Design and Analysis of Algorithms

2018A7PS0193P

April 28, 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 difference 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 The Boolean Satisfiability 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 satisfiable if 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.

Next we have to prove that SAT is NP Hard. **TODO**.

16.4 The 0-1 Integer Programming Problem

The 0-1 integer programming problem, $01IPROG$ is as follows - given list of m linear inequalities with rational coefficients over n variables $u_1, \cdots, u_n$, find out if there is an assignment of 0s and 1s satisfying all the inequalities.

It is simple to show $01IPROG$ is NP - an NTM N can non-deterministically guess an assignment 0 or 1 to the variables and then it will evaluate and verify all the inequalities in polynomial time.

$01IPROG$ is also NP-Hard. We can prove this by showing that $SAT \leq_p 01IPROG$. For a given CNF formula $\Phi = C_1 \wedge C_2 \wedge \cdots \wedge C_m$ where for each clause there will be an inequality whose solution will be in either 0 or 1. Suppose that the clause is $u_1 \vee \neg u_2 \vee \neg u_3$, then the corresponding linear inequality will be $u_1 + (1 - u_2) + (1 - u_3) \geq 1$. We can prove that these are equivalent. From this, we can say that Φ has an equivalent 0-1 integer program, whose transformation can be done in linear time. Hence, $SAT \leq_p 01IPROG$, so $01IPROG$ is NP Hard.

This also completes a proof that $01IPROG$ is NP-Complete.

16.5 The Independent Set Problem, Revisited

We have already proved that $INDSET \in NP$. We can also show it is NP-Hard by proving that $3SAT \leq_p INDSET$. Say we have a m clause 3CNF formula $\Phi = C_1 \wedge C_2 \cdots C_m$. We will convert it into a $7m$ vertex graph G in polynomial time such that Φ is satisfiable if and only if the independent set is of size at least m.

Let us associate a cluster of 7 vertices in G with each clause of Φ. The vertices in a cluster associated with a clause correspond to the seven possible satisfying partial assignments, e.g. (0,1,0) or (1,1,0). We will call a partial assignment **consistent** if they have same value for all shared variables. We put an edge between two vertices in G if they correspond to inconsistent partial assigments.

The final output graph will have $7m$ vertices, and can be generated in polynomial time. Suppose Φ is satisfiable. Then, all the clauses are true for some assignment of variables. In each clause, we select the vertex corresponding to the partial assignment. There cannot be an edge between two chosen vertices since the partial assignments are inconsistent, so that means that they form an independent set. If it is not satisfiable, then there cannot be an independent set of size m. Hence, $3SAT \leq_p INDSET$.

17 NP Optimization Problems

An NP optimization problem Π consists of:

- A set of valid instances D_Π recognizable in polynomial time. The size of an instance

$I \in D_\Pi$, denotes by $|I|$ is defined as the number of bits needed to write I under the assumption that all numbers occuring in the instance are written in binary

- Each instance $I \in D_\Pi$ has a set of feasible solutions $S_\Pi(I)$. We require that $S_\Pi(I) \neq \phi$ and that every solution $s \in S_\Pi(I)$ is of a length polynomially bounded in $|I|$. Furthermore, there is a polynomial time algorithm that given a pair (I, S) decides whether $S \in S_\Pi(I)$
- There is a polynomial time computable objective function *obj* that assigns a non-negative number to each pair (I, S), where I is an instance and S is a feasible solution for I.
- Π is specified to either be a minimization or maximization problem. The restriction of Π to unit cost instances is called the **cardinality version** of Π.

With every NP optimization problem, we can naturally create a decision problem by giving a bound on the optimal solution. Thus, the decision version of NP optimization problem Π consists of pairs (I, B) where I is an instance of Π and B is a rational number. If Π is a minimization problem, then the answer to the decision version will be "yes" if and only if there is a feasible solution to I of cost $\leq B$.

An **approximation algorithm** produces a feasible solution that is "close" to the optimal one, and is time efficient. Let Π be a minimization problem, and let δ be a function such that $\delta : Z^+ \to Q^+$, with $\delta \geq 1$. An algorithm A is said to be a δ approximation algorithm for Π if, on each instance I, A produces a feasible solution for I such that $f_\Pi(I, S) \leq \delta(|I|).OPT(I)$, where $OPT(I)$ is the optimal solution. The running time of A must be bounded by a fixed polynomial in $|I|$.

