Collection of Interesting Problems in Algorithms

Sameer Kumar

November 2019

Contents

1 Divide and Conquer

1. (**Domain transformation**) We analyzed the running time of Mergesort by the recurrence $T(n) = 2T(n/2) + O(n)$. The actual Mergesort recurrence is somewhat messier:

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + O(n).$$

We'll justify in this problem that ignoring the ceilings and floors in a recurrence is okay afterall using a technique called *domain transformation*.

First, because we are deriving an upper bound, we can safely overestimate $T(n)$, once by pretending that the two subproblem sizes are equal, and again to eliminate the ceiling:

$$T(n) \leq 2T(\lceil n/2 \rceil) + n \leq 2T(n/2 + 1) + n.$$

Second, we define a new function $S(n) = T(n + \alpha)$ for some α. **Complete the second step.** Show that you can find a nice α so that $S(n) \leq 2S(n/2) + O(n)$ does hold, and conclude from there that $T(n) = O(n \log n)$. [Think about: how to remove floors by similar arguments.]

Solution: *Our hope in doing domain transformation is that getting an exact solution with the messy recurrence equation is hard. But if we do domain transformation then we can produce a tight asymptotic solution. Given, that we are overestimating the time bound by pretending that the two subproblem sizes are equal, and again to eliminate the ceiling:*

$$T(n) \leq 2T(\lceil n/2 \rceil) + n$$
$$\leq 2T(n/2 + 1) + n$$

Now we define a new function $S(n) = T(n + \alpha)$, where α is a unknown constant, chosen so that $S(n)$ satisfies the Master-Theorem-ready recurrence $S(n) \leq 2S(n/2) + O(n)$. To figure out the correct value of α, we compare two versions of the recurrence for the function $T(n + \alpha)$:

$$S(n) \leq 2S(n/2) + O(n) \qquad \Longrightarrow \qquad T(n + \alpha) \leq 2T(n/2 + \alpha) + O(n)$$
$$T(n) \leq 2T(n/2 + 1) + n \qquad \Longrightarrow \qquad T(n + \alpha) \leq 2T((n + \alpha)/2 + 1) + n + \alpha$$

For these two recurrences to be equal, we need $n/2 + \alpha = (n + \alpha)/2 + 1$, which implies that α is 2. The Master Theorem now tells us that $S(n) = O(n \log n)$, so

$$T(n) = S(n - 2) = O((n - 2) \log(n - 2)) = O(n \log n)$$

Similarly we can remove floor by:

$$T(n) \geq 2T(\lfloor n/2 \rfloor) + n$$
$$\geq 2T(n/2 - 1) + n$$

Now we define a new function $S(n) = T(n + \alpha)$, where α is a unknown constant, chosen so that $S(n)$ satisfies the Master-Theorem-ready recurrence $S(n) \geq 2S(n/2) + O(n)$. To figure out the correct value of α, we compare two versions of the recurrence for the function $T(n + \alpha)$:

$$S(n) \geq 2S(n/2) + O(n) \qquad \Longrightarrow \qquad T(n + \alpha) \geq 2T(n/2 + \alpha) + O(n)$$
$$T(n) \geq 2T(n/2 - 1) + n \qquad \Longrightarrow \qquad T(n + \alpha) \geq 2T((n + \alpha)/2 - 1) + n + \alpha$$

For these two recurrences to be equal, we need $n/2 + \alpha = (n + \alpha)/2 - 1$, which implies that α is -2. The Master Theorem now tells us that $S(n) = \Omega(n \log n)$, so

$$T(n) = S(n + 2) = O((n + 2) \log(n + 2)) = \Omega(n \log n)$$

So, $T(n) = \Theta(n \log n)$. Domain transformations are useful for removing floors, ceilings, and lower order terms from the arguments of any recurrence that otherwise looks like it ought to fit either the Master Theorem or the recursion tree method.

2. (**Quicksort**) We were not precise about the running time of Quicksort in class (for a good reason). We will give some case studies in this problem (and appreciate the subtlety).

(a) Given an input array of n elements, suppose we are unlucky and the partitioning routine produces one subproblem with $n - 1$ elements and one with 0 element. Write down the recurrence and solve it. Describe an input array that costs this amount of running time to get sorted by Quicksort.

(b) Now suppose that the partitioning always produces a 9-to-1 proportional split. Write down the recurrence for $T(n)$ and solve it.

(c) What is the running time of Quicksort when all elements of the input array have the same value?

Solution: Generally for any Divide-and-Conquer problems we write the recursion relation as follows

$$T(n) = \begin{cases} \Theta(1) & if\, n \leq c \\ aT(n/b) + D(n) + C(n) & otherwise \end{cases}$$

$T(n)$ is the running time on a problem of size n. If the problem size is small enough, say $n \leq c$ for some constant c, the straightforward solution takes constant time, which we write as $\Theta(1)$. Suppose that our division of the problem yields a subproblems, each of which is $1/b$ the size of the original. It takes time $T(n/b)$ to solve one subproblem of size $n = b$, and so it takes time $aT(n/b)$ to solve a of them. If we take $D(n)$ time to divide the problem into subproblems and $C(n)$ time to combine the solutions to the subproblems into the solution to the original problem, then we end up with above recursion equation.

*All Divide-and-Conquer problems algorithms have same boilerplate template i.e., we have (a) **Divide** the problem into a number of subproblems that are smaller instances of the same problem, (b) **Conquer** the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner, and (c) **Combine** the solutions to the subproblems into the solution for the original problem.*

In Quicksort, no work is needed to combine the sorted array because sorting happens on the original array hence $C(n) = 0$ (no work in combining). Running time of Divide routine or the partition routine in Quicksort is $D(n) = \Theta(n)$, where n is the problem size. The for loop makes exactly n iterations, each of which takes at most constant time. The part outside the for loop takes at most constant time. Since n is the size of the subarray, partition routine takes at most time proportional to the size of the subarray it is called on. Running time of Conquer routine will depend how lucky we are in partitioning the array.

(a) ***Worst-case partitioning*** *The worst-case behavior for quicksort occurs when the partitioning routine produces one subproblem with $n - 1$ elements and one with 0 elements. Let us assume that this unbalanced partitioning arises in each recursive call. The partitioning costs $\Theta(n)$ time. Since the recursive call on an array of size 0 just returns,*

$T(0) = \Theta(1)$, *and the recurrence for the running time is*

$$T(n) = T(n-1) + T(0) + \Theta(n)$$
$$= T(n-1) + \Theta(n)$$

$\Theta(n)$ can be written as n in the equation which give us $T(n) = T(n-1) + n$. If we write the recursion tree for this then we get a tree with one branch at each level and the work done at each will be $n, n-1, n-2, \ldots, 1$. Hence total work needed would be summation of the arithmetic series which will give us $\Theta(n^2/2 + n/2)$. This can be approximated to $T(n) = \Theta(n^2)$.

Thus, if the partitioning is maximally unbalanced at every recursive level of the algorithm, the running time is $\Theta(n^2)$. Therefore the worst-case running time of quicksort is no better than that of insertion sort. Moreover, the $\Theta(n^2)$ occurs when the input array is already completely sorted; a common situation in which insertion sort runs in $O(n)$ time. We can give more examples for worst-case scenarios for Quicksort, for example reverse sorted array, or all the elements are same in the array. Input array with worst-case running time example: [1, 2, 3, 4, 5, 6]. For this example Quicksort algorithm has to go through each element and ends up calling the sort function again and again; as the array is sorted every time it picks the pivot it will be minimum.

(b) ***Balanced partitioning*** *Here we have unbalanced partitioning of 90:10 split. For this we can write the recurrence as $T(n) = T(9n/10) + T(n/10) + \Theta(n)$. To solve this recurrence problem we will use recursion trees. At root the tree will split in 9/10th and 1/10th parts of the original problem and this will keep on repeating for each node. Since, the 1/10th decreases faster it will reach to $T(1)$ faster than its counter part. This will happen when $n/(10^i) = 1 \implies i = \log_{10} n$. Also we observe that for each level with depth less than $\log_{10} n$ we will need to cn amount of work to be done (here c is a positive constant hidden in $\Theta(n)$). At the subtrees of the 9/10th part reach to $T(1)$ when $n/((10/9)^j) = 1 \implies j = \log_{10/9} n$. At for each level at depth between i and j we need to do at most cn work. Hence total work we need that has to done can be bounded by $O(n \log_{10/9} n)$. Since $\log_a n = \log_b n / \log_b a$, we can write $n \log_{10/9} n$ as $n \log n$ and neglect the term $\log_2 10/9$. Hence, total cost of Quicksort for this case is therefore $T(n) = O(n \log n)$.*

<u>***Note:***</u> *With a 9-to-1 proportional split at every level of recursion, which intuitively seems quite unbalanced, quicksort runs in $O(n \log n)$ time; asymptotically the same as if the split were right down the middle. Indeed, even a 99-to-1 split yields an $O(n \log n)$ running time. In fact, any split of* constant *proportionality yields a recursion tree of depth $\Theta(\log n)$, where the cost at each level is $O(n)$. The running time is therefore $O(n \log n)$ whenever the split has constant proportionality.*

(c) *If all the input values of the array are same then this comes under the case of the worst-partitioning as discussed in the above parts. Hence for array with all same values will have $\Theta(n^2)$ running time where n is the input array size. To reiterate the above discussion the traditional implementation of Quicksort which partitions the problem into two parts $<$ and $\geq$ will need more run time on identical inputs (or sorted arrays). While*

no swaps will necessarily occur, it will cause n recursive calls to be made and each of which need to make a comparison with the pivot and n-recursion elements i.e., $O(n^2)$ comparisons need to be made.

3. (**Sumerians' multiplication algorithm**) The clay tablets discovered in Sumer led some scholars to conjecture that ancient Sumerians performed multiplication by reduction to *squaring*, using an identity like

$$x \cdot y = (x^2 + y^2 - (x-y)^2)/2.$$

In this problem, we will investigate how to actually square large numbers.

   (a) Describe a variant of Karatsuba's algorithm that squares any n-digit number in $O(n^{\log 3})$ time, by reducing to squaring three $\lceil n/2 \rceil$-digit numbers. (Karatsuba actually did this in 1960.)

   (b) Describe a recursive algorithm that squares any n-digit number in $O(n^{\log_3 6})$ time, by reducing to squaring six $\lceil n/3 \rceil$-digit numbers.

   (c) Describe a recursive algorithm that squares any n-digit number in $O(n^{\log_3 5})$ time, by reducing to squaring only five $(n/3 + O(1))$-digit numbers. [Hint: What is $(a+b+c)^2 + (a-b+c)^2$?]

Solution:

   (a) We know that any number given base B can be written as

$$a = a_1 B^m + a_0,$$

   Here we are dealing with binary integers and if the size of the given array is n then we can write

$$a = a_1 2^{n/2} + a_0,$$
$$a^2 = a_1^2 2^n + a_1 a_0 2^{\frac{n}{2}+1} + a_0^2.$$

   Here a_1 and a_0 are $\lceil n/2 \rceil$ numbers. We can simplify $a_1 a_0$ using the identity given in the question to,

$$a_1 a_0 = \frac{a_1^2 + a_0^2 - (a_1 - a_0)^2}{2}.$$

   If we assume $a_1 = x$, $a_0 = y$ and $a_1 - a_0 = z$, then

$$a^2 = x^2 2^n + (x^2 + y^2 - z^2)2^{n/2} + y^2.$$

   Hence we only need square of three numbers of size $\lceil n/2 \rceil$ to square a number of size n. Using the Divide-and-Conquer approach we can write the recurrence relation. We need constant time in dividing the problem in to half i.e., $O(1)$. When we square a number of size $\lceil n/2 \rceil$ the size will be at most n hence at each step combining will take $O(n)$

time. At each step we divide the problem to 3 sub problems which are half the size of the original problem (neglecting the ceiling). Final equation will be

$$T(n) = 3T(n/2) + O(n)$$

Using master theorem we get $T(n) = O(n^{\log 3})$.

(b) *Using the same logic mentioned we can write a number of size n as sum of three $\lceil n/3 \rceil$ as follows,*

$$a = a_2 2^{2n/3} + a_1 2^{n/3} + a_0,$$

here a_2, a_1, a_0 are size of at most $\lceil n/3 \rceil$. After expand a^2 and simplifying we get,

$$a^2 = a_2^2 2^{4n/3} + a_1^2 2^{2n/3} + a_0^2 + a_1 a_2 2^{n+1} + a_1 a_0 2^{\frac{n}{3}+1} + a_2 a_0 2^{\frac{2n}{3}+1}.$$

If we assume $a_0 = x$, $a_1 = y$, $a_2 = z$, $a_1 - a_0 = p$, $a_2 - a_0 = q$, and $a_2 - a_1 = r$. Then using identity given in the question we can simplify $a_1 a_0$, $a_2 a_0$, and $a_1 a_2$ as

$$xy = \frac{x^2 + y^2 - p^2}{2}$$
$$zx = \frac{x^2 + z^2 - q^2}{2}$$
$$zy = \frac{z^2 + y^2 - r^2}{2}.$$

Now we can write a^2 as,

$$a^2 = z^2 2^{4n/3} + y^2 2^{2n/3} + x^2 + (z^2 + y^2 - r^2) 2^n$$
$$+ (x^2 + y^2 - p^2) 2^{n/3} + (z^2 + x^2 - q^2) 2^{2n/3}.$$

Hence we only need square of six numbers of size $\lceil n/3 \rceil$ to square a number of size n. Using the Divide-and-Conquer approach we can write the recurrence relation. We need constant time in dividing the problem in to one thirds i.e., $O(1)$. When we square a number of size $\lceil n/3 \rceil$ the size will be at most n hence at each step combining will take $O(n)$ time. At each step we divide the problem to 6 sub problems which are $1/3^{rd}$ the size of the original problem (neglecting the ceiling). Final equation will be

$$T(n) = 6T(n/3) + O(n)$$

Using master theorem we get $T(n) = O(n^{\log_3 6})$.

(c) **Toom - Cook multiplication:** *Here we will use the technique which was introduced by Toom and Cook in their algorithm for multiplication (source). Assume we have been given a digit of size n and it is in base 2 and we split it into three parts each of size $\lceil n/3 \rceil$ and the digits will be shifted according $\lfloor n/3 \rfloor$. Now we can write the number as follows,*

$$a = a_2 2^{2n/3} + a_1 2^{n/3} + a_0.$$

If we square the given number it will be,

$$a^2 = a_2^2 2^{4n/3} + a_1^2 2^{2n/3} + a_0^2 + a_1 a_2 2^{n+1} + a_1 a_0 2^{\frac{n}{3}+1} + a_2 a_0 2^{\frac{2n}{3}+1}$$
$$= a_2^2 2^{4n/3} + (2a_1 a_2)2^n + (a_1^2 + 2a_2 a_0)2^{\frac{2n}{3}} + (2a_1 a_0)2^{\frac{n}{3}} + a_0^2.$$

Now we will write the given number as terms of a polynomial in x i.e.,

$$p(x) = a_2 x^2 + a_1 x + a_0$$

where if we substitute $x = 2^{n/3}$ we get the number back,

$$p(2^{n/3}) = a = a_2 2^{2n/3} + a_1 2^{n/3} + a_0.$$

We know from algebra that to compute the coefficients of a polynomial of degree d we need to $d + 1$ points. Lets take small numbers and substitute them into the polynomial to get,

$$p(0) = a_0$$
$$p(1) = a_2 + a_1 + a_0$$
$$p(-1) = a_2 - a_1 + a_0$$
$$p(2) = 4a_2 + 2a_1 + a_0$$
$$p(-2) = 4a_2 - 2a_1 + a_0.$$

Let $r(x)$ be polynomial such that $r(x) = p(x) * p(x)$, then

$$r(0) = p(0)p(0)$$
$$= a_0^2$$

$$r(1) = p(1)p(1)$$
$$= (a_2 + a_1 + a_0)^2$$
$$= a_2^2 + a_1^2 + a_0^2 + 2a_1 a_2 + 2a_2 a_0 + 2a_1 a_0.$$

$$r(-1) = p(-1)p(-1)$$
$$= (a_2 - a_1 + a_0)^2$$
$$= a_2^2 + a_1^2 + a_0^2 - 2a_1 a_2 + 2a_2 a_0 - 2a_1 a_0$$

$$r(2) = p(2)p(2)$$
$$= (4a_2 + 2a_1 + a_0)^2$$
$$= 16a_2^2 + 4a_1^2 + a_0^2 + 16a_1 a_2 + 2a_2 a_0 + 4a_1 a_0$$

$$r(-2) = p(-2)p(-2)$$
$$= (4a_2 - 2a_1 + a_0)^2$$
$$= 16a_2^2 + 4a_1^2 + a_0^2 - 16a_1 a_2 + 2a_2 a_0 - 4a_1 a_0.$$

If we assume that the coefficients of the polynomial $r(x)$ as r_0, r_1, r_2, r_3, r_4, then we can write the following relation

$$r(x) = r_4 x^4 + r_3 x^3 + r_2 x^2 + r_1 x + r_0$$

$$\begin{bmatrix} r(0) \\ r(1) \\ r(2) \\ r(3) \\ r(4) \end{bmatrix} = \begin{bmatrix} 0^4 & 0^3 & 0^2 & 0^1 & 0^0 \\ 1^4 & 1^3 & 1^2 & 1^1 & 1^0 \\ -1^4 & -1^3 & -1^2 & -1^1 & -1^0 \\ 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\ -2^4 & -2^3 & -2^2 & -2^1 & -2^0 \end{bmatrix} \begin{bmatrix} r_0 \\ r_1 \\ r_2 \\ r_3 \\ r_4 \end{bmatrix}.$$

We can invert the matrix and compute the coefficients of the polynomial which gives us

$$\begin{bmatrix} r_0 \\ r_1 \\ r_2 \\ r_3 \\ r_4 \end{bmatrix} = \begin{bmatrix} 1/4 & -1/6 & -1/6 & 1/24 & 1/24 \\ 0 & -1/6 & 1/6 & 1/12 & -1/12 \\ -5/4 & 2/3 & 2/3 & -1/24 & -1/24 \\ 0 & 2/3 & -2/3 & -1/12 & 1/12 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} r(0) \\ r(1) \\ r(2) \\ r(3) \\ r(4) \end{bmatrix}.$$

Now we can solve for the coefficients for the polynomial $r(x)$. Instead of expanding everything till the end we can assume that the five $O(n/3) + 1$ digit squares that we are looking for are $r(0), r(1), r(2), r(3), r(4)$, then

$$r_0 = \frac{6r(0) - 4r(1) - 4r(2) + r(3) + r(4)}{24}$$

$$r_1 = \frac{-2r(1) + 2r(2) + r(3) - r(4)}{12}$$

$$r_2 = \frac{-30r(0) + 16r(1) + 16r(2) - r(3) - r(4)}{24}$$

$$r_3 = \frac{8r(1) - 8r(2) - r(3) + r(4)}{12}$$

$$r_4 = r(0).$$

If we substitute $x = 2^{n/3}$ in $r(x)$ we get,

$$r(2^{n/3}) = r_4 2^{4n/3} + r_3 2^n + r_2 2^{2n/3} + r_1 2^{n/3} + r_0,$$

this is nothing but a^2. If we compare the equations of a^2 and $r(2^{n/3})$ we get,

$$r_4 = a_2^2$$
$$r_3 = 2a_1 a_2$$
$$r_2 = a_1^2 + 2a_2 a_0$$
$$r_1 = 2a_1 a_0$$
$$r_0 = a_0^2.$$

8

Now we can write a^2 in terms of the five squares,

$$a_2^2 = r(0)$$

$$a_1 a_2 = \frac{8r(1) - 8r(2) - r(3) + r(4)}{24}$$

$$a_1^2 + 2a_2 a_0 = \frac{-30r(0) + 16r(1) + 16r(2) - r(3) - r(4)}{24}$$

$$a_1 a_0 = \frac{-2r(1) + 2r(2) + r(3) - r(4)}{24}$$

$$a_0^2 = \frac{6r(0) - 4r(1) - 4r(2) + r(3) + r(4)}{24}.$$

This technique of splitting terms and finding the coefficients of the squared number is very powerful because this can be used to decrease the computation of powers not only to 5 but even below. But linear run time is not achievable.

