in L$, and every sequence of non deterministic choices, M reaches a halting state within $T(|x|)$ steps.

15.2 Non-deterministic Time and the class NP

For every function $T : \mathbb{N} \to \mathbb{N}$ and $L \subseteq \{0,1\}^*$ we say that $L \in \mathbf{NTIME}(T(n))$ if there is a constant c and a $cT(n)$ time NDTM M such that for every $x \in \{0,1\}^*, x \in L \Leftrightarrow M(x) = 1$.

Then the complexity class **NP** is defined as:

$$\mathbf{NP} = \bigcup_{c \in \mathbb{N}} \mathbf{NTIME}(n^c)$$

Obviously, $\mathbf{P} \subseteq \mathbf{NP}$, since a DTM is a special case of an NTM, and hence any DTM part of **P** will also be part of **NP**.

15.3 The Certificate Definition of NP

P can be views as a set of problems that can be solved efficiently. **NP** can be viewed as a set of problems that can be efficiently verified given a possible solution. This gives us an alternate definition of **NP**.

A language L is in **NP** is there exists a polynomial p and a polynomial time DTM M called the verifier for L such that for every $x \in \{0,1\}^*, x \in L \Leftrightarrow \exists u \in \{0,1\}^{p(|x|)}$ such that M accepts x, u. If M does accept x, u, then u is the **certificate** of x.

The meaning of this definition is that we are given an input x and a possible solution u, and we should be able to devise a DTM that can verify u in polynomial time. The restriction of polynomial time also restricts the length of u to a polynomial in $|x|$.

15.4 The Independent Set Problem

A well known NP problem is the independent set problem. A language corresponding to the decision version of the problem is:

INDSET = $\{(G, k)|G$ is the adjacency matrix of an undirected graph having a subset of at least K vertices having no edge between them $\}$

First let us show the polynomial time solution with an NTM:

1. On input $G_{n\times n,k}$, append a string of length n after the input by using the first non-deterministic choice as writing 0, and the second non deterministic choice as writing 1.

2. Deterministically verify that the vertices corresponding to 1 make an independent set of size at leat k.

If the input is in INDSET, then step 1 will correctly guess an independent set of at least k, and the NTM will verify it correctly and accept the input.

If it is not in INDSET, then every guess in steo 1 will not be able to make an independent set of size at least k. NTM will reject the input in step 2 after verifying it to be either not an independent set or an independent set of size less then k.

We can also verify it with the second definition and desigining a polynomial time DTM that takes certificates as input. First the DTM will verify that the size of $|u| = n$ and that the vertices whose values are set to 1 are an independent set of G of size at least $|k|$. The DTM obviously runs in polynomial time and $|u| = n$ is polynomially bounded.

16 The Class NP-Complete and NP-Hard

16.1 Polynomial Time Reductions

We say that a language L_1 reduces to a language L_2 in polynomial time if there exists a polynomial time computable function $f(x)$ such that $x \in L_1$ if and only if $f(x) \in L_2$. By polynomial time computable function, we mean that there exists a DTM that can perform this function in polynomial time. This is denoted by $L_1 \leq_p L_2$.

If $L_1 \leq_p L_2$ and L_2 has a polynomial time algorithm A_2, then we can combine A_2 and f to get a polynomial time A_1 for L_1. First we give x as an input to the DTM to compute $f(x)$ in polynomial time. Then $f(x)$ can be given to the DTM for A_2. If A_2 accepts $f(x)$, then A_1 accepts x. Else, it rejects x. The total time taken will obviously also be polynomial.

Polynomial reduction is transitive - if $L_1 \leq_p L_2$ and $L_2 \leq_p L_3$ then $L_1 \leq_p L_3$.

16.2 NP-Complete and NP-Hard

A language L_1 is NP-hard if $\forall L \in \mathbf{NP}, L \leq_p L_1$.

A language $L_1 \in \mathbf{NP}$ is NP-Complete if $\forall L \in \mathbf{NP}, L \leq_p L_1$.

Notice the differnece in these definitions - a language L is NP-Complete if L is NP-Hard and $L \in NP$. Hence, a NP-Complete language is always NP-Hard, but a NP-Hard language need not be NP-Complete.

A more informal way of putting these definitions is that any NP-hard language is at least as hard as any other NP language. We have the term NP-Complete because to solve the **P** vs **NP** problem, it suffices to study whether an NP-complete problem can be decided in polynomial time. This is true since any other NP problem could be reduced to the NP-complete problem in polynomial time, hence also becoming polynomial.