A **polynomial time approximation scheme** (PTAS) is a family of algorithms $\{A_\epsilon\}$ where there is an algorithm for each $\epsilon > 0$, such that A_ϵ is a $(1 + \epsilon)$ - approximation algorithm for minimization and $(1 - \epsilon)$ for maximization.

A **fully polynomial time approximation scheme** (FPTAS) is an approximation scheme such that the running time of A_ϵ is bounded by a polynomial in $1/\epsilon$.

17.1 Linear Programming

Before we can get started with some algorithms, we must discuss linear programming. Given an n-vector c, a m-vector b and a $n \times m$ matrix a, a linear programming is of the form:

$$\min \sum_{j=1}^{n} c_j x_j$$

subject to

$$\sum_{j=1}^{n} a_{ij} x_j \geq b_i, i = 1, \cdots, m$$

$$x_j \geq 0, j = 1, \cdots, n$$

A solution is a n vector x that minimizes the given objective function $\sum_{i=1}^{n} c_j x_j$.

Solving LPs is in **P**. However, a variation of LP called Integer Linear Programming (ILP) is not. In ILPs we can add constraints that require the variable x_j to be an integer, or a member of a set. Solving ILPs can be proved to be NP-Complete.

Linear programming has a concept of **duality**. Consider the example of

$$\min 6x_1 + 4x_2 + 2x_3$$

subject to

$$4x_1 + 2x_2 + x_3 \geq 5$$

$$x_1 + x_2 \geq 3$$

$$x_2 + x_3 \geq 4$$

$$x_i \geq 0$$

Observe that since $6x_1 + 4x_2 + 2x_3 \geq (4x_1 + 2x_2 + x_3) + (x_1 + x_2) + (x_2 + x_3) \geq 5 + 3 + 4 = 12$. This means that the optimal value of the LP is at least 12.

Now suppose we take y_i of the i^{th} constraint. Then the lower bound achieved is $5y_1 + 3y_2 + 4y_3$. We need to ensure that:

$$6x_1 + 4x_2 + 2x_3 \geq y_1(4x_1 + 2x_2 + x_3) + y_2(x_1 + x_2) + y_3(x_2 + x_3)$$

This means that:

$$4y_1 + y_2 \leq 6$$

$$2y_1 + y_2 + y_3 \leq 4$$

$$y_1 + y_3 \leq 2$$

We want to maximize the lower bound subject to these constraints,which gives us a new LP:

$$\max 5y_1 + 3y_2 + 4y_3$$

subject to

$$4y_1 + y_2 \leq 6$$

$$2y_1 + y_2 + y_3 \leq 4$$

$$y_1 + y_3 \leq 2$$

$$y_i \geq 0$$

This maximization program is called the **dual** of the previous LP, which is the **primal**. The dual of a primal form can be expressed as:

$$\max \sum_{i=1}^{m} b_i y_i$$

subject to:

$$\sum_{i=1}^{m} a_{ij} y_i \leq c_j$$

$$y_i \geq 0$$

According to the **LP Duality Theorem**, the primal program has a finite optimum if and only if its dual has a finite optimum. Moreover, if x^* and y^* are the optimal solutions for the primal and dual programs respectively, then

$$\sum_{j=1}^{n} c_j x_j^* = \sum_{i=1}^{m} b_i y_i^*$$

The **weak duality theorem** states that if x and y are feasible solutions for the primal and dual respectively, then:

$$\sum_{j=1}^{n} c_j x_j \geq \sum_{i=1}^{m} b_i y_i$$

Let x and y be feasible solutions to the primal and dual LPs respectively. Then x and y obey the **complementary slackness conditions** if and only if they are optimal solutions to their respective LPs.

17.2 The Vertex Cover Problem

Given an undirected graph $G = (V, E)$ and a cost function on vertices $C : V \to Q^+$ find a minimum cost vertex cover. The special case in which all vertices are of unit cost , will be called the cardinality vertex cover problem.

Let us now define a 2-approximation algorithm for the cardinality vertex cover problem. Given a graph $H = (U, F)$, let M be a maximal matching. Let the set of matched vertices be V. No edge can be left uncovered by the set of vertices picked, since otherwise such an edge could have been added to the matching, and it would not be maximal.

Let us prove that this is a 2-approximation algorithm. The cardinality of the maching $|M| \leq |OPT|$. This means that $|V| = 2|M| \leq 2 \cdot OPT$. Hence proved.