We have used to five $O(n/3) + 1$ squared terms $r(0), r(1), r(2), r(3), r(4)$ to compute the square of a number a of size n. Using the Divide-and-Conquer approach we can write the recurrence relation. We need constant time in dividing the problem in to one thirds i.e., $O(1)$. When we square a number of size $\lceil n/3 \rceil$ the size will be at most n hence at each step combining will take $O(n)$ time. At each step we divide the problem to 5 sub problems which are $1/3^{rd}$ the size of the original problem (neglecting the ceiling). Final equation will be

$$T(n) = 5T(n/3) + O(n)$$

Using master theorem we get $T(n) = O(n^{\log_3 5})$.

4. (**Counting inversions**) Given a sequence of *n distinct* numbers $a_1, \ldots, a_n$, we call (a_i, a_j) an *inversion* if $i < j$ but $a_i > a_j$. For instance, the sequence $(2, 4, 1, 3, 5)$ contains three inversions $(2, 1)$, $(4, 1)$ and $(4, 3)$.

   (a) Given an algorithm running in time $O(n \log n)$ that counts the number of inversions. (Hint: does Merge-sort help?) Can you also output all inversions?

   (b) Let's call a pair a *significant inversion* if $i < j$ and $a_i > 2a_j$. Given an $O(n \log n)$ algorithm to count the number of significant inversions.

Solution:

   (a) **Modified Merge-sort:** *We will use merge-sort and modify it such a way that it helps us in counting all possible inversions. This basically tell us how close we are from sorted array in some notion. Idea is that if we keep track inversion in left and right sub array which are sorted, then we can compute the no. of inversions required, while doing the merge step. Finally adding these three values will give us total no.of inversions present in the given array. In merge step itself we can have a additional print step that will give us all the inversion present in the graph.*

9

Hence the idea for modified merge-sort is that sort each half sub array during the recursive call, then count inversions while merging the two sorted lists (merge-and-count). In terms of Divide-and-Conquer algorithm this will be

- *Divide: separate the list into two halves A and B.*
- *Conquer: recursively count inversions in each list.*
- *Combine: count inversions (a, b) with $a \in A$ and $b \in B$.*
- *Return sum of three counts.*

The modified step here is the combine *step. While combining we will count the inversions in the following way,*

- *Assume $a \in A$ and $b \in B$ and A and B are sorted.*
- *Scan A and B from left to right.*
- *Compare a_i and b_j.*
- *If $a_i < b_j$, then a_i is not inverted with any element left in B.*
- *If $a_i > b_j$, then b_i is not inverted with any element left in A.*
- *Keep track of all the inversions and append smaller element to sorted list C.*

Algorithm: *Modified Merge-sort (on list of elements L)*
Return number of inversions in L and L in sorted order
Sort-and-Count(L, l, r):
if *List L has one element*
    return *(0, L)*
{Divide list into two halves A and B}
$mid \leftarrow (l + r)/2$
$A \leftarrow L[l, mid]$
$B \leftarrow L[mid + 1, r]$
$(r_A, A) \leftarrow Sort - and - Count(A, l, mid)$
$(r_B, B) \leftarrow Sort - and - Count(B, mid + 1, r)$
$(r_{AB}, L) \leftarrow Merge - and - Count(A, B, l, mid, r)$
return $(r_A + r_B + r_{AB}, L)$

Merge-and-count(A, B, l, mid, r)
{Assume A and B are sorted}
$p = l; q = mid + 1; idx = l; r_{AB} = 0$
while $p \leq mid$ *and* $q \leq r$
    if $A[p] \leq B[q]$
{Doesn't form an inversion pair}
        $L[idx] = A[p]$
        $p = p + 1$
    else
{Yes we have inversion pairs count them and print them}
        $L[idx] = B[q]$
        $q = q + 1$

$$r_{AB} = r_{AB} + (mid - p + 1)$$
$$\textbf{for } i = p \text{ to } mid$$
$$\quad \textbf{print } \text{Inversion pair } B[q], A[i]$$
$$idx = idx + 1$$
$$\textbf{if } p > mid$$
$$\{Copy \text{ all the leftover elements from } B \text{ to } L\}$$
$$\quad \textbf{for } i = q \text{ to } r$$
$$\quad\quad L[idx] = B[i]$$
$$\quad\quad idx = idx + 1$$
$$\textbf{else}$$
$$\{Copy \text{ all the leftover elements from } A \text{ to } L\}$$
$$\quad \textbf{for } i = p \text{ to } mid$$
$$\quad\quad L[idx] = A[i]$$
$$\quad\quad idx = idx + 1$$
$$\textbf{return } L \text{ and } r_{AB}$$

Summary of the Merge − and − Count will be first check if there is any kind of inversion by comparing elements from A and B. If yes then keep track of them and print all the inversion pairs and merge the elements into L. If no then just merge the elements L. Finally merge all the left over terms and return the count and list/array L.

Using the Divide-and-Conquer approach we can write the recurrence relation. We need constant time in dividing the problem in to half i.e., $O(1)$. When we combine we will be merging at most n elements hence at each step combining will take $O(n)$ time. At each step we divide the problem to 2 sub problems which are half the size of the original problem. Final equation will be

$$T(n) = \begin{cases} \Theta(1) & if n = 1 \\ 2T(n/2) + O(n) & otherwise \end{cases}$$

Using master theorem we get $T(n) = O(n \log n)$.

(b) *The algorithm for counting* significant inversion *will be similar to for counting the inversion except for small modification to Merge − and − Count. The difference is that we will split the merge step into two steps. First step will be for normal merging and sorting the array. Second step will be for counting the significant inversion in array B', which is constructed as $B'[i] = 2B[i]$. Below I show the pseudo code only for the modified part,*

$$\underline{\textbf{\textit{Algorithm:}} \text{ Merge-and-count}(A, B, l, mid, r)}$$
$$\{Assume \text{ A and B are sorted}\}$$
$$p = l; q = mid + 1; idx = l; r_{AB} = 0$$
$$\textbf{while } p \leq mid \text{ and } q \leq r$$
$$\quad \textbf{if } A[p] \geq 2 * B[q]$$
$$\{Yes \text{ we have inversion pairs count them and print them}\}$$
$$\quad\quad q = q + 1$$

11

$$r_{AB} = r_{AB} + (mid - p + 1)$$
for $i = p$ to mid
    print Inversion pair $B[q]$, $A[i]$
  else
{Doesn't form an inversion pair}
    $p = p + 1$
{Reset counters and do normal merge}
$p \leftarrow l$
$q \leftarrow mid + 1$
while $p \leq mid$ and $q \leq r$
  if $A[p] \leq B[q]$
    $L[idx] = A[p]$
    $p = p + 1$
  else
    $L[idx] = B[q]$
    $q = q + 1$ $idx = idx + 1$ **if** $p > mid$
{Copy all the leftover elements from B to L}
    for $i = q$ to r
      $L[idx] = B[i]$
      $idx = idx + 1$
  else
{Copy all the leftover elements from A to L}
    for $i = p$ to mid
      $L[idx] = A[i]$
      $idx = idx + 1$
return L and r_{AB}

This algorithm is just modified form of Merge-sort and we can apply analysis that we did above as it is to conclude that running time of this algorithm will be $O(n \log n)$.

5. Consider a function f that takes two integer inputs i, j in $\{1, \ldots, n\}$ and returns a real number. A *local minimum* of f is a point (i, j) such that $f(i, j) \leq f(a, b)$ for all pairs $(a, b) \in \{1, \ldots, n\}^2$ where $|a - i| \leq 1$ and $|b - j| \leq 1$. The goal of this problem is to find an efficient algorithm that finds a local minimum of f, assuming nothing about the structure of f except that for any input (i, j), evaluating $f(i, j)$ takes *unit* time.

We can represent this problem via a grid-like graph G_n, where the vertices are pairs of integers and two pairs are connected if each of their components differs by at most 1, that is $((i, j), (a, b)) \in E$ iff. $|a - i| \leq 1$ and $|b - j| \leq 1$. Note that G_n has n^2 vertices and degree at most 8. We can think of f as a function that assigns a real number to each vertex. A local minimum is then a vertex v such that $f(v) \leq f(v')$ for all adjacent vertices v'.

(a) Give a recursive algorithm for this problem that cuts the graph G_n into four sub-graphs of the form $G_{n/2}$, and gets called recursively on at most one of the subgraphs.

(b) Prove correctness of your algorithm.

(c) State a recurrence that describes the worst-case running time of your algorithm. Solve it using any method of your choice.

Solution:

(a) *Our objective in this problem is to find a local minimum of function f which takes two inputs (i,j) and $i,j \in \{1,\ldots,n\}$. We have defined local minima of f is a point (i,j) such that $f(i,j) \leq f(a,b)$ for all pairs $(a,b) \in \{1,\ldots,n\}^2$ where $|a-i| \leq 1$ and $|b-j| \leq 1$. So at each point at most we have 8 neighbors to check. As suggested formulating this problem like a grid problem will help in solving the problem. We have grid-like graph G_n of size n^2. Value of the grid at each point (i,j) is determined by the function $f(i,j)$. If given a grid-like graph G_n of size n^2*

> ***Algorithm:*** *LocalMinimaFinder(G)*
> ***1.** Take a "window frame" formed by the first, middle and last row, and first, middle and column. Next we will find the minimum element of these 6n elements $g = G[i,j] = f(i,j)$.*
> ***2.** If g is less than or equal to its neighbors, then by definition, that element is a local minimum. Return its indices (i,j) and value of $f(i,j)$.*
> ***3.** Else there's an element that neighbors g that is less than g. Note that this element can't be on the window frame since g is the minimum element on the window frame, thus this element must be in one of the four quadrants. These quadrants will be of size $G_{n/2}$.*
> ***4.** Do recursion and use this algorithm on the grid-like graph formed by that quadrant $G_{n/2}$ (not including any part of the window frame). Repeat from 2 to 4.*

Intuition behind this algorithm comes from the binary-search algorithm that we implement on a 1D array when we are trying the find a peak value, which is like finding the 'local' maxima.

(b) *Correctness of this algorithm relies on the idea that every time when we do recursion and move to a smaller problem we are step by step going in the direction the which the minimum value decreases.*

Lemma 1.1. *If we recurse on a quadrant, there is indeed a local minima in that quadrant.*

Proof. The quadrant we selected contains an element lesser than g. Thus we know that the minimum element in this quadrant must also be lesser than g. Since g is the minimum element surrounding this quadrant, the minimum element in this quadrant must be smaller than any element surrounding this quadrant. This element must be lesser than or equal to all of its neighbors since it is lesser than all elements within the quadrant and directly outside of the quadrant, so the minimum element in this quadrant must be a local minima. $\square$

Lemma 1.2. If you find a minima on the subgraph, then that minima is a local minima.

Proof. The window frame of the subgraph contains an element lesser than *g*. Say *m* is the minimum element on the window frame. Since *g* is the smallest element directly surrounding the subgraph, that means *m* is smaller than all the elements surrounding the subgraph. If *m* is a minima in the subgraph and *m* is on the boundary, *m* must be a local minima since it is guaranteed that *m* is smaller than any neighbors outside the scope of the subgraph. If *m* is a minima in the subgraph and *m* is not on the boundary, then clearly *m* is lesser than or equal to its (at most) eight neighbors and thus is a local minima. □

Lemma 1.3. We will always find a minima on the subgraph.

Proof. In the case that we don't find a minima on the window frame of a subgraph, we recurse to try to find a minima in a strictly smaller subgraph. Eventually, if we keep not finding a peak, we will recurse into a small enough subgraph such that the window frame covers the entire subgraph (i.e. if the number of rows and columns are both 3 or below). By lemma 1.2, there is indeed a local minima in this subgraph if we recursed down to it. Since we're examining the entire subgraph, we must find that local minima. □

By lemma 1.2 and lemma 1.3, using this algorithm, we will always find a minima and that minima will be a local minima.

(c) To write the recurrence relation for this problem we will follow the description provided in the problem 1. We always start with problem of size n and reduce it to subproblem of the size n/2. Here we don't any time for combining but timing because we are not doing this operation. Time division of the problem will be of the order $O(n)$ because we have to find the minimum of the 6n elements, assuming that we do this in a simple for *loop. Also note that at each recursion step even though we divide problem into 4 subproblems, we only choose one. Hence, final relation is*

$$T(n) = T(n/2) + O(n)$$
$$= T(n/2) + n.$$

This can be solve using Master's theorem as described in problem 1 (we can also use recursion tree to solve). $n^{\log_2 1} = 1$, hence this fall in the third case where the function $f(n)$ grows faster than $n^{\log_b a}$. Hence total runtime of this algorithm will be $O(n)$. Here we are assuming comparisons and calls to the function f will take unit time.

6. Given an array of temperatures $t_1, \ldots, t_n$ measured on n days, the weather service would like to compare the biggest increase in temperature to the biggest decrease in temperature over the given period. Specifically, you would like to compute two numbers: (a) $\max_{up}$, the maximum over pairs $i < j$ of $t_j - t_i$ and (b) $\max_{down}$, the

14

maximum over pairs $i < j$ of $t_i - t_j$. For example, for the array $[10, 0, 1, 2, 3, 4, -1]$, you would compute $\max_{up} = 4$ and $\max_{down} = 11$.

  (a) Give a divide and conquer algorithm that computes these numbers in $O(n)$ time. [Hint: Cut the array into two and make a recursive call on each half. It may be helpful to have the recursive calls pass up some extra information (in addition to $\max_{up}$ and $\max_{down}$).]

  (b) Give a linear-time algorithm for the same problem which makes a single pass through the data and uses only a *constant* amount of workspace (beyond the space needed to store the input).

2 Data Structures, Amortized Analysis

7. Suppose we are maintaining a data structure under a series of n operations. Let $f(i)$ denote the actual running time of the ith operation. For each of the following functions f, determine the resulting amortized cost of a single operation.

  (a) $f(i)$ is the largest integer k such that 2^k divides i.

  (b) $f(i)$ is i if i is an exact power of 2, and 1 otherwise.

Solution: **Aggregate analysis:** *In aggregate analysis, we show that for all n, a sequence of n operations takes worst-case time $T(n)$ in total. In the worst case, the average cost, or amortized cost, per operation is therefore $T(n)/n$. We use aggregate analysis for both the problems to compute the amortized cost.*

  (a) *Lets look at an example which will tell how does the $f(i)$ will change with respect to the ith operation*

i^{th} operation	Largest 2^k dividing i	$f(i) = k$
1	2^0	0
2	2^1	1
3	2^0	0
4	2^2	2
5	2^0	0
6	2^1	1
7	2^0	0
8	2^3	3
9	2^0	0
10	2^1	1
11	2^0	0
12	2^2	2
13	2^0	0
14	2^1	1
15	2^0	0
16	2^4	4
17	15 2^0	0
18	2^1	1
19	2^0	0
20	2^0	0

We observe that for n operations there are approximately $n/4$ terms with running time as 1, $n/8$ terms with running time as 2, $n/16$ terms with running time as 3 and so on, and for all odd numbers we have running time equal to zero. Without loss of generality we can assume that $n = 2m$. So total running time will be

$$\sum_{i=1}^{n} f(i) = \sum_{i=1}^{m} 0 + \frac{n}{4} \sum_{i=1}^{m} \frac{i}{2^{i-1}}$$

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

$$\leq \frac{n}{2} \sum_{i=1}^{\infty} \frac{i}{2^i}.$$

We will now evaluate the sum running to infinity.

$$S = \sum_{i=1}^{\infty} \frac{i}{2^i}$$

$$= \frac{1}{2} + \frac{2}{4} + \frac{3}{16} + \frac{4}{32} + \frac{5}{64} \cdots$$

$$S/2 = \frac{1}{4} + \frac{2}{8} + \frac{3}{32} + \frac{4}{64} + \frac{5}{128} \cdots$$

$$S - S/2 = \frac{1}{4} + \frac{1}{8} + \frac{1}{32} + \frac{1}{64} + \frac{1}{128} \cdots$$

$$S/2 = \sum_{i=1}^{\infty} \frac{1}{2^i}$$

$$S/2 = 1$$

$$S = \sum_{i=1}^{\infty} \frac{i}{2^i} = 2.$$

Hence the total running time will be

$$\sum_{i=1}^{n} f(i) \leq \frac{n}{2} \sum_{i=1}^{\infty} \frac{i}{2^i}$$

$$\leq \frac{n}{2} * 2$$

$$\leq n.$$

In worst case to do n operations we would need $O(n)$ running time. Hence average running time i.e. amortized cost will be $O(n)/n$ which is equal to constant time $O(1)$.