16.3 An NP-Complete Problem

A **Boolean formula** over the variables $u_1 \cdots u_n$ consists of these variables and the logical operators AND,OR and NOT. A boolean formula Φ is satisfiablef there exists some assignment of value to the variables such that Φ evalueates to true.

This problem, called the SAT problem, is NP-complete (called the Cook-Levin THeorem). To prove this, we must first prove that it is NP. Given a boolean formula $\Phi(z)$, an NTM will guess an assignment of variables for z and then it will evaluate $\Phi(z)$ in polynomial time. If it evaluates to 1, then it will accept, otherwise it will reject. Hence, SAT $\in$ NP.

The next step is to prove that it is NP-Hard. For this, we have to prove that for an $L \in NP, L \leq_p SAT$. We will have to describe a polynomial time DTM M such that $x \in L \Rightarrow M(x) \in SAT$ and $x \notin L \Rightarrow M(x) \notin SAT$. Since L is in NP, there exists a polynomial time DTM D such that if $x \in L$, there exists $y \in \{0,1\}^{p(|x|)}$ such that $D(x,y) = 1$. If $x \notin L$, then there does not exist any such y such that $D(x,y) = 0$.

17 Problems

Exercise. Tile a $n \times n$ chessboard with one tile missing using L shaped triominoes, where n is a power of 2.

Solution. This can be done using divide and conquer. In the case that $n = 2$, it is trivial to fill with the triomino. In any other case, we can divide the chessboard into 4 equal subsquares. We then place a L shaped tile such that it does not cover the subsquare containing the missing cell. Now the problem can be solved recursively, since each of the subsquares now have a missing square.

Exercise. Solve the recurrence $T(n) = T(n/3) + T(2n/3) + 1$.

Solution. We guess that $T(n) = O(n)$ and substitute accordingly. $T(n) \leq cn - d$. Then:

$$\begin{aligned} T(n) &\leq c(\frac{n}{3}) - d + c(\frac{2n}{3}) - d + 1 \\ &\leq cn - 2d + 1 \\ &\leq cn - d \end{aligned}$$

The above holds as long as $d \geq 1$.

As for the base case, if $T(1) = 1$, we can choose a suitably large value for c and d for it to be true.

Exercise. Solve the recurrence $T(n) = T(n/3) + T(2n/3) + cn$

Solution. At each level, we can see that the sum of the "work done" is cn, up till some point. After that, since the right subtree would be doing more work, it would go deeper than the left subtree.

First let us consider a lower bound. The lower bound on the height of the tree is obviously $\log_3 n$, using the leftmost path. So, considering a full binary tree with that height,

$$T(n) \geq n \log_3 n$$

Now to consider the deepest path, which would be $\log_{3/2} n$. This tells us that:

$$T(n) \leq n \log_{3/2} n$$

From this, we find that $T(n) = \Theta(n \log n)$.

Exercise. How do you use Strassen's algorithm when n is not a power of 2?

Solution. You can pad the matrices A and B with 0 until it is a power of 2. This won't affect the complexity since $N > 2n$, where N is the new dimension.

Exercise. How quickly can you multiply a $kn \times n$ matrix by a $n \times kn$ matrix, using Strassen's algorithm? What if the input is reversed?

Solution. If we consider the first case, we can see that the $kn \times n$ matrix is composed of k $n \times n$ matrices. We can say the same for $n \times kn$. Therefore, we can write both matrices as $k \times 1$ and $1 \times k$, by assuming each $n \times n$ matrix as a single element. So the running time will be:

$$T(kn \times n, n \times kn) = k^2 T(n \times n, n \times n)$$

Using Strassen's algorithm,

$$T(kn \times n, n \times kn) = k^2 n^{\log 7}$$

In the second case, we find that using the same arguments and multiplying a $1 \times k$ and a $k \times 1$ matrix, we would also need to do k multiplications and $k - 1$ additions. Therefore,

$$T(n \times kn, kn \times n) = kT(n \times n) + O(k)$$

Which equals

$$T(n \times kn) = kn^{\log 7}$$

Exercise. What are the transitions for the Josephus problem?

Solution. The transitions are:

$$J(n, k) = (J(n - 1, k) + k - 1)\%n + 1$$

$$J(1, k) = 1$$

$J(n, k)$ is the number of the person who would survive if there were n people and every k^{th} person was killed.