17.3 The Weighted Vertex Cover Problem

We can formulate the weighted vertex cover problem as the following ILP:

$$\min \sum_{i \in V} w_i x_i$$

such that

$$x_i + x_j \geq 1 \forall (i, j) \in E$$

$$x_i \in \{0, 1\} \forall i \in V$$

We can relax the second condition to be:

$$0 \leq x_i \leq 1 \forall i \in V$$

Let S^* denote a vertex cover of minimum weight, found with the ILP. Then:

$$W_{LP} \leq W(S^*)$$

Given a fractional solution $\{x_i^*\}$, we define $S = \{i \in V : x_i^* \geq 0.5\}$. The set defined in this way is a vertex cover, and $|S| \leq 2 \cdot W_{LP}$. Hence, it is a 2 approximation.

17.4 The Set Cover Problem

17.4.1 The Problem

In the set cover problem, we are given a ground set of elements $E = \{e_1, \cdots, e_n\}$ and some subsets of those elements $S_1, S_2, \cdots, S_m$ where each $S_j \subseteq E$ as well as a non negative weight

$w_j \geq 0$ for each subset S_j. The goal is to find a minimum weight collection of subsets that covers all of E; that is, we wish to find $I \subseteq \{1, \cdots, m\}$ that minimizes $\sum_{j \in I} w_j$ such that $\bigcup_{j \in I} S_j = E$.

The vertex cover problem is a special case of the set cover problem. We can convert a vertex cover problem into a set cover problem by letting the ground set be the set of edges, and a subset is created for each vertex i containing edges incident to i.

17.4.2 The Rounding Algorithm

We can formulate it as the following ILP:

$$\min \sum_{j=1}^{m} w_j x_j$$

subject to

$$\sum_{j:e_i \in S_j} x_j \geq 1, i = 1, \cdots, m$$
$$x_j \in \{0, 1\}, j = 1, \cdots, m$$

Let us relax the problem, to get the following LP:

$$\min \sum_{j=1}^{m} w_j x_j$$

subject to

$$\sum_{j:e_i \in S_j} x_j \geq 1, i = 1, \cdots, m$$
$$x_j \geq 0, j = 1, \cdots, m$$

Let Z^*_{IP} denote the optimum value of the ILP, and Z^*_{LP} be the optimum value of the LP.

$$Z^*_{LP} \leq Z^*_{IP} = OPT$$

Let x^* denote on optimum solution to the LP. We include subset S_j in our solution if and only if $x^*_j \geq 1/f$, where f is the maximum number of sets in which any element appears. Formally, $f_i = |\{j : e_i \in S_j\}|$, then $f = \max_{i \leq n} f_i$. This is known as the **rounding algorithm**, since it is in effect rounding x^*_j to 1 if $x^*_j \geq 1/f$.

Firstly, is this algorithm even correct? Let e_i be covered if the solution contains some subset containing e_i. Because the optimal solution x^* is a feasible solution to the linear program,

$$\sum_{j:e_i \in S_j} x^*_j \geq 1$$

for e_i. There are $f_i \leq f$ terms in this sum, so at least one term must be at least $1/f$. Thus, there does exist $x_j^* \geq 1/f$, and e_i will be covered. Hence, the algorithm is correct, and provides a set cover.

This algorithm is in fact an f approximation algorithm. By our construction, $1 \leq f \cdot x_j^*$ for all $j \in I$. Hence,

$$\begin{aligned}\sum_{j \in I} w_j &\leq \sum_{j=1}^{m} w_j \cdot (f \cdot x_j^*) \\ &= f \sum_{j=1}^{m} w_j x_j^* \\ &= f \cdot Z_{LP}^* \leq f \cdot OPT\end{aligned}$$

17.4.3 The Dual Rounding Algorithm

The dual LP for our set cover problem is given by:

$$\max \sum_{i=1}^{n} y_i$$

subject to:

$$\sum_{i:e_i \in S_j} y_i \leq w_j$$

$$y_i \geq 0$$

By the weak duality theorem, for any feasible solution to the dual,

$$\sum_{i=1}^{n} y_i \leq Z_{LP}^* \leq OPT$$

Let y^* be the optimal solution to the dual LP, and consider the solution in which we choose all subsets for which the corresponding dual inequality is tight, i.e. the inequality is met with equality for S_j:

$$\sum_{i:e_i} y_i^* = w_j$$

Let I' denote the indices of the subsets in this solution. We will now prove that this algorithm is also an f-approximation algorithm for the set cover problem.