(b) *Lets look at an example which will tell how does the $f(i)$ will change with respect to the ith operation*

i^{th} operation	$f(i)$
1	1
2	2
3	1
4	4
5	1
6	1
7	1
8	3
9	1
10	1

Cost of an operation is given by the following equation

$$f(i) = \begin{cases} i & \text{if } i \text{ is an exact power of 2} \\ 1 & \text{otherwise.} \end{cases}$$

Now we can write the cost of n operations as

$$\sum_{i=1}^{n} f(i) = \sum_{j=0}^{\lceil \log_2 n \rceil} 2^j + \sum_{i \in J} 1,$$

here J is a set containing all the indices which are not exact power of 2 and the first summation is over all the indices which are exact power of 2. For a given n there can be only at most $\lceil \log_2 n \rceil$ terms which are exact power of 2. Now we will compute the worst case running time for above operations,

$$\begin{aligned} \sum_{i=1}^{n} f(i) &= \sum_{j=0}^{\lceil \log_2 n \rceil} 2^j + \sum_{i \in J} 1 \\ &\leq \sum_{j=0}^{\log_2 n} 2^j + n \\ &= \frac{1(2^{\log_2 n} - 1)}{2 - 1} + n \\ &= n - 1 + n \\ &= 2n - 1. \end{aligned}$$

In worst case to do n operations we would need $O(n)$ linear running time. Hence average running time i.e. amortized cost will be $O(n)/n$ which is equal to constant time $O(1)$.

3   Dynamic Programming, Greedy Algorithm

8. (**Longest forward-backward contiguous substring**) Describe and analyze an efficient
   algorithm to find the length of the longest *contiguous* substring that appears both
   forward and *backward* in an input string $T[1, \dots, n]$. The forward and backward
   substrings must NOT overlap. Here are several examples.

   - Given the input string ALGORITHM, your algorithm should return 0.
   - Given the input string REC<u>U</u>RSION, your algorithm should return 1, for the
     substring R.
   - Given the input string RED<u>IVI</u>DE, your algorithm should return 3, for the sub-
     string EDI. (The forward and backward substrings must not overlap!)
   - Given the input string D<u>YNAM</u>ICPROGRAMMING<u>MANY</u>TIMES, your algorithm should
     return 4, for the substring YNAM. (It should not return 6, for the subsequence
     YNAMIR, because it's not contiguous.).

   *Solution: **Definition:** A **substring** is a contiguous sequence of characters within a string.
   For instance, "the best of" is a substring of "It was the best of times".*

   *Note: Prefix and suffix are special cases of substring. A prefix of a string S is a substring of S
   that occurs at the beginning of S. A suffix of a string S is a substring that occurs at the end of
   S.*

   ***Definition:** A **subsequence** is a sequence that can be derived from another sequence by
   deleting some or no elements without changing the order of the remaining elements. For
   example, "It was times" is a subsequence of "It was the best of times", but not a substring.*

   *Let $T[1, \dots, n]$ be given input string, let $c[i, j]$ be the length of longest forward-backward
   contiguous substring for a string starting at i^{th} index and ending at j^{th} i.e.,*

   $$c[i, j] = |LFBCS(T[i, \dots, j])|.$$

   To develop dynamic programming algorithm for this problem we consider four cases as follows

   - ***Case 1:** If the length of the string $T[i, \dots, j]$ is 2 i.e., there are only two elements in the
     string then*
     - *If $T[i] = T[j]$ then $c[i, j] = 1$.*
     - *If $T[i] \neq T[j]$ then $c[i, j] = 0$.*

   - ***Case 2:** If $i = j$ then $c[i, i] = 0$, because we don't want to consider cases where the
     strings overlap so we make the effect induced by them in the length equal to zero. Notice
     when overlap occurs for example at index $i, i+1, i+2$ then, if $T[i] = T[i+2]$, we don't
     want the count the for $c[i, i+2]$ to be 2 but we want it to be 1 so we make $c[i, i] = 0$.*

- **_Case 3:_** _If $T[i] = T[j]$, then $c[i, j] = c[i + 1, j - 1] + 1$. Because if we have a string going from $T[i, \ldots, j]$ has the head and tail matching then, we want to add the 1 to the length of longest forward-backward contiguous substring for a string starting at $(i + 1)^{th}$ index and ending at $(j - 1)^{th}$. Which is nothing but the inner subproblem for $T[i + 1, \ldots, j - 1]$, that must be solved._

- **_Case 4:_** _If $T[i] \neq T[j]$, then $c[i, j] = max\{c[i + 1, j - 1], c[i + 1, j], c[i, j - 1]\}$. Because if the string $T[i, \ldots, j]$ has different head and tail then we want check which inner subproblem of this string will lead to maximum length of longest forward-backward contiguous substring. So we consider for the cases for all possible substrings, namely $T[i + 1, \ldots, j - 1]$, $T[i + 1, \ldots, j]$ and $T[i, \ldots, j - 1]$._

By looking at the above cases we can conclude that we will need a matrix of size in the order of $O(n^2)$, where the lower triangle of the matrix will be zero because it doesn't make sense to compare the strings in reverse order. And we will fill the matrix parallel to the direction of the principal diagonal in the upper triangular matrix. If the word doesn't many repeating letters then this matrix will be mostly sparse and filled with zeros mostly. Finally we are looking for the solution for the problem $c[1, n]$.

> **_Algorithm:_** LFBCS(T)
> _//c[i, j] memoize subproblems_
> **_for_** _i from 1 to n_
> $c[i, i] = 0$ _//principal diagonal values are zero_
> **_for_** _l form 1 to n − 1 //diagonals_
>   **_for_** _i from 1 to n − l //rows_
> $j \leftarrow i + l$ _//column of row i on l^{th} diagonal_
>   **_if_** _len$(T[i, \ldots, j]) == 2$_
>   **_if_** _$T[i] == T[j]$_
> $c[i, j] = 1$
>   **_else_**
> $c[i, j] = 0$
>   **_if_** _$T[i] == T[j]$_
> $c[i, j] = c[i + 1, j - 1] + 1$
>   **_else_**
> $c[i, j] = max\{c[i + 1, j - 1], c[i + 1, j], c[i, j - 1]\}$

As we can see from the above pseudo code that this algorithm has two for loops so it should take $O(n^2)$ running time at worst case, where n is the total number of letters (characters) in the given word (string). To extract the string we can keep track where the maximum encountered then back track to position where one was encountered.

9. (**Burrito-Delivery**) You've just accepted a job from Elon Musk, delivering burritos from San Francisco to Houston. You get to drive a Burrito-Delivery Vehicle through Elon's new _Transcontinental Underground Burrito-Delivery Tube_, which runs in a direct line between these two cities. Your Burrito-Delivery Vehicle runs on single-use batteries, which must be replaced after at most 100 miles. The actual fuel is virtually

free, but the batteries are expensive and fragile, and therefore must be installed only by official members of the Transcontinental Underground Burrito-Delivery Vehicle Battery-Replacement Technicians' Union. Thus, even if you replace your battery early, you must still pay full price for each new battery to be installed. Moreover, your Vehicle is too small to carry more than one battery at a time.

There are several fueling stations along the Tube; each station charges a different price for installing a new battery. Before you start your trip, you carefully print the Wikipedia page listing the locations and prices of every fueling station along the Tube. Given this information, how do you decide the best places to stop for fuel?

More formally, suppose you are given two arrays $D[1, \ldots, n]$ and $C[1, \ldots, n]$, where $D[i]$ is the distance from the start of the Tube to the ith station, and $C[i]$ is the cost to replace your battery at the ith station. Assume that your trip starts and ends at fueling stations (so $D[1] = 0$ and $D[n]$ is the total length of your trip), and that your car starts with an empty battery (so you must install a new battery at station 1).

(a) Describe and analyze a greedy algorithm to find the *minimum number* of refueling stops needed to complete your trip. Don't forget to prove that your algorithm is correct.

(b) But what you really want to minimize is the *total cost* of travel. Show that your greedy algorithm in part (a) does not produce an optimal solution when extended to this setting.

(c) Sketch an efficient algorithm to compute the locations of the fuel stations you should stop at to minimize the total cost of travel. You can use any algorithm we've discussed so far.

Solution:

(a) Greedy strategy for this problem would be to pick the refueling stop only when we need to change the battery. Hence this will lead us to go far as possible to before we hit next low fuel scenario which will make us to take a stop for refueling. So, if we always next stop in such a way that we get as close to the end point as possible, then we will end up taking minimum number of refueling stops to complete the trip.

> ***Algorithm:*** *RFStops(D)*
> $A \leftarrow 1$ *// set of refueling locations*
> ***for*** $j = 2, \ldots, n$
>   *if j compatible with A*
> $A \leftarrow A \cup \{j\}$

Here compatibility indicates that we pick a location that is farthest away from the previous stop but lies within 100 miles away from it, so that we can refuel and we keep on doing that until we reach the end point. Let x be the last term in the set A then we can find the next compatible location by finding inserting the value $D[x] + 100$ into an array and finding the term which is just below it, and this cane be done in constant time by efficient data structures like Fibonacci Heap. Hence running time of this algorithm will be $O(n)$

(linear). If we use data structures like priority queue or binary heap then the running time would be $O(n \log n)$ in the worst case, this running time would be an example of quasi-linear.

Proof of correctness: *Suppose greedy is not optimal. Then consider an optimal strategy: one that agrees with Greedy for as many initial refueling stops as possible. Look at the first place that they differ: and then we will show a new optimal solution that agrees with greedy more than the expected. We will prove this by contradiction. Suppose greedy is not optimal then*

- *Let $i_1, i_2, \ldots, i_k$ denotes set of refueling stops selected by greedy.*
- *Let $j_1, j_2, \ldots, j_m$ be set of refueling stops in the optimal solution OPT where $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ for the largest possible value r.*
- *Substitute i_{r+1} for j_{r+1} in OPT and it will be still feasible and optimal. Because we are bound to have a refueling stop between current stop and next stop within 100 miles. So, if j_{r+1} occurs at $x \leq 100$ miles from current location and i_{r+1} occurs at $y \leq 100$ miles, then we can switch j_{r+1} with i_{r+1} without changing the optimal solution and feasibility (its like assuming that we have a equal cost for stopping at all points). But we assumed that Greedy solution agrees till r at most but we have shown that it agrees till $r + 1$ which contradicts the maximality of r.*

Hence greedy solution is indeed optimal.

(b) *Let's take stops such that $D = [0, 50, 75, 100, 150]$ and $C = [1, 3, 10, 1000, 5]$. Greedy solution will pick stops at $A = \{1, 4, 5\}$ and the cost incurred will be 1006 and if we use a better method then we will get optimal solution as $A = \{1, 2, 5\}$ and optimal cost incurred will be 9. Hence, by proof by counter-example we can show that using the same greedy algorithm and extending it to find minimum cost will not work.*

(c) *We will apply dynamic programming to compute the locations of the fuel stations to minimize the total cost of travel. Consider an optimal solution for a given input instance; in this solution, we either place a refueling stop at site n or not. If we don't, the optimal solution on sites $1, \ldots, n$ is really the same as the optimal solution $1, \ldots, n - 1$; if we do, then we should eliminate d_n and all other sites that are not more than 100 miles away from it, and find an optimal solution on what's left. The same reasoning applies when we're looking at the problem defined by just the first j sites, $1, \ldots, j$: we either include j in the optimal solution or we don't, with the same consequences. Let's define some notation to help express this. For a site j, we let $p(j)$ denote the all possible site starting that are at most 100 miles away from j but no more than that. All sites that are more than 100 miles farther than j are discarded. Whatever site we choose from the set $p(j)$ we want to make sure that we choose the one which gives us the least cost. Now, our reasoning above justifies the following recurrence. If we let $OPT(j)$ denote the cost from the optimal subset of sites among $1, \ldots, j$, then we have*

$$OPT(j) = \begin{cases} c_1 & \text{if } j = 1 \\ min\left(c_j + min_{k \in p(j)}OPT(k), OPT(j-1)\right) & \text{otherwise.} \end{cases}$$

We are looking for the solution for $OPT(n)$. If we are given the two arrays D and C with values arranged in ascending order based on distances then

> **Algorithm:** RFStopsMinCost(D, C)
> *Initialize $M[1] = c_1$*
> **for** $j = 2, \ldots, n$
> $M[j] \leftarrow \infty$
> $k \leftarrow j + 1$
>   **while** $0 \leq D[k] - D[j] \leq 100$ // $k \in p(j)$
> $M[j] = min(c_j + M(k), M[j])$
> $k \leftarrow k + 1$
> $M[j] = min(M[j], M(j-1))$

In worst case this algorithm can take $O(n^2)$ running time because for each refueling site j, all the sites $1, \ldots, j$ can be present in the set $p(j)$ hence at each step we need to iterate over j indices hence in total we need $O(n^2)$ time. This gives us the optimal solution for minimum total cost required to reach the destination. Also, we can back track the array M to find out the optimal location i.e. where to place the rest stops.

Note: If we were give that the distance between the sites must at least 20 (or some constant c, it doesn't matter) then the while loop will run for constant time because there will be at most 5 stops that we need to check hence total runtime will be of the order $O(n)$.

10. (**Stabbing points**) Let X be a set of n intervals on the real line. We say that a set P of points *stabs* X if every interval in X contains at least one point in P. Describe and analyze an efficient algorithm to compute the smallest set of points that stabs X. Assume that your input consists of two arrays $L[1, \ldots, n]$ and $R[1, \ldots, n]$, representing the left and right endpoints of the intervals in X. (N.B. If you use a greedy algorithm, you must prove its correctness.)

Solution: Solution of this problem will rely on the following claim.

<u>Claim:</u> For any set of intervals there exist a stabbing set of minimum size where every stabbing point is the right end point of some interval (an element from array $R[1, \ldots, n]$).

<u>Proof:</u> To prove the above claim we show that if we are given any stabbing set of that does not satisfy this property, then it can be converted to one that has the equal or lower cardinality stabbing set which satisfies this property. Which will lead us to new stabbing set of lower size or size remains the same. Hence, if we claim to have minimum size stabbing set that doesn't above property either we can get new set with lower size or of the same size.
Consider any stabbing set $X = \{x_1, \ldots, x_k\}$. If it satisfies the right end property then we are done. If not, let x_i be the smallest end point that is not right end point of some interval. Let $R[j]$ be the smallest right end point that is greater than x_i. Replace x_i with $R[j]$. After replacing we still get a valid stabbing set, because x_i stabs a set of intervals I hence it must lie between their left and right end points. Now we are replacing x_i with closest right end point $R[j]$, which means that $R[j]$ will still lie on the right side of all left end points of I. Since,

$R[j]$ is the closest right end point to x_j so it lies to on the left side of all right end points of I. Hence, $R[j]$ lies between all left and right end points of I.

Conclusion as we have increased x_i to a value that is not greater than any of the right endpoints stabbed by x_i, every interval stabbed by x_i is also stabbed by $R[j]$. Thus, the modified set is also a stabbing set. After the replacement, the cardinality of the stabbing set is either the same, or it decreases by one (if $R[j]$ was in the original stabbing set). After doing this at most k times, X will be converted into a stabbing set of equal or lower cardinality, and all of its endpoints are all right endpoints of some interval, as desired.

Greedy Algorithm: *Using above claim we will solve this problem by implementing greedy algorithm which is similar to the interval scheduling problem taught in the class. Our greedy strategy is to arrange the array $R[1, \ldots, n]$ in increasing order, then we will pick the first right end point as a stabbing point and add it into the set P and remove all the intervals that are stabbed by it. We keep on doing this procedure until we are left no intervals to stab. Here deletion of stabbed intervals means that we skip over them. In the interval scheduling problem we had step where we skipped over the jobs that were not compatible to the current one, we are doing the same thing here. Every time when we add new stabbing point into the set we keep a counter to see the cardinality of the set P. All the points in set P are right end points of different intervals.*

Proof of Correctness: *To prove that above algorithm will work, we will use greedy exchange argument. Sketch of the proof is proof by contradiction as follows*

- *Suppose greedy is not optimal.*

- *Consider an optimal strategy, one that agrees with greedy for as many initial intervals as possible.*

- *Look at the first place that they differ, show a new optimal that agrees with greedy more. Hence by contradiction greedy solutions agrees with optimal solution at all locations hence they are same.*

Let $G = \{i_1, \ldots, i_m\}$ denote the stabbing set generated by above greedy algorithm and let $O = \{j_1, \ldots, j_m\}$ optimal stabbing set. From above claim we may assume that O consists of only of right end points. Let us assume that the elements of both G and O are sorted in increasing order.

If $G = O$, then we are done. Otherwise, let j_r denote the first element where G and O differ, and let i_r denote the corresponding element of G. First, observe that because i_r intersects the leftmost right endpoint of an interval that is not stabbed by one of the earlier elements of the stabbing set, we can infer that $j_r < i_r$ (since otherwise the optimum would not stab this interval, contradicting the fact that it is a stabbing set). We assert that any interval stabbed by j_r is stabbed by either i_r or one of its predecessors. Consider any interval $[L[p], R[p]]$ that is stabbed by j_r. If it is not stabbed by one of i_r's predecessors then by definition of greedy $i_r \leq R[p]$. (Greedy selects the leftmost right endpoint that is not already stabbed.) Also, since the interval contains j_r, we have $L[p] \leq j_r$. In conclusion, we have $L[p] \leq j_r \leq i_r \leq R[p]$, implying that i_r stabs this interval, as desired.

Now, let us define a new set O' by replacing j_r with i_r in O. The resulting set is still a stabbing set, since every interval stabbed by j_r is also stabbed by i_r or one of its predecessors. Clearly,

$|O'| \leq |O|$, implying that O' is also optimal. After repeating this a sufficient number of times, we will convert O into G, without increasing the size of the stabbing set.

> *__Algorithm:__ GreedyStab(L, R)*
> *__Sort__ the array of right end points of the interval in increasing order*
> $P \leftarrow \phi$
> *prev_rep* $= -\infty$
> *__for__ $i = 1, \ldots, n$*
>   *__if__ $L[i] >$ prev_rep*
>   *__append__ right end point to P*
>   *prev_rep* $= R[i]$
> *__return__ P*

The running time is dominated by the $O(n \log n)$ time needed to sort the jobs by their finish times. After sorting, the remaining steps can be performed in $O(n)$ time.

11. Suppose you are given a set L of n line segments in the plane, where each segment has one endpoint on the vertical line $x = 0$ and one endpoint on the vertical line $x = 1$, and all $2n$ endpoints are distinct. Describe and analyze an algorithm to compute the largest subset of L in which no pair of segments intersects.

__Solution: Algorithm based on Dynamic Programming:__
Based on the assumption that all endpoints are unique we can write a dynamic programming equation as

$$OPT(L) = maxOPT(L \setminus l_i), 1 + OPT(L \setminus Intersection(l_i)),$$

where $OPT(P)$ is largest subset of non-intersecting segments of L, $Intersection(l_i)$ is the set of segments in L intersecting with l_i. However, in order to use the dynamic programming approach, it is not clear how the subproblems overlap and in what order should we evaluate them. An alternative approach is to formulate the problem as a graph problem: each segment l_i correspond to a vertex in the graph, and two vertices are connected by an edge if the segment corresponding to them intersect. Then the problem is equivalent to finding a maximum independent set in the graph. However, only when the graph instance has some special properties, we can compute the maximum independent set in polynomial time easily (via dynamic programming); for example, when the underlying graph is a tree. But the graph representing the intersections of segments clearly does not have such a property.

To use dynamic programming we make key observation that suppose we renumber the indices of left endpoints of segments $l_i, i \in \{1, \ldots, n\}$, $\{a_1, \ldots, a_n\}$ on $x = 0$, so that they are ordered from left to right, and we also renumber the corresponding indices of b_i's which are right endpoints of of segments $l_i, i \in \{1, \ldots, n\}$, $\{b_1, \ldots, b_n\}$ on $x = 1$ (so when we change the index of some a_k to a_i, we also change b_k to b_i). Then we looked at the indices of b_i's. The key observation is that a maximum non-intersecting set of segments correspond to the longest increasing subsequence. The longest increasing subsequence problem is closely

related to the longest common subsequence problem, which has a quadratic time dynamic programming solution: the longest increasing subsequence of a sequence S is the longest common subsequence of S and T, where T is the result of sorting S. This takes $O(n^2)$ running time. However, for the special case in which the input is a permutation of the integers 1, 2, ..., n, this approach can be made much more efficient, leading to time bounds of the form $O(n \log \log n)$. We can also solve the longest increasing subsequence problem efficiently with arrays and binary searching. It processes the sequence elements in order, maintaining the longest increasing subsequence found so far. Because the algorithm performs a single binary search per sequence element, its total time can be expressed as $O(n \log n)$. There are more faster algorithms for this problem. Hence, we can find the largest subset on non-intersecting segments in $O(n \log n)$ time using the idea of longest increasing subsequence.

12. You have reached the inevitable point in the semester where it is no longer possible to finish all of your assigned work without pulling at least a few all-nighters. The problem is that pulling successive all-nighters will burn you out, so you need to pace yourself (or something). Let's model the situation as follows. There are n days left in the semester. For simplicity, let's say you are taking one class, there are no weekends, there is an assignment due every single day until the end of the semester (would you like that?), and you will only work on an assignment the day before it is due. For each day i, you know two positive integers:

   - *Score*$[i]$ is the score you will earn on the ith assignment if you do not pull an all-nighter the night before.
   - *Bonus*$[i]$ is the number of additional points you could potentially earn if you do pull an all-nighter the night before.

   However, pulling multiple all-nighters in a row has a price. If you turn in the ith assignment immediately after pulling k consecutive all-nighters, your actual score for that assignment will be $(Score[i] + Bonus[i])/2^{k-1}$. Design and analyze an algorithm that computes the maximum total score you can achieve, given the arrays $Score[1, \ldots, n]$ and $Bonus[1, \ldots, n]$ as input.

4   Graph Theory

13. (**Cycles**) Give an algorithm to detect whether a given undirected graph contains a cycle. If the graph contains a cycle, then your algorithm should output one (not all cycles, just one of them). The running time of your algorithm should be $O(m + n)$ for a graph with n nodes and m edges.

    Solution: Without loss of generality, assume the graph is connected (if not, do the following for each connected component).

    Pick an arbitrary vertex as root and do a DFS, while maintaining a DFS tree. The DFS tree initially contains the root and at each point, when you encounter an edge to a new vertex, the

new vertex is added to the tree and the edge is added between the new vertex and the parent. If at any point you encounter an edge from a node to a seen vertex, then there is a cycle in the graph, and you can return the cycle by tracing the path from each end point of the latest edge backwards in the DFS tree till they meet (linear time), and displaying it in the correct order. If the DFS ends without incident, there is no cycle. DFS runs in linear time on the edges/vertices, as each edge/vertex is visited only once.

__Note:__ In an undirected graph, the edge to the parent of a node should not be counted as a back edge, but finding any other already visited vertex will indicate a back edge ([source](#)).

14. (**Shortest cycles containing a given edge**) Give an algorithm that takes as input an undirected graph $G = (V, E)$ and an edge $e \in E$, and outputs a shortest cycle that contains e (if no cycle containing e exists, the algorithm should output "none"). (Note: Give as efficient an algorithm as you can.)

__Solution:__ __Special property of BFS:__ This problem is about finding shortest cycle in graph $G = (V, E)$ containing an edge $e \in E$. If no cycle containing e exists the algorithm should output NONE.

Our main idea behind solving this problem is that the shortest cycle containing an given edge (which means that we are given two fixed points/vertices/nodes on the graph) will be the shortest path possible between two the vertices plus the edge. Let edge $e = (a, b)$, joining vertices a, b, then shortest cycle S will be,

$$S(e) = S(a, b) = \text{Shortest path between } (a, b) + \text{Edge } e.$$

Now our problem has been simplified to finding the shortest path between the given vertices a, b joined by the edge e. To solve the problem of finding shortest path can be done using BFS ([Chapter 4 of Algorithms by Dasgupta](#)).

Let an edge joining two vertices u and v be $e = (u, v) \in E$. Now we need to find a shortest cycle containing it. First we remove the edge e from the given graph $G = (V, E)$. After removing the edge e from $G = (V, E)$ we get new graph $G' = (V, E \setminus e)$. Now we run the BFS algorithm on the new graph $G' = (V, E \setminus e)$ starting from u and find the shortest path from u to v. If we find such path between u and v then the shortest cycle will be shortest path plus the edge that we removed. The total distance of this cycle will be $d(u, v) + 1$, where $d(u, v)$ is length of the shortest path between u and v and additional one is for the edge e between u and v (assuming the graph has unit weights for all edges). If we fail to find an shortest path between u and v then the edge e is the only way to go from vertex u to vertex v; which means that there is no shortest cycle containing edge e.

The assumption of all edges weights equal to one is important because for connected undirected graph G the algorithm for computing the shortest path will vary based upon the assumption on the weights of the edges ([source](#)). We might need to use different algorithms like Dijkstra's algorithm (if weights are non-negative), Bellman-Ford algorithm (if weights can be negative). But luckily in the case of undirected graph with all edge weights equal to one, the shortest path trees coincide with breath-first search (BFS) trees. We can state the theorem for shortest path tree for BFS algorithm as follows,

__Theorem:__ The BFS algorithm

- *visits all and only nodes reachable from source node s,*
- *for all nodes v sets $d(v)$ to the shortest path distance from s to v,*
- *sets parent variables to form a shortest path tree.*

The above theorem has been taken from Chapter 8 of Algorithms by Jeff Erickson.

__Algorithm:__ Modified BFS (on $G = (V, E)$ and edge $e = (a, b) \in E$)
Return the shortest cycle S such that it contains e and $min(S_i(e))$ i.e., distance wise minimum of all possible cycles
__ModBFS__(G, e):
__Construct__ Graph $G' = (V, E \setminus e)$ (graph G with edge e removed)
parent, d = BFS(G', a)
__if__ parent$[b]$ = nil
    __return__ NONE (Graph G has no shortest cycle containing edge e)
__print__ Shortest cycle starting from b to a
__print__ Node b to
temp $\leftarrow$ b
__while__ parent$[temp]$!= s
    __print__ Node parent$[temp]$ to
    temp $\leftarrow$ parent$[temp]$
__print__ Finally edge e which connects the node a and b
__print__ Total distance we need to travel is $d[b] + 1$

Do BFS on the given graph G and source node s
__BFS__(G, s):
__Mark__ all nodes in $v \in V$ as unvisited
parent$[v] \leftarrow$ nil
$d[v] = \infty, \forall v \in V$
__Mark__ source node s as visited
parent$[s] = s$
$d[s] = 0$
Let Q be a queue, Q.enqueue(s)
__while__ Q not empty
    u = Q.dequeue
    __for__ all neighbours v of u in graph G
        __if__ v is not visited
            Mark v as visited
            parent$[v] = u$
            $d[v] = d[u] + 1$
__return__ parent, d

Running time for this algorithm will $O(n + m)$ where n is total no. of nodes in the graph and m is total no. of edges in the graph, since the major computation time will be spent on doing

the BFS on the graph G′. Hence running time for finding the shortest cycle for given edge e in graph G will be linear in time.

15. (**Component graph**) Given a directed graph $G = (V, E)$, we define another graph $G^{SCC} = (V^{SCC}, E^{SCC})$ called the *component graph* as follows. Suppose that G has strongly connected components $C_1, C_2, \ldots, C_k$. The vertex set V^{SCC} is $\{v_1, \ldots, v_k\}$ where $v_i \in C_i$. There is an edge $(v_i, v_j) \in E^{SCC}$ if G contains a directed edge $x \to y$ for some $x \in C_i$ and some $y \in C_j$. Alternatively, imagine contracting all edges whose incident vertices are within the same strongly connected component of G, and the resulting graph will be G^{SCC}.

   (a) Prove or disapprove that G^{SCC} is a DAG.

   (b) Give an $O(|V| + |E|)$-time algorithm to compute the component graph of a directed graph $G = (V, E)$.

Solution:

(a) **TRUE** *G^{SCC} is a DAG.*

   Proof: *We can prove this by contradiction. Assume that G^{SCC} has a cycle. Let C and C′ be distinct strongly connected components in a directed graph $G = (V, E)$, let $u, v \in C$, let $u′, v′ \in C′$, and suppose that G contains a path $u \rightsquigarrow u′$ and $v \rightsquigarrow v′$. Then graph G contains paths $u \rightsquigarrow u′ \rightsquigarrow v′$ (because $u′, v′$ are part of a strongly connected component and by definition every vertex is mutually reachable in it) and $v′ \rightsquigarrow v \rightsquigarrow u$ (similar to previous reasoning). Thus, u and v′ are reachable from each other, thereby contradicting the assumption that C and C′ are distinct strongly connected components.* □

   Hence we conclude that,

   - *G^{SCC} is DAG.*
   - *Let C and C′ be distinct strongly connected components in a directed graph $G = (V, E)$, let $u, v \in C$, let $u′, v′ \in C′$ and suppose that G contains a path $u \rightsquigarrow u′$. Then G cannot also contain a path $v′ \rightsquigarrow v$. This statement is direct implication of the above property. If this was false then we end up having a cycle in the G^{SCC} graph. Or one can say we C and C′ are not two distinct and strongly connected components and they can be merged into one.*

(b)   *Algorithm:* Strongly-Connected-Components(G)
   1 - call $DFS(G)$ to compute the finishing times u.f for each vertex u
   2 - compute G^T
   3 - call $DFS(G^T)$, but in the main loop of DFS, consider the vertices in order of decreasing u.f (as computed in line 1)
   4 - output the vertices of each tree in the depth first forest formed in line 3 as a separate strongly connected components.

   Running time: *is $O(V + E)$ because line 1-3 takes $\Theta(V + E)$ and line 4 takes $O(V + E)$.*

28

Lets define a graph $G = (V, E)$ and its transpose will be $G^T = (V, E^T)$ where $E^T = (u, v) : (v, u) \in E$. That is, E^T consists of edges of G with their direction reversed. Given an adjacency list representation of G, the time to create G^T will be $O(V + E)$. Also note that G and G^T will have same strongly connected components. We will also extend the notion for discovery and finishing times to set of vertices. If $U \subseteq V$, then we define $d(U) = min_{u \in U}\{u.d\}$ and $f(U) = max_{u \in U}\{u.f\}$. That is $d(U)$ and $f(U)$ are the earliest discovery time and latest finishing time, respectively of any vertex in U. Reason behind by the algorithm works relies on following lemmas,

- ***Lemma 1:** Let C and C' be distinct strongly connected components in directed graph $G = (V, E)$. Suppose there is a edge $(u, v) \in E$, where $u \in C$ and $v \in C'$. Then $f(C) > f(C')$.*

- ***Lemma 2:** Let C and C' be distinct strongly connected components in directed graph $G = (V, E)$. Suppose there is a edge $(u, v) \in E^T$, where $u \in C$ and $v \in C'$. Then $f(C) < f(C')$.*

Lemma 2 tell us that each edge in G^T that goes between different strongly connected components goes from a component with an earlier finishing time (in the first depth-first search) to a component with a later finishing time.

From above, lemma 2 provides the key to understanding why the strongly connected components algorithm works. Let us examine what happens when we perform the second depth-first search, which is on G^T. We start with the strongly connected component C whose finishing time $f(C)$ is maximum. The search starts from some vertex $x \in C$, and it visits all vertices in C. By lemma 2, G^T contains no edges from C to any other strongly connected component, and so the search from x will not visit vertices in any other component. Thus, the tree rooted at x contains exactly the vertices of C. Having completed visiting all vertices in C, the search in line 3 (of the pseudo code) selects a vertex as a root from some other strongly connected component C' whose finishing time $f(C')$ is maximum over all components other than C. Again, the search will visit all vertices in C', but by lemma 2, the only edges in G^T from C' to any other component must be to C, which we have already visited. In general, when the depth-first search of G^T in line 3 visits any strongly connected component, any edges out of that component must be to components that the search already visited. Each depth-first tree, therefore, will be exactly one strongly connected component.

To get the component graph we will execute the $Strongly - Connected - Components(G)$ as described above, and then assign each node a value in $[1, k]$, if we the generate k strongly connected components. Then Traversing each node i, for each node j of $Adj[i]$, if $k[i]$ and $k[j]$ have no edges before, then we will add an edge. This process will take constant time hence entire process will take $O(V + E)$ runtime.

16. (**Singly connected graph**) A directed graph $G = (V, E)$ is *singly* connected if G contains at most one simple (i.e. no vertex repeated) path from u to v for all vertices $u, v \in V$. Give an efficient algorithm to determine whether or not a directed graph is singly connected.

Solution: To solve this problem we need to notice that if we DFS on a directed graph then graph will be singly connected if and only if all edges are tree edges or back edges. We will fix a vertex u and do DFS on $G = (V, E)$ starting from u will generate a tree T rooted at x, each vertex of the tree is reachable from u in the graph G. Other vertices that are not in the tree are not reachable from u in the graph G. An edge $(u, v) \in E \setminus E(T)$, $E(T)$ are edges of the tree, where $\{u, v\} \subseteq V(T)$, $V(T)$ are vertices of the tree, belongs to one of the following cases,

- A back edge (which is not a problem)

- A forward edge $\implies$ Graph is not singly connected (since there are at least two paths from u to v)

- A cross edge $\implies$ Graph is not singly connected (since there are at least two paths from u to v).

This idea gives us an algorithm to do a DFS on every every vertex and check if for any cross edges and forward edges. This algorithm will take $O(V(V + E))$, since generally we have more edges than vertices running time will become $O(EV)$.

Prof. Samir Khuller in 1999 proposed an improvement in algorithm. Based on the following theorem.

Theorem: Let H be a strongly connected graph. H is not singly connected if and only if at least one of the following conditions holds. The DFS search in H either yields a cross edge, or a forward edge, or a vertex v such that from the subtree rooted at v, there are at least two back edges to proper ancestors of v.

Based on this theorem we can generate an algorithm. For given directed graph G create a component graph G^{SCC} then, if we pick any two vertices such that it is in one of the strongly connected components then we can design an algorithm with run time $O(E)$ such that it checks for all the conditions. If the two vertices lie in two distinct strongly connected components then we can do DFS on the component graph from each vertex and this take $O(V)$ for each vertex and for all vertices it will take at most $O(V^2)$. Hence total runtime of the this algorithm will be $O((V + E) + (E) + (V^2))$, because linear time for generating the component graph plus for checking the conditions on the strongly connected components plus for vertices in two unique strongly connected components. Hence total runtime will be dominated by the $O(V^2)$ term.

17. (**Semi-connected graphs**) A directed graph G is *semi-connected* if, for every pair of vertices u and v, either u is reachable from v or v is reachable from u (or both).

    (a) Give an example of a DAG with a unique source (a source is a vertex with no entering edges) that is **not** semi-connected.

    (b) Describe and analyze an algorithm to determine whether a given DAG is semi-connected.