First, let us see if $S_j, j \in I'$ is a set cover. Suppose that e_k is uncovered. Then, for ever S_j containing e_k,

$$\sum_{i:e_i \in S_j} y_i^* < w_j$$

Let ϵ be the smallest difference between the RHS and LHS of all constraints involving e_k. Then, $\epsilon > 0$. Consider a new dual solution y' such that $y'_k = y^*_k + \epsilon$ and every other component of y' is the same as in y^*. Then y' is a dual feasible solution since for each j such that $e_k \in S_j$,

$$\sum_{i:e_i \in S_j} y'_i = \sum_{i:e_i \in S_j} y^*_i + \epsilon \leq w_j$$

by the definition of ϵ. For each j such that $e_k \notin S_j$,

$$\sum_{i:e_i \in S_j} y'_i = \sum_{i:e_i \in S_j} y^*_i \leq w_j$$

as before. This contradicts the optimality of y^*. Thus, it must be covered. Hence, I' is a set cover.

Now that we know it is a set cover, we can prove that is an f-approximation.

$$j \in I' \Leftrightarrow w_j = \sum_{i:e_i \in S_j} y^*_i$$

$$\begin{aligned}
\sum_{j \in I'} w_j &= \sum_{j \in I'} \sum_{i:e_i \in S_j} y^*_i \\
&= \sum_{i=1}^{n} |\{j \in I' : e_i \in S_j\}| \cdot y^*_i \\
&\leq \sum_{i=1}^{n} f_i y^*_i \\
&\leq f \sum_{i=1}^{n} y^*_i \\
&\leq f \cdot OPT
\end{aligned}$$

This algorithm is known as the **dual rounding algorithm**.

17.4.4 The Primal Dual Algorithm

The previous two algorithms for set cover required solving an LP. In the dual rounding algorithm, the optimal dual solution is a lower bound for OPT. Any feasible dual solution is also a lower bound for OPT, as:

$$\sum_{i=1}^{n} y_i \leq OPT$$

for any feasible dual solution.

In the Primal-Dual Algorithm, we start with a dual feasible solution, and use information from the dual to infer a primal, possibly infeasible solution. If the primal solution is indeed infeasible, then the dual solution is modified to increase the value of the dual objective function.

Algorithm 7: Primal Dual Algorithm

$y = 0$

$I = \phi$

while *there exists* $e_i \in \cup_{j \in I} S_j$ **do**

  Increase the dual variable y_i until there is some l with $e_i \in S_l$ such that

  $\sum_{j:j \in S_l} y_j = w_l$

  $I = I \cup \{l\}$

end

17.5 The Knapsack Problem

17.5.1 The Problem

In the knapsack problem, we are given a set of n items $I = \{1 \cdots n\}$where each item has a value v_i and a size s_i. The knapsack has a capacity B, where B is also a positive integer. The goal is to find a subset of items $S \subseteq I$ that maximizes the value $\sum_{i \in S} v_i$ of items in the knapsack, subject to the constraint $\sum_{i \in S} s_i \leq B$.

17.5.2 The Dynamic Programming Solution

There is an obvious DP solution to this.

Algorithm 8: Dynamic Programming Solution

$A(1) = \{(0,0), (s_1, w_1)\}$
for *j from 2 to n* **do**
    $A(j) = A(j-1$
    for *each* $(t, w) \in A(j-1$ **do**
        if $t + s_j \leq B$ **then**
            Add $(t + s_j, w + v_j)$ to $A(j)$
        end
    end
    Remove dominated pairs from $A(j)$
end
return $\max_{(t,w)\in A(n)} w$

This will always find the correct solution in $O(n \min(B, V))$ time. This is not a polynomial time algorithm. Encoding the input number in binary would give it a length of $\log B$, so the running time $O(nB)$ is exponential in the size of the input number B. However, if we encoded it in unary, it woud be polynomial in the size of the input. This makes it **pseudopolynomial** .

Formally, an algorithm for a problem Π is said to be pseudopolynomial if its running time is polynomial in the size of the input when the numeric part of the input is encoded in unary.

17.6 FPTAS Solution

Suppose we measure value as integer multiples of μ and convert each value v_i by rounding down to the nearest integer multiple of μ, called v_i'. We can then run the dynamic programming solution on the items with size s_i and values v_i'. The optimal solution for this can be considered a near optimal solution for the true data.