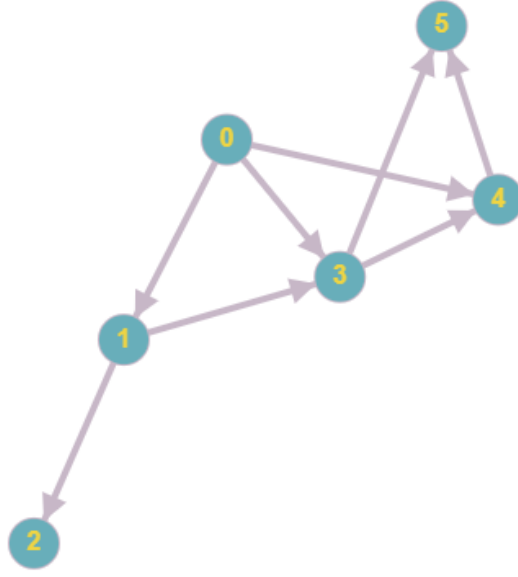   (c) Describe and analyze an algorithm to determine whether an arbitrary directed graph is semi-connected.

Figure 1: Example of a DAG that is not semi-connected graph

Solution:

(a) *We can see a example for DAG that is not semi-connected in **Fig 1**. In that figure we have one unique source vertex which is vertex 0 (it has no entering edges) and we have sinks (no outgoing edges) vertex 2 and vertex 5. We notice that there is no connection between vertices 2 and 3, 0 and 5. But according to the definition of semi-connected graph, each pair of vertices must have path between them but here we have vertex pairs which don't have edge joining them. Hence this graph is not semi-connected.*

(b) ***Definition:** a **Hamiltonian path** (or traceable path) is a path in an undirected or directed graph that visits each vertex exactly once. A **Hamiltonian cycle** (or Hamiltonian circuit) is a Hamiltonian path that is a cycle. Determining whether such paths and cycles exist in graphs is the Hamiltonian path problem, which is NP-complete.*

Now we will define a necessary and sufficient condition for DAG to be semi-connected.

***Theorem:** A DAG is semi-connected if and only if the topological sort of the DAG has a single path that goes through all the vertices i.e., there exists a Hamiltonian path. This is equivalent to saying that after we do topological sort on DAG and arrange them in linear order then there must exit an edge between all the consecutive pairs.*

***Proof:** Let there be k vertices of the DAG G arranged in the topological order $v_1, v_2, \ldots, v_k$.*

$\implies$: *Given the DAG arranged in topological order is semi-connected then for any i, there cannot be a path from v_{i+1} to v_i because we have assumed that the vertices are arranged in topological order. The only possible path from v_i and v_{i+1} is a direct edge between them and this edge must be present between them for DAG to be semi-connected. Since there are edges between all consecutive pairs of vertices in the topological order, there is a path through all vertices. Hence there must be a Hamiltonian path.*

31

$\Longleftarrow$: *Given DAG arranged in topological order has a Hamiltonian path then there is a path that goes through all vertices. That means that there is path between any pair of vertices v_i, v_j by simply following the relevant part of this global path. Hence the DAG must be semi-connected.*

<u>Observation:</u> *If a DAG is semi-connected then the graph must have one source in topological sort. If it has two source then we have two vertices with no path between which implies the graph is not semi-connected.*

Let's try to understand by theorem using an example, assume we have a DAG with 5 vertices arranged in topological order. If this DAG is semi-connected then for each pair of vertex v_i, v_j there must be edge. But since they have been arranged in topological order and there can be edges from left to right only. Which means that there can be edge from v_i to v_j where $i < j$ but there can't be an edge from v_j to v_i. If the graph is semi-connected then v_1 must have edges going to $v_2, \ldots, v_5$ (to satisfy the property of semi-connectedness), v_2 must have edges $v_3, \ldots, v_5$ (to satisfy the property of semi-connectedness) and so on. This leads to existence of edge between all consecutive vertices. If we can find such kind of a path then we have path which goes from all vertices, which is nothing but a Hamiltonian path.

Above theorem gives an skeleton for an algorithm. First we will do topological sorting on DAG and then we will check if there are edges between each consecutive pairs. If yes, then DAG is semi-connected, if no, then DAG is not semi-connected.

> **<u>Algorithm:</u>** Semi-connected(G)
> *Do topological sort on graph G*
> **if** *there exist a edge between all consecutive vertices v_i, v_j*
>   **return** *Graph is semi-connected*
> **return** *Graph is not semi-connected*

Time complexity of this algorithm is $O(V + E)$ because time taken for topological sorting on the DAG is linear time and checking the existence of an edge between all the pairs can be done in constant time.

(c) *If we are given an arbitrary directed graph, then to check if the given graph is semi-connected we can convert the original graph into a component graph. We are doing this because as proved in previous problem, a component graph of a directed graph is a DAG so, we can apply above theorem (and algorithm) on the component graph to check the semi-connectedness of the given directed graph.*

Intuition behind why this works is, if the component graph is semi-connected then it implies that there exist a path between all distinct strongly connected component. So there will be path for any vertex from one strongly connected component to another (from the definition of semi-connectedness there is no necessity on existence of path in backwards direction; we just need a path in one direction) and vertices within a strongly connected component are mutually reachable hence the original graph is semi-connected.

> **<u>Algorithm:</u>** Semi-connected(G)
> *Compute the component graph of G, call it G^{SCC}*

Perform topological sort on G^{SCC} to get the ordering of its vertices $v_1, v_2, \ldots, v_k$
 for $i = 1, \ldots, k-1$
  if *there is no edge from v_i to v_{i+1}*
   return *G is not semi-connected*
 return *G is semi-connected*

Time complexity of this algorithm will be of order $O(V + E)$ because to compute topological ordering and component graph both take linear time in the order of $O(V + E)$.

18. (**Euler tour**) An *Euler tour* of a strongly connected, directed graph $G = (V, E)$ is a cycle that traverses each *edge* of G exactly once, although it may visit a vertex more than once.

 (a) Show that G has an Euler tour if and only if in-degree(v) = out-degree(v) for each vertex $v \in V$.

 (b) Describe an $O(|E|)$-time algorithm to find an Euler tour of G if one exists.

Solution:

 (a) **Definition:** *A* **path** *or* **walk** *in a graph is a sequence of adjacent edges, such that consecutive edges meet at shared vertices. A path that begins and ends on the same vertex is called a* **cycle** *or* **circuit**. *Hence a path in a graph is a sequence of vertices such that every vertex in the sequence is adjacent to vertices before and after in the sequence.*

 Note: A cycle is a simple cycle in which the only repeated vertices are the first and last vertices.

 Definition: *A graph is said to be* **connected** *if any two of its vertices are joined by a path. A graph that is not connected is a* **disconnected** *graph. A disconnected graph is made up of connected subgraphs that are called components.*

 Definition: *An* **Euler path** *in a graph G is a path that includes* every *edge in G; an* **Euler cycle** *is a cycle that includes every edge. Euler cycle is also know with different names like* Euler tour *and* Euler circuit. *Here the starting and ending vertices are the same, so if we trace along every edge exactly once and end up where we started, it becomes a Euler cycle.*

 Note: If a graph is not connected, there is no hope of finding such a path or circuit.

 Definition: *In graph theory, the* **degree** *or* **valency** *of a vertex of a graph is the number of edges that are incident to the vertex, and in a multigraph, loops are counted twice. The degree of a vertex v is denoted $deg(v)$. For a directed graph degree of vertex will be two kinds. First one in-degree i.e., the total no. of the incoming edges and second one out-degree i.e., the total no. of the outgoing edges.*

 Handshaking lemma: *The degree sum formula states that, given a graph $G = (E, V)$ (for a graph with vertex set V and edge set E),*

$$\sum_{v \in V} deg(v) = 2|E|.$$

Note: Since $|E|$ has to be an integer that implies any graph G always has even number of vertices with odd degree.

Theorem: *A connected undirected graph $G = (E, V)$ has Euler cycle if and only if for all vertex $v \in V$ has even degree. For directed graph this theorem will be modified as follows, a connected directed graph $G = (E, V)$ has Euler cycle if and only if for all vertex $v \in V$ has in-degree(v) = out-degree(v).*

This theorem, with its "if and only if" clause, makes two statements. One statement is that if every vertex of a connected graph has an even degree then it contains an Euler cycle. It also makes the statement that only such graphs can have an Euler cycle. In other words, if some vertices have odd degree, the the graph cannot have an Euler cycle. Notice that this statement is about Euler cycles and not Euler paths.

Proof: $\Longrightarrow$ *: If G is Eulerian then there is an Euler Cycle, P, in G. Every time a vertex is listed, that accounts for two edges adjacent to that vertex, the one before it in the list and the one after it in the list. This circuit uses every edge exactly once. So every edge is accounted for and there are no repetition of edges while counting. But vertices can be repeated, so if we count the degree for each vertex then, always we will have an incoming edge and outgoing edge, and for points in the extreme there are joined by an edge (since there are no repetition of edges there will be no duplicates that we need worry while computing the degree of a vertex). Thus every degree must be even for undirected graph and for directed graph we will have in-degree(v) = out-degree(v).*

This can be thought as P is either simple cycle (does not intersect itself), or not. If P is a simple cycle, each vertex in a simple cycle has in-degree(v) = out-degree(v), so the claim is true. If P is a cycle but not a simple cycle, then it must contain a simple cycle; remove it from G and from P; the remaining P is still an Euler cycle for the remaining G. Repeat removing (simple) cycles until no edges left. When removing a cycle, an in-edge and out-edge of the vertices on the cycle are removed. After a cycle deletion, the in-degree and out-degree of a node on the cycle decrease by exactly 1. At the end, when no edges are left, all in-degrees and out-degrees are 0. So all vertices v must have started with in-degree(v) = out-degree(v).

Claim: *For a graph G if it has in-degree(v) = out-degree(v) (or even degree) for every vertex, then for any vertex v there must be a path starting from v that comes back to v.*
Proof *for any vertex v, there must be a cycle that contains v. Start from v, and chose any outgoing edge of v, say (v, u). Since in-degree(v) = out-degree(v) we can pick some outgoing edge of u and continue visiting edges. Each time we pick an edge, we can remove it from further consideration. At each vertex other than v, at the time we visit an entering edge, there must be an outgoing edge left unvisited, since in-degree(v) = out-degree(v) for all vertices. The only vertex for which there may not be an unvisited entering edge is v because we started the cycle by visiting one of v's outgoing edges. Since there's always a leaving edge we can visit for any vertex other than v, eventually the cycle must return to v, thus proving the claim. In conclusion, we know that we will return to v eventually because every time we encounter a vertex other than v we are listing one edge adjacent to it. There are an even number of edges adjacent to every vertex, so there will always be a suitable unused edge to list next. So this process will always lead us back to v.* $\square$

$\Longleftarrow$: *Suppose every degree is even, then we have to prove that the graph has Euler Cycle. We will show that there is an Euler cycle by induction on the number of edges in the graph. The base case is for a graph G with two vertices with two edges between them. This graph is obviously Eulerian.*

Now suppose we have a graph G on $m > 2$ edges. We assume for total no. of edges less than m, we have Euler cycle. We start at an arbitrary vertex v and follow edges, arbitrarily selecting one after another until we return to v. Call this trail W. Let E_W be edges of W. We make new graph $G' = (V, E \setminus E_W)$. Assume, new graph G' has components $C_1, C_2, \ldots, C_k$. These components satisfy the induction hypothesis i.e., they are connected, have edges less than m, every vertex has in-degree(v) = out-degree(v) (even degree for undirected graph). We know that every vertex has in-degree(v) = out-degree(v) in the new graph G' because when we removed W, we removed an even number of edges from the vertices listed in that cycle. By induction each component has an Euler cycle, lets call them $P_1, P_2, \ldots, P_k$. By this process we get k edge disjoint cycles with at least one common vertex between each of them. Since G is connected which means there is a vertex a_i in each component C_i, which is present in both W and P_i. With loss of generality, assume that as we follow W, the vertices $a_1, a_2, \ldots, a_k$ are encountered in that order. We describe an Euler cycle G by starting at v follow W until reaching a_1, follow the entire P_1 ending back at a_1, follow W until reaching a_2, follow the entire P_2 ending back at a_2 and so on. End by following W, until reaching a_k follow the entire P_k ending back at a_k, then finish off W, ending at v. $\square$

Corollary: *A connected undirected graph contains an Euler Path (not Euler Cycle) if and only if it contains only two vertices with odd degrees. For a connected directed graph, it will contain an Euler path if and only if it contains only two vertices with $|deg_{in}(v) - deg_{out}(v)| = 1$.*

(b) *As mentioned in the proof we will use that to construct an algorithm which merges the edge disjoint cycles. This is known as Hierholzer's algorithm.*

- *Choose any starting vertex v, and follow a trail of edges from that vertex until returning to v. It is not possible to get stuck at any vertex other than v, because the even degree of all vertices ensures that, when the trail enters another vertex w there must be an unused edge leaving w. The tour formed in this way is a closed tour, but may not cover all the vertices and edges of the initial graph.*

- *As long as there exists a vertex u that belongs to the current tour but that has adjacent edges not part of the tour, start another trail from u, following unused edges until returning to u, and join the tour formed in this way to the previous tour.*

Also remember we need to check if for all vertices we have in-degree(v) = out-degree(v). If this condition is violated then the graph has no Euler Cycles. If the condition is true then, pick a vertex v and perform DFS from it until finding a back edge that links back to v. Once you find this cycle, traverse all edges of the cycle, and delete the corresponding edge in the adjacency list of G; to delete edges of G quickly we assume that we have modified DFS so that for each edge that we traverse we store a pointer to the corresponding edge in the adjacency list of G. With this information, deletion of an edge can be done in constant

time, basically because we don't need to search for the edge in G. Then we repeat the process. Overall this takes $O(|E|)$ time.

19. (**Shortest path with bounded negative edges**) Suppose we are given a directed graph G with weighted edges and two vertices s and t.

   (a) Describe and analyze an algorithm to find the shortest path from s to t when exactly one edge in G has negative weight.

   (b) Describe and analyze an algorithm to find the shortest path from s to t when exactly k edges in G have negative weights. How does the running time of your algorithm depend on k?

Solution:

(a) *Let G denote the input graph, let $w(x \to y)$ denote the weight of the edge $x \to y$, and let $u \to v$ denote the unique edge in G with negative weight. Remove edge $u \to v$ from G and let G' denote the resulting graph. Note that G' has no negative length edges. For any nodes x and y, let $dist(x, y)$ and $dist'(x, y)$ denote the distances from x to y in G and G', respectively.*

First we will check if the graph G has negative length cycle. If G has negative length cycle then the cycle must contain the only negative edge $u \to v$, and that cycle must pass through this edge. This implies that there is a path from v to u, which has only positive weights. This gives us a way to check for negative length cycles. If G has a negative length cycle then it must contain the edge $u \to v$: the shortest length cycle containing this arc can be seen to consist of a shortest path P from v to u in G' together with the arc $u \to v$. The length of this cycle is $dist'(v, u) + w(u \to v)$. G has a negative length cycle iff this quantity is negative. Thus, we can check if G has a negative length cycle by computing $dist'(v, u)$ in G' via Dijkstra's algorithm.

Suppose G does not have a negative length cycle. The shortest path in G from s to t either traverses the edge $u \to v$ or it doesn't; we consider each case separately. Then we have

$$dist(s, t) = min\left\{ dist'(s, t), dist'(s, u) + w(u \to v) + dist'(v, t) \right\}.$$

Thus, we can compute $dist(s, t)$ by running Dijkstra twice in G': once starting at s to compute both $dist'(s, t)$ and $dist'(s, u)$, and once starting from v to compute $dist'(v, t)$. The algorithm runs in $O(E \log V)$ time because, when the graph is connected then running time $O((E + V) \log V)$ can be reduced to $O(E \log V)$ source.

(b) *We will develop a similar algorithm like above but we will also use Bellman-Ford algorithm in combination with Dijkstra's algorithm. Key idea is here we will reduce the problem into a problem that we know how to solve. First we will remove all the negative edges from the graph and create new graph. All the definitions that are explained will be used here. After we get new graph G' we will create another graph such that has only following vertices $s, u_1, v_1, u_2, v_2, \ldots, u_k, v_k, t$. Where s, t are source and destination and*

u_i, v_i are endpoints/vertices of the negative edges, lets call this graph G''. Here an edge between two vertices x and y will be represented by the length of the shortest path from x and y with no negative edges. To get these values we will apply Dijkstar's algorithm in graph G' for all the vertices in graph of G''. Then we will add back all the negative edges to graph G'' and while doing so if edge $x \rightarrow y$ has more weight than the actual edge present between them in original graph (assuming it is present, if not do nothing) G, i.e. $w''(x \rightarrow y) > w(x \rightarrow y)$ then we will replace the w'' with w. Also, note that by construction $w''(x \rightarrow y) = dist'(x \rightarrow y)$.

Now we have graph G'' with reduced edges and with negative weights. If we apply Bellman-Ford Algorithm on this starting from s we will get shortest distance from s to t. The running time of this algorithm will be $O(k(E + V) \log V + k^3)$. We will apply Dijkstra's algorithm on graph G' for $O(k)$ times hence we get $O(k(E + V) \log V)$ and we apply Bellman-Ford algorithm on graph G'' which has $O(k)$ vertices and at most k^2 edges, so it takes $O(k \times k^2)$ running time.

20. (a) Your friend Hulk proposes adapting Dijkstra's algorithm in the following way to deal with negative-length edges. Pick the edge with the smallest length $\ell < 0$ (e.g., -10), and then increase the length of each edge by $|\ell|$ so all edges will become non-negative. Then run Dijkstra's on G with this updated lengths to find a shortest path from s to t. Does this also give you a shortest $s - t$ path in the original graph? Justify your answer.

   (b) We have a connected graph $G = (V, E)$. Suppose that we run both BFS and DFS on a vertex $u \in V$, and obtain the same BFS search tree and DFS search tree T, which contains all vertices of G. Prove or disapprove: $G = T$.
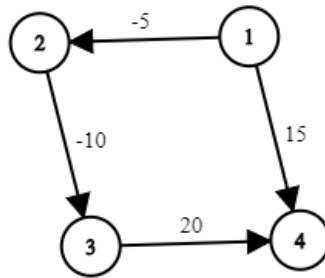

Figure 2: Counter example for modified Dijkstra's algorithm

Solution:

(a) *In the **Fig. 2**, I have provided a counter example for why the modification suggested in the question will not work. In that figure if we take an edge with smallest weight $l < 0$ to be -10 and add all the edges with $|l| = 10$, positive quantity then we will get edges of weight 0, 5, 25, 30. If we run algorithm then it would end up choosing the path $1 \to 4$, because it weight 25 whereas the other path has total weight to be 35 ($1 \to 2 \to 3 \to 4$). But in the original graph the shortest path would be to go from $1 \to 2 \to 3 \to 4$ because it has total weight equal to 5, but the path $1 \to 4$ we selected has total weight of 15.*

(b) **Proof by contradiction** <u>Note:</u> *G must be a tree in the first place. We will prove this by contradiction. Let us denote the tree produced by BFS and DFS be T. Suppose, G is not a tree. This implies that there must be an edge $(u, v) \in G$ such that $(u, v) \notin T$. In such a case, in the DFS tree, one of the u or v, will be an ancestor of the other(i.e, there will be a backedge). This is because if, for example, v was discovered first by DFS, we must explore until we find u while still exploring v, or it will make use of the edge from v to u. At the same time, in the BFS tree, u and v can differ by only one level (because there is a direct edge from u to v in the original graph G). Since, both BFS and DFS tree are same tree, it follows that one of u and v should be an ancestor of the other and they can differ by only one level. This implies that the edge connecting them must be in T. So, we arrive at the contradiction and hence it is proved that G is in fact a tree or G has no edges other than those in the BFS/DFS tree. Hence G = T.*

21. (**Minimum spanning tree**) This question explores the uniqueness of MST.

    (a) Prove or disprove (both directions) that an edge-weighted graph G has a unique minimum spanning tree *if and only if* the following conditions hold:

        - For any partition of the vertices of G into two subsets, the minimum weight edge with one endpoint in each subset is unique.
        - The maximum-weight edge in any cycle of G is unique.

    (b) Describe and analyze an algorithm to determine whether or not a graph has a unique minimum spanning tree.

Solution:

(a) *($\Longleftarrow$) Suppose that for every cut of the graph there is a unique minimum edge crossing the cut and the maximum-weight edge in any cycle of G is unique, but that the graph has two minimum spanning trees T and T'. Since T and T' are distinct, there must exist edges $e = (u, v)$ and $e' = (x, y)$ such that (u, v) is in T but not T' and (x, y) is in T' but not T. Let cut $S = (u, x)$. There is a unique minimum edge which spans this cut. Without loss of generality, suppose that it is not (u, v). Then we can replace (u, v) by this edge in T to obtain a spanning tree of strictly smaller in weight, a contradiction. Thus the spanning tree is unique.*

Now, if we take the edge e and add it to the MST T' we would get a cycle in the MST T', since this cycle will also part of the graph G that means we have unique maximum

weight edge. Without loss of generality, suppose maximum-weight edge is not e instead let that be e''. Then we can make new graph by disconnecting the edge e'' from the graph T' to get $T'' = T' \cup (u,v) \setminus e''$. Then we can replace e'' by (u,v) edge in T' to obtain a spanning tree of strictly smaller in weight, a contradiction. Thus the spanning tree is unique.

We have proved the implied by direction but the order direction is false and we will disprove by showing counter example. Conditions provided in the problem can be combined into statement as if the edge-weighted graph G has unique weights then it has unique minimum spanning tree again the converse is false.

($\implies$) If edge-weighted graph G has unique minimum spanning tree then the two conditions provided in the question need not be true. For a counter example to the converse, let $G = (V, E)$ where $V = \{x, y, z\}$ and $E = \{(x, y), (y, z), (x, z)\}$ with weights 1, 2, and 1 respectively. The unique minimum spanning tree consists of the two edges of weight 1, however the cut where $S = \{x\}$ doesn't have a unique light edge which crosses it, since both of them have weight 1. Similarly if the edge weights were 2, 2, 1 then we cycle doesn't have unique maximum edge weight bu the minimum spanning tree will be unique.

(b) <u>***Algorithm:***</u> UniqMST(G)

- *Run Kruskal's algorithm on G to find an MST m.*
- *Try running Kruskal's algorithm on G again. In this run, whenever we have a choice among edges of equal weights, we will first try the edges not in m, after which we will try the edges in m. Whenever we have found an edge not in m connects two different trees, we claim that there are multiple MSTs, terminating the algorithm.*
- *If we have reached here, then we claim that G has a unique MST.*

An ordinary run of Kruskal's algorithm takes $O(E \log V)$ time. The extra selection of edges not in m can be done in $O(E)$ time. So the algorithm achieves $O(E \log V)$ time-complexity. **Argument for working:** *Suppose we have an MST m' that is not the same as m. It is enough to show that the algorithm running on G will not reach step 3, since the edge found at the end of step 2, which is not in m and connecting two different trees would have been included in the resulting MST had we run Kruskal's algorithm to completion. Let w be the largest weight such that for any edge weighing less than w, it is in m if and only if it is in m'. Because m and m' have the same number of edges of weight w, there exist edges of weight w that are in m' but not in m. If the algorithm has exited before processing edges of those edges, we are done. Otherwise, suppose the algorithm is going to process the first edge e' among those edges now. Let S be the set of all edges that have been preserved so far to be included in the resulting MST. $S \subset m$. Since the algorithm have not finished processing edge of weight w not in m such as e', it must have not begun processing edges of weight w in m. So edges in S weigh less than w. That means $S \subset m'$. Recall that e' is in m'. Since $\{e'\} \cup S \subset m'$, where m' is a tree, e' must connect two different trees in S and the algorithm exits at this point.*

22. (**Maximum spanning tree**) Describe and analyze an algorithm to compute the *maximum*-weight spanning tree of a given edge-weighted graph.

Solution: Kruskal's Algorithm: A maximum spanning tree is a spanning tree (a tree that connects all vertices) of a weighted undirected graph having maximum weight. It can be computed by negating the weights for each edge and applying Kruskal's algorithm. Here, we assume that all edges weights are distinct. A spanning tree T of maximum weight is a maximum spanning tree which is given by

$$w(T) := max \left[\sum_{(u,v) \in T} w(u,v) \right].$$

Kruskal's algorithm can be summarized as follow as

- *Sort the edges of G into ascending order by weight. Let T be the set of edges comprising the maximum weight spanning tree. Set $T = \phi$.*

- *Add the first edge to T.*

- *Add the next edge to T if and only if it does not form a cycle in T. If there are no remaining edges exit and report G to be disconnected.*

- *If T has $n - 1$ edges (where n is the number of vertices in G) stop and output T. Otherwise go to step 3.*

If were to implement Kruskal's algorithm as it is we will get minimum spanning tree, because we are arranging weights in ascending order and then we are making a minimum spanning tree by picking edges which have low weight and add them into the tree. If we do the something instead if we arrange weights in descending order then we end up picking weights with high value and create a spanning tree which will give us maximum spanning tree.

> *__Algorithm:__ MaxSpanTree(G, $\{w_e\}$)*
> *$T \leftarrow \phi$ // Kruskal's Algorithm*
> *__Sort__ m edges so that $w(e_1) \geq w(e_2) \geq \ldots w(e_m)$*
> *__for__ $v \in V$*
>   *MakeSet(v)*
> *__for__ $i = 1, \ldots, n$*
>   *$(u,v) \leftarrow e_i$ //i^{th} expensive edge*
>   *__if__ FindSet(u) $\neq$ FindSet(v)*
>   *$T \leftarrow T \cup \{e_i\}$*
>   *UnionSet(u, v)*
>   *__return__ T*

We will use disjoint-set as our data structure, for example linked list or balanced tree.

- *MakeSet(x) create a singleton set containing x.*

- *FindSet(x): return the name of the unique set containing x.*

- *Union(x, y): merge the sets containing x and y respectively.*

We will need $O(m \log m)$ time to sort the array and we will have at most $2m$ FindSet operations and n UnionSet operations. Hence, total running time would be $O(m \log n)$.

23. (a) You are given a directed graph $G = (V, E)$ with $|V| = n$ vertices and $|E| = m$ edges. Let G^R be the graph obtained by reversing the directions of all the edges in G. Prove or disprove: you can compute adjacency matrix of G^R given the adjacency matrix of G in $o(n^2)$ time.

   (b) Given a set V of n points in the plane, let δ be the distance between two closest points in V. Consider the graph $G_{2\delta}$ with vertex set V and an edge (u, v) for every pair of points u and v in V at distance 2δ or less. Prove or disprove: $G_{2\delta}$ can have $\Omega(n \log n)$ edges.

5 Max-Flow Min-Cut, Linear Programming, Duality

24. Let f and f' be feasible (s, t)-flows in a flow network G, such that $v(f') > v(f)$. Prove that there is a feasible (s, t)-flow with value $v(f') - v(f)$ in the residual network G_f.

25. Let $u \to v$ be an arbitrary edge in an arbitrary flow network G. Prove that if there is a minimum (s, t)-cut (S, T) such that $u \in S$ and $v \in T$, then there is *no* minimum cut (S', T') such that $u \in T'$ and $v \in S'$.

26. (**Demand**) Suppose instead of capacities, we consider networks where each edge $u \to v$ has a non-negative *demand* $d(u \to v)$. Now an (s, t)-flow f is *feasible* if and only if $f(u \to v) \geq d(u \to v)$ for every edge $u \to v$. (Feasible flow values can now be arbitrarily large.) A natural problem in this setting is to find a feasible (s, t)-flow of *minimum* value.

   (a) Describe an efficient algorithm to compute a feasible (s, t)-flow, given the graph, the demand function, and vertices s and t as input. (Hint: find a flow that is non-zero everywhere, and then scale it up to make it feasible.)

   (b) Suppose you have access to a subroutine MaxFlow that computes *maximum* flows in networks with edge capacities. Describe an efficient algorithm to compute a *minimum* flow in a given network with edge demands; your algorithm should call MaxFlow exactly once.

   (c) State and prove an analogue of the max-flow min-cut theorem for this setting. (Do minimum flows correspond to maximum cuts?)

   Solution:

   (a) *For finding a feasible solution first we make sure that we have flow equal to demand on every edge then we balance the inflow and outflow at every vertex to final get a feasible*

solution. While doing this we remember the edges that were balanced already so that we don't change the value on them to disturb the balance on other vertices. We will keep doing this until we find an feasible solution. A trivial basic feasible solution will be ∞ on all edges.

> ***Algorithm:** FesiableSol(G, s, t, d)*
> *1 - Initialisation: $E_{in} \leftarrow E$, $E_{out} \leftarrow \phi$*
> *2 - Meeting Demand:*
> ***for** $e \in E$*
>   *if $d(e) - f(e) \neq 0$*
>   *$f(e) \leftarrow d(e)$*
> *3 - Balancing:*
> ***for** $v \in V$*
>   *$m \leftarrow \sum_{e \text{ into } v} f(e) - \sum_{e \text{ out of } v} f(e)$*
>   *if $m > 0$*
>   *Add the value of m to an e which is going into v and e should be*
> *part of E_{in}, if we fail to find any edge then $e \in E$*
>   *$E_{in} \setminus \{e\}$, $E_{out} \cup \{e\}$*
>   *else*
>   *Add the value of m to an e which is going out of v and e should be*
> *part of E_{in}, if we fail to find any edge then $e \in E$*
>   *$E_{in} \setminus \{e\}$, $E_{out} \cup \{e\}$*
> *4 - Repeat: We will repeat this process until we balance the graph. Go to step 3.*

Since the main steps that will take lot of time are steps 3 and 4 hence, the worst case running time of this algorithm is $O(E + V)$. We might have to do balancing multiple times but it will be linear with no. of vertices.

(b) *To find a optimal solution we will convert the graph into the max flow problem. We construct a new graph $G' = (V', E')$ from G by adding new source and target vertices s' and t', adding edges from s' to each vertex in V, adding edges from each vertex in V to t', and finally adding an edge from t to s. We also define a new capacity function $c' : E' \rightarrow \mathcal{R}$ as follows:*

- *For each vertex $v \in V$, we set $c'(s' \rightarrow v) = \sum_{u \in V} d(u \rightarrow v)$ and $c'(v \rightarrow t') = \sum_{w \in V} d(v \rightarrow w)$.*
- *For each edge $u \rightarrow v \in E$, we set $c'(v \rightarrow u) = d(u \rightarrow v)$.*
- *Finally, we set $c'(s \rightarrow t) = \infty$.*

Intuitively, we construct G' by replacing any edge $u \rightarrow v$ in G with three edges: an edge $v \rightarrow u$ with capacity $d(u \rightarrow v)$, an edge $s' \rightarrow v$ with capacity $d(u \rightarrow v)$, and an edge $u \rightarrow t'$ with capacity $d(u \rightarrow v)$. If this construction produces multiple edges from s' to the same vertex v (or to t' from the same vertex v), we merge them into a single edge with the same total capacity.

In G', the total capacity out of s' and the total capacity into t' are both equal to D. We call a flow with value exactly D a saturating flow, since it saturates all the edges leaving s' or entering t'. If G' has a saturating flow, it must be a maximum flow, so we can find it using any max-flow algorithm.

Lemma: G has a feasible (s, t)-flow if and only if G' has a saturating (s', t')-flow.
Proof: Let $f : E \to \mathcal{R}$ be a feasible (s, t)-flow in the original graph G. Consider the following function $f' : E' \to \mathcal{R}$:

$$f'(u \to v) = f(u \to v) - d(u \to v)$$
$$f'(s' \to v) = \sum_{u \in V} d(u \to v)$$
$$f'(v \to t') = \sum_{w \in V} d(u \to w)$$
$$f'(t \to s) = |f|$$

We easily verify that f' is a saturating (s', t')-flow in G. The admissibility of f implies that $f(e) \geq d(e)$ for every edge $e \in E$, so $f'(e) \geq 0$ everywhere. Note

$$\sum_{u \in V'} f'(u \to v) = \sum_{w \in V'} f(v \to w)$$

for every vertex $v \in V$ (including s and t). Thus, f' is a legal (s', t')-flow, and every edge out of s' or into t' is clearly saturated. Intuitively, f' diverts $d(u \to v)$ units of flow from u directly to the new target t', and injects the same amount of flow into v directly from the new source s'. The same tedious algebra implies that for any saturating (s', t')-flow $f' : E' \to \mathcal{R}$ for G', the function $f = f'|E + d$ is a feasible (s, t)-flow in G.

Thus, we can compute a feasible (s, t)-flow for G, if one exists, by searching for a maximum (s', t')-flow in G' and checking that it is saturating. Once we have found a feasible (s, t)-flow in G, we can transform it into a maximum flow using an augmenting-path algorithm, but with one small change. To ensure that every flow we consider is feasible, we must redefine the residual capacity of an edge as follows:

$$c_f(u \to v) = \begin{cases} f(u \to v) & \text{if } u \to v \in E \\ f(v \to u) - d(v \to u) & \text{if } v \to u \in E \\ 0 & \text{otherwise} \end{cases}$$

Otherwise the algorithm is unchanged. So, if we have access to MaxFlow subroutine then we can compute the minimum flow for the given network with one call on the new graph. And the running time for this algorithm will be $O(EV)$.

(c) Most of the definitions are same as what has been defined in the lecture notes.

Definition: Demand of a (s, t)-cut (A, B) is defined as

$$dem(A, B) = \sum_{e \text{ out of } A} d(e).$$

43

<u>*Flow Value Lemma:*</u>

$$v(f) = \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ into } A} f(e).$$

Weak Duality: *Let F be any flow, and let (A, B) be any (s, t)-cut. Then the value of the flow is at least the demand of the cut.*

$$v(f) \geq dem(A, B)$$

<u>*Proof:*</u>

$$v(f) = \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ into } A} f(e)$$

$$v(f) \geq \sum_{e \text{ out of } A} d(e) - \sum_{e \text{ into } A} d(e)$$

$$v(f) \geq \sum_{e \text{ out of } A} d(e)$$

$$v(f) \geq dem(A, B).$$

First and second line comes due to the fact that $d(e) \leq f(e)$. Third line comes from $\sum_{e \text{ into } A} d(e) \leq \sum_{e \text{ into } A} f(e)$, hence the inequality sign remains the same. From this weak duality we get certificate of the optimality. Let f be any flow, and (A, B) be any (s, t)-cut. If $v(f) = dem(A, B)$, then f is a min flow, and (A, B) a max cut.

Now, we will state that value of the minimum flow is equal to demand of the maximum cut. To prove this we will use the residual flow network. We will create new graph by creating edges with residual capacities similar to what has been described in above part of the problem:

$$c_f(e) = \begin{cases} f(e) & \text{if } e \in E \\ f(e) - d(e) & \text{if } e^R \in E \end{cases}$$

and in residual graph we will keep edges with positive edges only. Now with the help of augmenting path theorem we can state the following: f is a minimum flow if and only if there are no augmenting paths in residual graph G_f i.e.

- *f is a minimum flow*
- *There is no augmenting path with respect to f*
- *There exists a cut (A, B) such that $dem(A, B) = v(f)$.*

This analogous to max-flow min-cut theorem.
<u>**Proof:**</u> *We will show that*

- *$(A \implies B)$ Let P be an augmenting path with respect to f Then f' below is a feasible flow with $v(f') < v(f)$. This can be proven by the bottleneck argument.*
- *$(B \implies C)$ Assuming G_f has no augmenting path. Let A be the set of nodes reachable from s in G_f. Clearly this is a (s, t)-cut. From this we can see that $v(f) = dem(A, B)$.*
- *$(C \implies A)$ This is the implication of weak duality.*

27. **(Vertex cover)** A *vertex cover* of an undirected graph $G = (V, E)$ is a subset of the vertices which touches every edge that is, a subset $S \subseteq V$ such that for each edge $u, v \in E$, one or both of u, v are in S. Describe and analyze an algorithm, as efficient as you can, to find a minimum vertex cover in a bipartite graph.

Solution: **Konig's theorem:** *If G is bipartite, the cardinality of the maximum matching is equal to the cardinality of the minimum vertex cover.*

This gives us a polynomial time algorithm for vertex cover in bipartite graph.