Imagine we used $\tilde{v}_i = v_i'\mu$ instead of v_i. Then, the value is inaccurate by at most μ, so each optimal solution has its value changed by at most $n\mu$. We want the error introduced to be at most ϵ times a lower bound on the optimal value. We can see that $m = \max_{i\in I} v_i$ is a lower bound on OPT. Thus, it would make sense to have μ such that $n\mu = \epsilon M$, i.e.

$$\mu = \frac{\epsilon M}{n}$$

With the modified values,

$$V' = \sum_{i=1}^{n} v_i' = \sum_{i=1}^{n} \left\lfloor \frac{v_i}{\epsilon M/n} \right\rfloor = O(n^2/\epsilon)$$

Thus, the running time is $O(n \min(B, V')) = O(n^3/\epsilon)$ and is bounded by a polynomial in $1/\epsilon$.

Let us now show that this is an FPTAS with a solution at least $(1 - \epsilon)$ times the value of an optimal solution. Let S be the set of items returned by the algorithm, and let O be an optimal set of items. Then:

$$\begin{aligned}
\sum_{i \in S} v_i &\geq \mu \sum_{i \in S} v_i' \\
&\geq \mu \sum i \in O v_i' \\
&\geq \sum_{i \in O} v_i - |O|\mu \\
&\geq \sum_{i \in O} v_i - n\mu \\
&= \sum_{i \in O} v_i - \epsilon M \qquad\qquad \geq OPT - \epsilon OPT = (1 - \epsilon)OPT
\end{aligned}$$

17.7 Complexity classes for Approximation

17.7.1 NPO and APX

An optimization problem belongs to the class **NPO** if and only if it's decision version is a member of **NP**.

The complexity class **APX** is the set of NPO problems having a constant factor approximation algorithm.

It is obvious that $\mathbf{APX} \subseteq \mathbf{NPO}$. Let us show that $\mathbf{APX} \neq \mathbf{NPO}$ by taking up the example of TSP - a problem not in APX but in NPO.

This means that for any polynomial time computable function $\alpha(n)$, TSP cannot be approximated within a factor of $\alpha(n)$, unless $P = NP$.

Assume that there exists a factor $\alpha(n)$ polynomial time approximation algorithm, A, for the general TSP algorithm. We will show that A can be used for deciding the Hamiltonian cycle problem in polynomial time, implying $P = NP$.

Consider the following reduction from a graph G on n vertices to an edge weighted complete graph G'. Assign a weight of 1 to edges of G, and a weight $\alpha(n) \cdot n$ to non edges, to obtain

G'. Now, if G has a Hamiltonian cycle, then the corresponding tour in G' has a cost n. On the other hand, if it does not, then any tour must use an edge of cost $\alpha(n) \cdot n$, and therefore has cost greater than $\alpha(n) \cdot n$. Now we can use A for deciding which of the two cases holds, and then return one answer accordingly.

17.7.2 PTAS and APX

We defined the class PTAS as the set of NPO problems that may be approximated to any constant factor close to 1 ($1 + \epsilon$ or $1 - \epsilon$). For a fixed ϵ the running time will be polynomial in input size m. In general, the complexity may not be polynomial in $1/\epsilon$.

By definition, $PTAS \subseteq APX$, but we can show that $PTAS \neq APX$. An example of a problem in APX and not in PTAS is the **bin packing problem**.

In the bin packing problem, we are given n items with sizes $\alpha_1, \alpha_2, \cdots, \alpha_n \in [0, 1]$. We want to find a packing in unit size bins that minimizes the number of bins used.

The first-fit algorithm considers items in an arbitrary order. In the i^{th} step, it has a list of partially packed bins, say $B_1, \cdots, B_k$. It attempts to put the next item α_i in one of these bins in this order. If α_i does not fit in any of the bins, it will open anew bin B_{k+1} and put α_i in there.

If the algorithm uses m bins, then at least $m - 1$ bins are more than half full. Therefore

$$\sum_{i=1}^{n} \alpha_i > \frac{m-1}{2}$$

Since the sum of the item sizes is a lower bound on OPT, $m - 1 < 2OPT \Rightarrow m < 2OPT$.

For any *epsilon* > 0, there is no approximation algorithm having a guarantee of $3/2 - \epsilon$ unless $P = NP$.