> **Algorithm:** VertexCover(G)
> - *Input undirected bipartite graph $G = (V, E)$, partition of V into sets L, R.*
> - *Construct a network $G' = (V', E')$ as follows*
>   - *Vertex set $V' := V \cup \{s, t\}$, where s and t are two new vertices.*
>   - *E' contains a directed edge (s, u) for every $u \in L$, a directed edge (u, v) for every edge $(u, v) \in E$, where $u \in L$ and $v \in R$, and a directed edge (v, t) for every $v \in R$.*
>   - *Each edge has capacity 1.*
> - *Find a minimum-capacity cut S in the network G'*
> - *Define $L_1 := L \cap S$, $L_2 := L - S$, $R_1 := R \cap S$, $R_2 := R - S$.*
> - *Let B be the set of vertices in R_2 that have neighbors in L_1.*
> - *$C := L_2 \cup R_1 \cup B$*
> - *Output C.*

We want to show that the algorithm outputs a vertex cover, and that the size of the output set C is indeed the size of the minimum vertex cover.

Claim: *The output C of the algorithm is a vertex cover.*

Proof: *The set C covers all edges that have an endpoint either in L_2 or R_1, because C includes all of L_2 and all or R_1. Regarding the remaining edges, that is, those that have endpoint in L_1 and the other endpoint in R_2, all such edges are covered by B.* □

Claim: *There is no vertex cover of size smaller than $|C|$.*

Proof: *Let k be the capacity of the cut. Then k is equals to $|L_2| + |R_1| + edges(L_1, R_2)$ and $k \geq |L_2| + |R_1| + |B| = |C|$ but k is equal to the capacity of the minimum cut in G', which is equal to the cost of the maximum flow in G' which is equal to the size of the maximum matching in G. This means that G has a matching of size k, and so every vertex cover must have size $\geq k \geq |C|$.* □

Runtime of this algorithm is $O(EV)$ if we use a variant of Ford-Fulkerson algorithm to compute the minimum cut.

28. **(Updating max flow)** You are given a flow network $G = (V, E)$ with source s and sink t, and integer capacities.

(a) Suppose that you are given a max flow in G. Now we increase the capacity of a single edge $(u,v) \in E$ by 1. Given an $O(m+n)$-time algorithm to update the max flow.

(b) Now suppose all edges have unit capacity and you are given a parameter k. The goal is to delete k edges so as to reduce the maximum $s-t$ flow in G as much as possible. In other words, you should find a set of edges $F \subseteq E$ so that $|F| = k$ and the maximum $s-t$ flow in $G' = (V, E - F)$ is as small as possible subject to this. Describe and analyze a polynomial-time algorithm to solve this problem.

Solution:

(a) *If there exists a minimum cut on which (u,v) doesn't lie then the maximum flow can't be increased, so there will exist no augmenting path in the residual network. Otherwise it does cross a minimum cut, and we can possibly increase the flow by 1. Perform one iteration of Ford-Fulkerson. If there exists an augmenting path, it will be found and increased on this iteration. Since the edge capacities are integers, the flow values are all integral. Since flow strictly increases, and by an integral amount each time, a single iteration of the while loop of Ford-Fulkerson will increase the flow by 1, which we know to be maximal. To find an augmenting path we use a BFS, which runs in $O(m+n)$.*

(b) *First observe that by removing any k edges in a graph, we reduce the capacity of any cut by at most k, and so, the min-cut will reduce by at most k. Therefore, the max-flow will reduce by at most k. Now we show that one can in fact reduce the max-flow by k. To achieve this, we take a min-cut (A,B) of G and remove k edges going out of it. The capacity of this cut will now become $f - k$, where f is the value of the max-flow. Therefore, the min-cut becomes $f - k$, and so, the max-flow becomes $f - k$.*

Consider the set of edges E' that cross the cut (A,B), i.e. all of the forwards edges that start in A and end in B. If E' has k or more edges in it, then the set F can consist of any k edges of E'. If E' has fewer than k edges, then for the set F, we can take all of the edges of E', together with any other edges of G, so that F has exactly k edges in the overall set.

Now to find k edges in the graph G, we need $O(mn)$ runtime, because we can find the min-cut of the graph G in $O(mn)$ time. Then identifying the cut and the edges crossing the cut can be performed in time $O(nm)$ in the residual graph (by BFS, say) and in time $O(m)$, respectively. Hence total runtime would be $O(mn)$.

29. **(Filling classrooms)** Faced with the threat of brutally severe budget cuts, Potemkin University has decided to hire actors to sit in classes as "students", to ensure that every class they offer is completely full. Because actors are expensive, the university wants to hire as few of them as possible.

Building on their previous leadership experience at the now-defunct Sham-Poobanana University, the administrators at Potemkin have given you a directed acyclic graph $G = (V, E)$, whose vertices represent classes, and where each edge $i \to j$ indicates that the same "student" can attend class i and then later attend class j. In addition, you are also given an array $cap[1, \ldots, V]$ listing the maximum number of "students"

who can take each class. Describe an analyze an algorithm to compute the minimum number of "students"that would allow every class to be filled to capacity.

*Solution: **Disjoint-Path Covers**: This problem is a application of max-flow min-cut problem and we will apply a variant Ford-Fulkerson algorithm on bipartite graph to solve this problem.*

*A **path cover** of a directed graph G is a collection of directed paths in G such that every vertex of G lies on at* least *one path. A **disjoint-path cover** of G is a path cover such that* every *vertex of G lies on* exactly *one path. Trivially every directed graph has a trivial disjoint-path cover consisting of several paths (i.e. of size $|V|$) of length zero. Instead, we will look for disjoint-path covers that contain as few paths as possible. This problem is NP-hard in general a graph has a disjoint-path cover of size 1 if and only if it contains a Hamiltonian path. Also, there is an efficient flow-based algorithm for directed acyclic graphs (DAG).*

To solve this problem for a given directed acyclic graph $G = (V, E)$, we construct a new bipartite graph $G' = (V', E')$ as follows

- *G' contains two vertices $v^{\sharp}$ and $v^{\flat}$ for every vertex $v \in G$,*

- *G' contains an undirected edge $(u^{\flat}, v^{\sharp})$ for every directed edge $u \to v \in G$.*

Note if G is represented as an adjacency matrix, then G' is the bipartite graph represented by the same adjacency matrix. Bipartite graph G' is made of two independent disjoint vertex sets made from $\sharp$ and $\flat$.

Definition: (Subgraph) Subgraph of a graph is another graph, formed from a subset of the vertices of the graph and all of the edges connecting pairs of vertices in that subset.

Definition: (Matching) Given an undirected graph $G = (V, E)$. A subset of edges $M \in E$ is a matching if each node in V appears in at most one edge in M.

Definition: (Maximal Matching) A maximal matching is a matching to which no more edges can be added without increasing the degree of one of the nodes to two; it is a local maximum.

Definition: (Maximum Matching) A maximum matching is a matching with the largest possible number of edges; it is globally optimal. Here we are interested in finding a matching of max cardinality i.e., adding any edge will make it no longer a matching.

Claim: G can be covered by k disjoint paths if and only if the new graph G' has a matching of size $V - k$.

Proof: ($\implies$) Suppose G has a disjoint path cover P with k paths; think of P as a subgraph of G. Every vertex in P has in-degree either 0 or 1; moreover, there is exactly one vertex with in-degree 0 in each path in P. It follows that P has exactly $V - k$ edges. Now define a subset M of the edges of G' as follows

$$M := \{(u^{\flat}, v^{\sharp}) \in E' | u \to v \in P\}.$$

By definition of disjoint-path cover, every vertex of G has at most one incoming edge in P and at most one outgoing edge in P. We conclude that every vertex of G' is incident to at most one edge in M; that is, M is a matching of size V − k.

($\Longleftarrow$) Suppose G' has a matching M of size V − k. We project M' back to G by defining a subgraph P = (V, M'), where

$$M' := \{u \to v \in E | (u^\flat, v^\sharp) \in M\}.$$

By definition of matching, every vertex of G has at most one incoming edge in P and at most one outgoing edge in P. It follows that P is a collection of disjoint directed paths in G; since P includes every vertex, P defines an disjoint path cover with V − k edges. The number of paths in P is equal to the number of vertices in G that have no incoming edge in M'. We conclude that P contains exactly k paths. $\square$

It follows immediately that we can find a minimum disjoint-path cover in G by computing a maximum matching in G', using Ford-Fulkerson's maximum-flow algorithm, in $O(VE)$ time. Despite its formulation in terms of dags and paths, this is really a maximum matching problem: We want to match as many vertices as possible to distinct successors in the graph. The number of paths required to cover the dag is equal to the number of vertices with no successor. (And of course, every bipartite maximum matching problem is really a flow problem.)

Now using above theory we can solve our given problem. In the given problem we have classes represented by vertices and each directed edge i → j represents a path for students who attend class i and can attend class j. So, if we find minimum disjoint-path that covers all vertices then we are done, because then we can find the maximum no. of students required to fill classes to full capacity on each disjoint-path and then finally sum all these values to gives minimum number of "students" that would allow every class to be filled to capacity. To solve this problem we will compute the minimum disjoint-path cover of the dag G by computing the maximum matching of the bipartite graph G' (which is formed as explained above). After computing the maximum matching we can compute all the paths disjoint-paths P. After computing the paths P we can compute the maximum required capacity for each path and sum them up, which will give minimum students required. Running time for an algorithm will be $O(EV)$ and we apply a variant of Ford-Fulkerson's maximum-flow algorithm.

Note: Computing the disjoint-cover paths and then computing maximum over it will take less time compared computing the actual max-flow. Reason: We know that once we have max-matching ($\Longrightarrow$ max-flow) on the bipartite graph G' we can find the min-cut in polynomial time. Which will give us all the matching ($\Longrightarrow$ vertices) that gives max-flow on the graph G'. Using this we can compute all the disjoint-cover paths and max over all paths (and sum of them) in linear time because we use same adjacency matrix to represent G and G'.

30. (**Max bipartite matching**)

   (a) Give a linear-programming formulation of the bipartite maximum matching problem. The input is a bipartite graph $G = (U \cup V; E)$, where $E \subseteq U \times V$; the output is the largest matching in G. Your linear program should have one

variable for each edge.

(b) Now dualize the linear program from part (a). What do the dual variables represent? What does the objective function represent? What problem is this!?

31. (**Integer linear programming**) An *integer linear-programming* problem is a linear-programming problem with the additional constraint that the variables x must take on *integral* values.

(a) Show that *weak duality* (CLRS Lemma 29.8) holds for an integer linear program.

(b) Show that *strong duality* (CLRS Theorem 29.10) does not always hold for an integer linear program.

(c) Given a primal linear program in standard form, let us define P to be the optimal objective value for the primal linear program, D to be the optimal objective value for its dual, ILP to be the optimal objective value for the integer version of the primal (that is, the primal with the added constraint that the variables take on integer values), and ID to be the optimal objective value for the integer version of the dual. Assuming that both the primal integer program and the dual integer program are feasible and bounded, show that

$$ILP \le P = D \le ID.$$

Solution:

(a) *If we refer the literature on the integer linear programming (ILP) and mixed integer linear programming (MILP) we will notice that there is no direct way of writing the dual of ILP/MILP problem. Duality of ILP/MILP problem depends on how the primal ILP/MILP problems is being is solved. If we solve primal ILP/MILP problem using branch and bound method or branch and cut method or cutting-plane method, all the three methods will give us different dual characterization for primal ILP/MILP [source1], [source2], [source3]. These dual problems use something known as priced functions.*

Here in our problem we notice that CLRS Lemma 29.8 derivation doesn't make any kind of assumptions on the dual variables and the constraints hence the Lemma must hold even for the ILP. More rigours proof for "Weak Duality" was shown by Jeroslow in 1978, 1979.

(b) *Consider the integer linear program given in standard form by maximize x subjected to $A = 1$, $b = 0.3$, and $c = 2$. The highest we can get this is 0 since that's that only integer value that x can be. Now, consider the dual to this, that is, we are trying to minimize $0.3x$ subject to the constraint that $x \ge 2$. This will be minimized when $x = 2$, so, the smallest solution we can get is 0.6. Since we have just exhibited an example of a integer linear program that has a different optimal solution as it's dual, the duality theorem does not hold for integer linear program.*

(c) *The first inequality comes from looking at the fact that by adding the restriction that the solution must be integer valued, we obtain a set of feasible solutions that is a subset*

49

of the feasible solutions of the original primal linear program. Since, to get ILP, we are taking the max over a subset of the things we are taking a max over to get P, we must get a number that is no larger. In short we are restricting our feasible set of solutions in ILP hence the optimal value can't be larger than the optimal value of P which is relaxed linear programming of the ILP. The third inequality is similar, except since we are taking minimum over a subset, hence the inequality goes the other way. The middle equality is given by CLRS Theorem 29.10 which is nothing the duality of linear programming problem.

6 NP, NPC, Complexity Theory

32. (**Reduction**) Given a set X of n Boolean variables $x_1, \ldots, x_n$; each can take the value 0 or 1 (equivalently, "false" or "true"). By a term over X, we mean one of the variables x_i or its negation $\bar{x}_i$. Finally, a clause is simply a disjunction of distinct terms $t_1 \vee t_2 \vee \ldots \vee t_\ell$. (Again, each $t_i \in \{x_1, x_2, \ldots, x_n, \overline{x_1}, \ldots, \overline{x_n}\}$.) We say the clause has length l if it contains l terms. A truth assignment for X is an assignment of the value 0 or 1 to each x_i; in other words, it is a function $v : X \to \{0,1\}$. The assignment v implicitly gives $\bar{x}_i$ the opposite truth value from x_i. An assignment satisfies a clause C if it causes C to evaluate to 1 under the rules of Boolean logic; this is equivalent to requiring that at least one of the terms in C should receive the value 1. An assignment satisfies a collection of clauses $C_1, \ldots, C_k$ if it causes all of the C_i to evaluate to 1; in other words, if it causes the conjunction $C_1 \wedge C_2 \wedge \ldots C_k$ to evaluate to 1. In this case, we will say that v is a satisfying assignment with respect to $C_1, \ldots, C_k$; and that the set of clauses $C_1, \ldots, C_k$ is *satisfiable*. For example, consider the three clauses

$$(x_1 \vee \overline{x_2}), (\overline{x_1} \vee \overline{x_3}), (x_2 \vee \overline{x_3}).$$

A truth assignment v that sets all variables to 1 is not a satisfying assignment, because it does not satisfy the second of these clauses; but the truth assignment v' that sets all variables to 0 is a satisfying assignment.

We can now state the `Satisfiability` Problem, also referred to as SAT:

Given a set of clauses $C_1, \ldots, C_k$ over a set of variables $X = \{x_1, \ldots, x_2\}$, does there exist a satisfying truth assignment?

Suppose we are given an oracle $\mathcal{O}$ which can solve the `Satisfiability` problem. Namely if we feed a set of clauses to $\mathcal{O}$, it will tell us YES if there is a satisfying assignment for all clauses or NO if there exists none. Show that you can actually find a satisfying truth assignment v for $C_1, \ldots, C_k$ by asking questions to $\mathcal{O}$. Describe your procedure, and analyze how many questions you need to ask.

Solution: *Problems of the above category fall into topics of computational complexity theory and computability theory. When we are trying to solve a NP-hard or NP-complete problem we are interested in two things (a) for the given problem does a solution exist (b) what is the solution/solution set. (I am not going to formally describe and write rigorous definitions for all the theory because it will take lot space)*

Every NP problem can be described in two versions as follows

- *Problem of deciding whether a solution exist is know as decision problem. In these kind of problems we try to formulate the problems in terms of decision, meaning if we pose a question we want to know if the answer is 'yes' or 'no'. A method for solving a decision problem, given in the form of an algorithm, is called a decision procedure for that problem. An example of a decision problem is deciding whether a given natural number is prime.*

- *Problem of finding the exact solution is know as search problem. Generally, we are interested in finding a solution not just knowing if one exists. Solving the search the problem is harder than the decision problem.*

The Boolean satisfiability problem or SAT problem is known to be NP-complete. In the search version of the problem, we can't stop merely at saying 'yes' or 'no' to the question of whether the given formula is satisfiable; if the answer is "yes" we must actually find and output a satisfying assignment. In decision version of the problem 'yes' or 'no' answers to the question of whether the given formula is satisfiable is enough. In the SAT problem we are given the conjunctive normal form (CNF) which are nothing but conjunction of clauses (or a single clause). Here our decision problem is if CNF is satisfiable and search problem is find a satisfying assignment to CNF if one exist else output NONE.

<u>Note:</u> *If we can solve the search problem then we can certainly solve the decision problem. In SAT if we are given a CNF, and if we can find a satisfying assignment or tell that one does not exist, we can certainly say whether or not there exists a satisfying assignment. So search is harder; if we can solve it we can certainly solve decision.*

We are given a oracle machine $\mathcal{O}$ which solves the decision problem and we don't need to worry how it does. We need to build a procedure to solve the "equivalent" search problem for the SAT problem. This is know as self-reduction and using the oracle we can build a procedure to solve the search problem in polynomial time. That is, search reduces to decision for SAT.

Now our problem is that we are given a CNF ϕ and we want to find a satisfying assignment to if one exist. To help us we have, oracle $\mathcal{O}$. We can use oracle $\mathcal{O}$ to test whether or not ϕ is satisfiable. Also oracle $\mathcal{O}$ can be invoked on any formula hence we will invoke it on formulas constructed out if ϕ and extract the truth assignment one bit at a time. For example consider,

$$\phi(x_1, x_2, x_3, x_4) = (x_1 \vee \neg x_2) \wedge (x_2 \vee x_3 \vee \neg x_1) \wedge (x_4 \vee \neg x_3)$$

First ask $\mathcal{O}$ if ϕ is satisfiable. If it says no, we output NONE and we are done since there is no solution. If it says yes, then we need to find values for each Boolean variable. Consider two formulas, which we denote as ϕ_0 and ϕ_1. Each of the new formulas we created are will be one less variable, formed by setting Boolean variable x_1 to one of the truth values.