If there was such an algorithm, we can use it to solve the NP-Hard problem of deciding if there is a way to partition n non-negative numbers $a_1, \cdots, a_n$ into two sets each adding up to $(1/2) \sum_i \alpha_i$. Clearly, the answer to this question is "yes" the n items can be packed in 2 bins of size $(1/2) \sum_i a_i$. If the answer is yes, the $3/2 - \epsilon$ factor algorithm will have to give an optimal packing, and thereby solve the partition problem.

17.7.3 FPTAS and PTAS

We already know that $FPTAS \subseteq PTAS$. Let us show that $FPTAS \neq PTAS$.

An optimization problem Π is **polynomially bounded** if there exists a polynomial p such that, for any instance x and for $y \in S_\Pi(x), Obj_\Pi(x,y) \leq p(|x|)$. No NP-Hard polynomially bounded optimization problem belongs to the class FPTAS unless $P = NP$.

Suppose we have a FPTAS A for Π which for any instance x and for any ration $\epsilon(0 < \epsilon < 1)$ runs in time bounded by $q(|x|, 1/\epsilon)$ for a suitable polynomial q. Since Π is polynomially bounded, there exists a polynomial p such that for any instance x, $OPT(x) \leq p(|x|)$. If we choose $\epsilon = 1/p(|x|)$, then $A(x,\epsilon)$ provides an optimal solution of x as follows:

$$A \in FPTAS \Rightarrow \frac{OPT(x)}{obj_\Pi(x, A(x,\epsilon))} \leq 1 + \frac{1}{p(|x|)}$$

$$obj_\Pi(x, A(x,\epsilon)) \geq OPT(x)\frac{p(|x|)}{p(|x|)+1} = -\frac{OPT(x)}{p(|x|)+1} + OPT(x) > OPT(x) - 1$$

This means that obj_Π is $OPT(x)$, meaning A is an optimal solution. If $P \neq NP$, then $FPTAS \neq PTAS$.

18 Probabilistic Turing Machine

18.1 Markov and Chebyshev's Inequalities

See my notes on ASP for this.

18.2 Chernoff Bounds for the Sum of Poisson Trials

Let X_i be independent random variables such that $P(X_i = 1) = p_i$. Let $X = \sum X_i$, and let $\mu = E[X] = \sum E[X_i] = \sum p_i$.

For a given $\delta > 0$, we are interested in bounds on $P(X \geq (1+\delta)\mu)$ and $P(X \leq (1-\delta)\mu)$, i.e., the probability that X deviates from μ by $\delta\mu$ or more.

$$E[e^{tx_i}] = p_i e^t + (1 - p_i) = 1 + p_i(e^t - 1) \leq e^{p_i(e^t-1)}$$

$$E[e^{tX}] = \prod E[e^{tX_i}] \leq e^{(e^t-1)\sum p_i} = e^{(e^t-1)\mu}$$

For any $t > 0$:

$$P(X \geq a) = P(e^{tX} \geq a^{ta}) \leq \frac{E[e^{tX}]}{e^{ta}} \leq \frac{e^{(e^t-1)\mu}}{e^{ta}}$$

Putting $a = (1+\delta)\mu$, we get:

$$P(X \geq (1+\delta)\mu) \leq \frac{e^{(e^t-1)\mu}}{e^{t(1+\delta)\mu}}$$

For any $\delta > 0$, we can set $t = \ln(1+\delta) > 0$ to get:

$$P(X \geq (1+\delta)\mu) \leq \left(\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right)^{\mu}$$

Using this same approach, we can find $P(X \leq (1-\delta)\mu)$. By setting $t = \ln(1-\delta) < 0$ for $0 < \delta < 1$,

$$P(X \leq (1-\delta)\mu) \leq \left(\frac{e^{-\delta}}{(1-\delta)^{1-\delta}}\right)^{\mu}$$

18.3 Definition

A PTM is a TM with two transition functions δ_0, δ_1. To execute a PTM M on an input x we choose in each step with probability 0.5 to apply δ_0, and probability 0.5 to apply δ_2. This choice is independent of all previous choices. We denote the random variable corresponding to the output of M by $M(x)$.

For a function $T : N \to N$, we say that M runs in $T(n)$ time if for any input x, M halts on X within $T(|x|)$ steps regardless of the choices made.

18.4 The complexity class BPTIME and BPP

For a language $L \subseteq \{0,1\}^*$ and $x \in \{0,1\}^*$, we define $L(X) = 1$ if $x \in L$ and $L(x) = 0$ otherwise.

For $T : N \to N$ and $L \subseteq \{0,1\}^*$ we say that a PTM decides L in time $T(n)$ if for every $x \in \{0,1\}^*$, M halts in $T(|x|)$ steps regardless of its random choices, and $P(M(x) = L(x)) \geq 2/3$.

BPTIME$(T(n))$ is the class of languages decided by PTMs in $O(T(n))$.

BPP is defined as:

$$\mathbf{BPP} = \bigcup_{c} \mathbf{BPTIME}(n^c)$$

Algorithms in the class **BPP** are called **Atlantic City Algorithms**.

Let M be an Atlantic City algorithm accepting L with error probability bounded from above by p. We want to reduce error probability to ϵ. We create a PTM M' which simulate M n times, and picks the majority answer. Hence,

$$P(error) = P(X < \frac{n}{2})$$

where X is a r.v. defined by $X = \sum_{i=1}^{n} X_i$. X_i corresponds to the i^{th} trial such that $X_i = 1$ if i^{th} simulation of $M(x)$ gives correct answer.

$$P(X_i = 1) \geq 1 - p$$

$$P(X_i = 0) \leq p$$

$$E(X_i) \geq 1 - p$$

$$\mu \geq n(1-p) \geq \frac{n}{2}$$

We apply Chernoff's Bounds, and find that

$$n \geq \frac{\ln \epsilon}{\frac{1}{2} + (\frac{1}{2} - p) \ln \frac{0.5-p}{1-p}}$$

18.5 The complexity class RTIME and RP

RTIME$(T(n))$ contains every language L for which there is a PTM M running in $T(n)$ time such that:

$$x \in L \Rightarrow P[M(x) = 1] \geq \frac{2}{3}$$
$$x \notin L \Rightarrow P[M(x) = 1] = 0$$

Then **RP** is defined as:

$$\mathbf{RP} = \bigcup_{c>0} \mathbf{RTIME}(n^c)$$

The complexity class **CoRP** is such that $L \in$ **CoRP** if and only if:

$$x \in L \Rightarrow P[M(x) = 0] = 0$$

$$x \notin L \Rightarrow P[M(x) = 0] \geq \frac{2}{3}$$

Algorithms in the class **RP** and **CoRP** are called **Monte Carlo Algorithms**.

Let M be a Monte Carlo algorithm with error probability bounded by p, accepting L. We want to reduce this to ϵ. Consider a PTM M' which simulate M, n times. Then, the probability of error would be p^n. Hence, to reduce it to ϵ, we want

$$n \geq \frac{\log \epsilon}{\log p}$$

18.6 The complexity class ZTIME and ZPP

The class $\mathbf{ZTIME}(T(n))$ contains all the languages for which there is an expected time $O(T(n))$ PTM M such that for every input x, whenever M halts on x, the output $M(x)$ is exactly $L(x)$.

We define

$$\mathbf{ZPP} = \bigcup_{c>0} \mathbf{ZTIME}(n^c)$$

Algorithms in this class are called **Las Vegas Algorithms**.

18.7 The relation between complexity classes

We have the following properties:

1. $\mathbf{ZPP} = \mathbf{RP} \cap \mathbf{CoRP}$
2. $\mathbf{RP} \subseteq \mathbf{BPP}$
3. $\mathbf{CoRP} \subseteq \mathbf{BPP}$
4. $\mathbf{RP} \subseteq \mathbf{NP}$
5. $\mathbf{P} \subseteq \mathbf{BPP}$
6. $\mathbf{P} \subseteq \mathbf{RP}$
7. $\mathbf{P} \subseteq \mathbf{CoRP}$
8. $\mathbf{P} \subseteq \mathbf{ZPP}$

First let us prove property (1). Let $L \in ZPP$, which means there exists a PTM M such that $x \in L \Rightarrow M(x) = 1$ and $M(x) = 0$ otherwise, with an expected polynomial time complexity of $p(n)$.

Let us show that $L \in \mathbf{RP}$. We construct a PTM M_1 that simulates the moves of M in $3p(n)$ steps. If during this time M outputs 0 or 1, then M_1 outputs 0 or 1. If M does not give any output, then M_1 outputs 0.

L must also be a member of $\mathbf{CoRP}$. We do the same thing as with M_1, but it instead gives output 1 if M does not give any output. We can see the error in this case from Markov's inequality:

$$P(error) \leq P(T(n) > 3p(n)) < \frac{1}{3}$$

This way, $ZPP \subseteq RP \cap CoRP$.

Let $L \in RP \cap CoRP$. This means there exist PTMs M_1 and M_2 such that both run in polynomial time. Let the larger time complexity be $p(n)$. Then,

$$x \in L \Rightarrow P(M_1(x) = 0) \leq \frac{1}{3}$$

$$x \notin L \Rightarrow P(M_1(x) = 1) = 0$$

$$x \in L \Rightarrow P(M_2(x) = 0) = 0$$

$$x \notin L \Rightarrow P(M_2(x) = 1) \leq \frac{1}{3}$$

We construct a PTM M that runs M_1 and M_2 simultaneously. If they both output 0, then M outputs 0. If they both output 1, then M outputs 1. If M_1 outputs and M_2 outputs 1, then M repeats running them until it finds a solution. We can prove that the expected running time of this is $\frac{3}{2}p(n)$.

18.8 The Birthday Problem

If there are m people and n possible birthdays, then the probability that they all have different birthdays is:

$$\left(1 - \frac{1}{n}\right) \cdots \left(1 - \frac{m-1}{n}\right) = \prod_{j=1}^{m-1} \left(1 - \frac{j}{n}\right)$$

Using the fact that $1 - \frac{k}{n}$ is approximately $e^{-k/n}$ when k is small compared to n, we see that if m is small compared to n then:

$$\prod_{j=1}^{m-1} \left(1 - \frac{j}{n}\right) = \prod_{j=1}^{m-1} e^{-j/n} = e^{-m(m-1)/2n} = e^{-m^2/2n}$$

Let E_k be the event that the k^{th} person's birthday does not match any of the birthdays of the first $k-1$ people. Then the probability that the first k fail to have distinct birthdays is:

$$P(\overline{E_1} \cup \overline{E_2} \cup \cdots \cup \overline{E_k}) \leq \sum_{i=1}^{k} P(\overline{E_i}) \leq \sum_{i=1}^{k} \frac{i-1}{n} = \frac{k(k-1)}{2n}$$

If $k \leq \sqrt{n}$ this probability is less than 0.5, so with $\sqrt{n}$ people the probability is at least 0.5 that all birthdays will be distinct.

Now assume that the first $\sqrt{n}$ people all have distinct birthdays. Each person after that has probability of at least $1/sqrtn$ of having the same birthday as one of the first $\sqrt{n}$ people.

Hence the probability that the next $\sqrt{n}$ peole all have different birthdays than the first $\sqrt{n}$ people is at most:

$$\left(1 - \frac{1}{\sqrt{n}}\right)^{\sqrt{n}} < \frac{1}{e} < \frac{1}{2}$$

Hence, once there are $2\sqrt{n}$ people, the probability is at most $1/e$ that all birthdays will be distinct.

18.9 The Balls and Bins Model

The birthday paradox is an example of a more general mathematical framework that is often formulated in the terms of balls and bins. We have m balls that are thrown into n bins, with the location of each ball chosen independently and uniformly at random from the n possibilities. What does the distribution of the balls in the bins look like? The question behind the birthday paradox is whether or not there is a bin with two balls.

When n balls are thrown independently and uniformly at random into n bins, the probability that the maximum load is more than $3 \ln n / \ln \ln n$ is at most $1/n$ for n sufficiently large.

The probability that bin 1 receives m balls is at most $\binom{n}{m}(\frac{1}{n})^m$. We use the inequality

$$\binom{n}{m}\left(\frac{1}{n}\right)^m \leq \frac{1}{m!} \leq \left(\frac{e}{m}\right)^m$$

Applying a union bound again allows us to find that, for $m \geq 3 \ln n / \ln \ln n$, the probability that any bin receives at least m balls is bounded above by:

$$\begin{aligned} n\left(\frac{e}{m}\right)^m &\leq n\left(\frac{e \ln \ln n}{3 \ln n}\right)^{3 \ln n / \ln \ln n} \\ &\leq n\left(\frac{\ln \ln n}{\ln n}\right)^{3 \ln n / \ln \ln n} \\ &\leq \frac{1}{n} \end{aligned}$$

For sufficiently large n (steps omitted because I am lazy).

19 Euclid's Algorithm

See Number Theory notes for the necessary prerequisites and understanding this algorithm.

Let us find the complexity of Euclid's algorithm. In Number Theory, we proved that the number of steps is at most 5 times the number of digits in it's base 10 representation. This implies that it is $O(\log n)$.

20 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.