- $\phi_0(x_2, x_3, x_4) = \phi(0, x_2, x_3, x_4)$. *Substituting $x_1 = 0$ and simplifying the original equation will give $\neg x_2 \wedge (x_4 \vee \neg x_3)$.*

- $\phi_1(x_2, x_3, x_4) = \phi(1, x_2, x_3, x_4)$. *Substituting $x_1 = 1$ and simplifying the original equation will give $(x_2 \vee x_3) \wedge (x_4 \vee \neg x_3)$.*

The key point is that one of the following must be satisfiable. Because x_1 must be either 1 or 0. If we ask $\mathcal{O}$ (since this is a decision problem) which is the correct solution then we will know what is the correct truth assignment for the Boolean variable x_1 (search problem being solved with the help of oracle). Suppose $\mathcal{O}$ says yes to $x_1 = 0$ then it is the solution; if it says no then $x_1 = 1$ is the solution. Because there must exist some satisfying assignment to ϕ with x_1 being 0 or 1. We can keep repeating this process for all Boolean variables in the ϕ CNF.

Finally we make total of $(n+1)$ calls to the oracle, and end up with a satisfying assignment. Here n is the total no. of Boolean variables present in the problem and for each Boolean variable there is a associated call to the oracle. Finally, the additional one call comes from the initial check done on the ϕ to know if a solution exists or not in the first place. Also, note that if there is no solution to the SAT problem then we make only 1 call to the oracle. Additionally, note that we can have Boolean variable with two possible correct assignments but we are only interested in one.

33. (NP **closed under regular operations**) Given any sets $X \subseteq \{0,1\}^*$ and $Y \subseteq \{0,1\}^*$, we can define new sets under regular operations. For example, the *union* is $X \cup Y = \{x : x \in X \text{ or } x \in Y\}$; and the *concatenation* is $X \circ Y = \{xy : x \in X \text{ and } y \in Y\}$. We consider applying these operations on problems in NP.

    (a) Prove or disprove that NP is closed under union. Namely for any $X \in$ NP and $Y \in$ NP, does $X \cup Y \in$ NP always hold?

    (b) Prove or disprove that NP is closed under concatenation.

    (c) Prove or disprove that NP is closed under complement.

Solution: We will take advantage of the machinery of formal-language theory and define some concepts.

*Definition: (**Alphabet**) Alphabet Σ is a finite set of symbols.*

*Definition: (**Language**) A language L over Σ is any set of strings made up of symbols from Σ. For example if $\Sigma = \{0,1\}$, the set*

$$L = \{10, 11, 101, 111, 1011, 1101, 10001, \ldots\}$$

is the language of binary representations of prime numbers.

We denote the empty string by ϵ, the empty language by ϕ, and the language of all strings over Σ by Σ^*. For example, if $\Sigma = \{0,1\}$, then

$$\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$$

is the set of all binary strings. Every language L over Σ is a subset of Σ^*.

(a) Let X and Y be languages in NP. Let $V_1(z, c_X)$ be an algorithm that, for a string z and a possible certificate c, verifies whether c is actually a certificate for $z \in X$ (and similarly we will define for $z \in Y$). Thus, $V_1(z, c_X) = 1$ if certificate c verifies $z \in X$, and $V_1(z, c_X) = 0$ otherwise. Since both X and Y are both in NP, we know that $V_1(z, c_X)$ terminates in polynomial time $O(|z|^{d_1})$ for some constant d_1 and $V_2(z, c_Y)$ terminates in polynomial time $O(|z|^{d_2})$ for some constant d_2.

To show that $X \cup Y$ is also in NP, we will construct a polynomial-time verifier V_3 for $X \cup Y$. Since a certificate c for $X \cup Y$ will have the property that either $V_1(z, c_X) = 1$ or $V_2(z, c_Y) = 1$, we can easily construct a verifier $V_3(z, c) = V_1(z, c_X) \lor V_2(z, c_Y)$. Clearly then $z \in X \cup Y$ if and only if there is a certificate c such that $V_3(z, c) = 1$. Notice also that the new verifier V_3 will run in time $O(|z|^{max(d_1, d_2)})$, which is polynomial. Therefore, the union $X \cup Y$ of two languages in NP is also in NP. So NP is closed under union.

(b) Now we will show NP is closed under concatenation. Our goal is to construct a polynomial-time verifier $V_4(z, c)$ for a string z and the possible certificate c. Suppose $|z| = n$. We can define a verifier $V_4(z, c) = 1$ iff for some $k \in \{0, 1, 2, \ldots, n\}$ we have $V_1(z_1 z_2 \ldots z_k, c_X) = 1$ and $V_2(z_{k+1} z_{k+2} \ldots z_n, c_Y) = 1$. We have to check this for all possible k's. Hence running time for this will be in the order of $O(|z|^{1+max(d_1, d_2)})$, which is polynomial. Because we have to check for $|z| + 1$ possible k's and each check will take at most $O(|z|^{max(d_1, d_2)})$ time. Therefore, the concatenation $X \circ Y$ of two languages in NP is also in NP. So NP is closed under concatenation. Note that k specifies the position where the original string z should be split into two parts.

(c) In computational complexity theory, the complement of a decision problem is the decision problem resulting from reversing the yes and no answers. Equivalently, if we define decision problems as sets of finite strings, then the complement of this set over some fixed domain is its complement problem. For example, one important problem is whether a number is a prime number. Its complement is to determine whether a number is a composite number (a number which is not prime). Here the domain of the complement is the set of all integers exceeding one. It is not known whether NP is closed under complement. This question in the literature is known as "NP versus co-NP" question.

34. (**2-SAT**) Consider 2-SAT, in which the input is a formula with at most 2 literals per clause. Show that 2-SAT is in P. (Hint: you may want to consider an algorithm for testing if a graph is bipartite).

Solution: First we will create an implication graph. Consider a clause $(x_1 \lor x_2)$. We can view this as two implications $\overline{x_1} \to x_2$ and $\overline{x_2} \to x_1$. Now consider the implication $u \to v$ as a directed edge in graph from u to v. We call this graph the implication graph. The general claim is that a 2-SAT formula is satisfiable iff there is no path in the implication graph from x_i to $\overline{x_i}$ for any i. Furthermore, we can compute a truth assignment for all x_i as follows

Algorithm: 2SAT

- *Build the implication graph.*

- *Identify all strongly connected components in the implication graph.*

- *If any SCC has both x_i and $\overline{x_i}$, then the formula is not satisfiable.*

- *Construct the SCC graph where each vertex in the SCC graph is a single SCC and there is and edge from u to v in the SCC graph if there is an edge from one term in the SCC of u to a term in the SCC of v.*

- *Topologically order the SCC graph.*

- *For each component in topological order, if its terms do not already have truth assignments, set all the terms in the component to be false. This may cause you to set the negation of all these terms to true in other components.*

Running time for above algorithms depends on finding the strong connected components and doing topological ordering on the DAG, but these both algorithms when doing using a simple depth first search take linear time i.e. $O(E + V)$, hence this shows that 2-SAT is in P.

Claim: *If implication graph G contains a path from $x_1 \rightsquigarrow x_2$, then it also contains a path from $\overline{x_2} \rightsquigarrow \overline{x_1}$.*

Proof: *If there is a path from $x_1 \rightsquigarrow x_2$ it means it comes from the clause $(\overline{x_1}, x_2)$. Hence, by construction there will be a path from $\overline{x_2} \rightsquigarrow \overline{x_1}$. For example let the path from $x_1 \rightsquigarrow x_2$ be $x_1 \rightarrow v_1 \rightarrow v_2 \ldots \rightarrow x_2$. Now, by construction of G, if there's an edge (u, v), then there's also an edge $(\overline{v}, \overline{u})$. Hence, the edges $(\overline{x_2}, \overline{v_k}), \ldots, (\overline{v_1}, \overline{x_1})$ will be present in G. Hence, there is a path from $\overline{x_2} \rightsquigarrow \overline{x_1}$. □*

Claim: *a 2-SAT formula is satisfiable iff there is no path in the implication graph from x_i to $\overline{x_i}$ for any i.*

Proof: *(By contradiction) Suppose there is path from x_i to $\overline{x_i}$ and $\overline{x_i}$ to x_i for some variable x_i in G, and there is a satisfying assignment to 2-SAT problem. This can only happen when we have both variables in the same SCC. Then we get two cases*

- *($x_i = TRUE$) Let the path from $x_i \rightsquigarrow \overline{x_i}$ be $x_i \rightarrow v_1 \rightarrow v_2 \ldots \rightarrow \overline{x_i}$. If there is an edge (v_i, v_j) then we have clause $(\overline{v_i}, v_j)$. Also let variables from x_i to v_k be TRUE and from v_{k+1} to $\overline{x_i}$ be FALSE. This means for the edge (v_k, v_{k+1}) we have clause $(\overline{v_k}, v_{k+1})$ which is evaluation to clause and this clause will make entire 2-SAT unsatisfiable. Hence we arrived to at a contradiction. We can arrive at the same contradiction starting from other direction (because above claim is true).*

- *($x_i = FALSE$) Using similar argument provided above we will arrive to at a contradiction.*

Hence 2-SAT is satisfiable if and only if for a variable x_i, we can't have x_i and $\overline{x_i}$ be present in the same strongly connected components. □

Using above claims (which provides proof of correctness to our algorithm) and the algorithm provided we can concluded that 2-SAT is in P.

35. (Bonnie and Clyde) Bonnie and Clyde have just robbed a bank. They have a bag of money and want to divide it up. For each of the following scenarios, either give a

Number	i			j			
	1	2	3	1	2	3	4
t_1	1	0	0	1	0	0	1
f_1	1	0	0	0	1	1	0
t_2	0	1	0	1	0	1	0
f_2	0	1	0	0	1	0	1
t_3	0	0	1	1	1	0	1
f_3	0	0	1	0	0	1	0

Figure 3: Reducing 3SAT to Subset problem

Number	i			j			
	1	2	3	1	2	3	4
x_1	0	0	0	1	0	0	0
y_1	0	0	0	1	0	0	0
x_2	0	0	0	0	1	0	0
y_2	0	0	0	0	1	0	0
x_3	0	0	0	0	0	1	0
y_3	0	0	0	0	0	1	0
x_4	0	0	0	0	0	0	1
y_4	0	0	0	0	0	0	1

Figure 4: Reducing 3SAT to Subset problem

polynomial-time algorithm, or prove that the problem is NP-complete. The input in each case is a list of the n items in the bag, along with the value of each.

(a) The bag contains n coins, but only 2 different denominations: some coins are worth x dollars, and some are worth y dollars. Bonnie and Clyde wish to divide the money exactly evenly.

(b) The bag contains n coins, with an arbitrary number of different denominations, but each denomination is a non-negative integer power of 2, i.e., the possible denominations are 1 dollar, 2 dollars, 4 dollars, etc. Bonnie and Clyde wish to divide the money exactly evenly.

(c) The bag contains n checks, which are, in an amazing coincidence, made out to "Bonnie or Clyde." They wish to divide the checks so that they each get the exact same amount of money.

(d) The bag contains n checks as in part (c), but this time Bonnie and Clyde are willing to accept a split in which the difference is no larger than 100 dollars.

Solution: *We will prove somethings which will be useful in solving this problem. First we will assume that 3-SAT is NP-complete, and next we will show that Subset-Sum problem is NP-complete by reducing it to 3-SAT.*

Definition: (Subset Sum) *Given a set A of integer numbers and an integer number s, does there exist a subset of A such that the sum of its elements is equal to s?*

*We will show that this is a **NP-complete** problem. We can verify if subset B is a set of*

Number	i			j			
	1	2	3	1	2	3	4
t_1	1	0	0	1	0	0	1
t_2	0	1	0	1	0	1	0
t_3	0	0	1	1	1	0	1
x_2	0	0	0	0	1	0	0
y_2	0	0	0	0	1	0	0
x_3	0	0	0	0	0	1	0
y_3	0	0	0	0	0	1	0
x_4	0	0	0	0	0	0	1
s	1	1	1	3	3	3	3

Figure 5: Reducing 3SAT to Subset problem

numbers whose sum is s i.e., verify if $\sum_{a \in B} a = s$. We will show

$$3\text{-}SAT \leq_P \text{Subset-Sum}.$$

Reduction: Lets consider a 3-SAT problem with n variables x_i and m clauses c_j. For example consider following SAT problem

$$(x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_3}) \wedge (x_1 \vee \overline{x_2} \vee x_3).$$

For each variable x_i, construct numbers t_i and f_i of $n + m$ digits. Please refer to **Fig.3**.

- The i_{th} digit of t_j and f_i is equal to 1.
- For $n + 1 \leq j \leq n + m$ the jth digit of t_i is equal to 1 if x_i is in clause c_{j-n}.
- For $n + 1 \leq j \leq n + m$ the jth digit of f_i is equal to 1 if $\overline{x_i}$ is in clause c_{j-n}.
- All other digits of t_i and f_i are 0.

For each clause c_j construct numbers x_j and y_j of $n + m$ digits. Please refer to **Fig.4**.

- The $(n + j)^{th}$ digit of x_j and y_j is equal to 1.
- All other digits of x_j and y_j are 0.

Finally, construct a sum number s of $n + m$ digits. Please refer to **Fig.5**.

- For $1 \leq j \leq n$, the jth digit of s is equal to 1.
- For $n + 1 \leq j \leq n + m$, the jth digit of s is equal to 3.

Proof: ($\implies$) We will prove that if there is satisfiable solution for 3-SAT then there is a solution for Subset-Sum.

- Take t_i if x_i is true.
- Take f_i if x_i is false.
- Take x_j if number of true literals in c_j is at most 2.
- Take y_j if number of true literals in c_j is 1.

($\Longleftarrow$) *We will prove that if there is a solution for Subset-Sum then there is a solution for 3-SAT.*

- *Assign value true to x_i if t_i is in subset.*

- *Assign value false to x_i if f_i is in subset.*

- *Exactly one number per variable must be in the subset. Otherwise one of first n digits of the sum is greater than 1*

- *We have found an assignment for variables which is consistent.*

- *At least one variable number corresponding to a literal in a clause must be in the subset. Otherwise one of next m digits of the sum is smaller than 3.*

- *Each clause is satisfied. Hence we have found an satisfiable solution for 3-SAT.* $\square$

(a) *We can solve this problem in **polynomial time** as follows. Let a denote the number of coins of denomination x and b denote the number of coins of denomination y. Then we must have $a + b = n$. In order to divide the money exactly evenly between Bonnie and Clyde, we need to know if there is a way to make an amount $A = (ax + by)/2$ out of the coins. In other words, we need to determine whether there exist non-negative integers c and d less than or equal to a and b respectively such that $cx + dy = (ax + by)/2$. If Bonnie gets c coins of value x and d coins of value y then he total amount he gets should be $cx + dy$ and we want this to be equal to A. Similarly Clyde would get $a - c$ coins of value x and $b - d$ coins of value y and again we want $(a - c)x + (b - d)y$ to be equal to A. If such pair of non-negative integers (c, d) doesn't exist then the algorithm must be tell us that it is not possible to split the bounty evenly.*

Since a, b, n are finite integers we can test all possible combinations of c, d which would satisfy above condition. In linear amount of time we can count how many coins of value x are present and y are present which would give us integers a, b. Then there are $(a + 1)(b + 1) \le (n + 1)^2$ many such combinations of possible c, d exists. We can compute each one in time polynomial in the length of the input numbers, and there are polynomial many combinations to compute, so we can just check all combinations to see if we find one that works.

(b) *We can solve this problem in **polynomial time** as follows. Start by arranging the coins from largest to smallest denomination. If there are an even number of the current largest coin, distribute them evenly to Bonnie and Clyde. Otherwise, give the extra one to Bonnie and then only give coins to Clyde until the difference has been resolved. This clearly runs in polynomial time, so we just need to show that this will always yield an even split of the bounty if such a division is possible.*

We can argue that this algorithm will always and tell us whether there exists way in which we can split the bounty evenly. Suppose assume that we have even number of coins for all the denominations then this algorithm will work because we for each denomination we would give half of the coins to Bonnie and Clyde. This is a trivial case. Now lets consider a case where for 2^ith denomination we have odd number of coins. According to our algorithm we will give that one additional coin to Bonnie. This will create a difference

57

2^i value between Bonnie and Clyde. Now, our next denomination would be 2^{i-1} and to make the money difference between Bonnie and Clyde zero we will give 2 coins of value 2^{i-1} to Clyde first. (It can also be 1 coin of 2^{i-1} and 2 coins of 2^{i-2} or 4 coins of 2^{i-2} or some other combinations of coins of denomination equal to a power of 2 which is strictly less than 2^i. Lets assume that in the process of making the net worth zero between Bonnie and Clyde ends up at denomination 2^{i-k}, where $k \leq i$). Next we will get two cases

- *If number of coins (of denomination 2^{i-k}) left after giving the first two to Clyde is even then we are done. Because we can just split them in half and give it to Bonnie and Clyde equally. And proceed to next denomination.*

- *If number of coins (of denomination 2^{i-k}) left after giving the first two to Clyde is odd then we have a similar situation as above and we will give the Bonnie the additional coin. After that we proceed to next denomination and give first two coins to Clyde first to make the net worth difference zero.*

If we follow the above steps and split the bounty then we either we split the bounty equally between Bonnie and Clyde or we may end up in the last denomination where we can't split the coins between them such a way that net worth between them becomes zero.

(c) *This problem is **NP-complete** problem. Here we consider the bag as set S where the elements are the values present on the check, and we are interested in finding a partitioning the set into two smaller disjoint subsets such that sum of all elements in both of the sets are equal. In the literature this problem is known as Partitioning Problem. and finally we show that **Partitioning Problem** NP-complete by reducing it to Subset-Sum problem.*

Definition: *(**Set Partition**) Given a set S of integer numbers. The question is whether the numbers can be partitioned into two sets A and $\overline{A} = S - A$ such that*

$$\sum_{x \in A} x = \sum_{x \in \overline{A}} x.$$

We will now show that Set Partition is NP-Complete. First we will show that there is poly-time certificate to tell if the given solution is correct. That is guess the two partitions and verify that the two have equal sums. We will now see how to do reduction of Subset Sum to Set Partition. Recall Subset Sum is defined as follows, given a set X of integers and a target number t, find a subset $Y \subseteq X$ such that the members of Y add up to exactly t. Let s be the sum of members of X. Feed $X' = X \cup (s - 2t)$ into Set Partition. Accept if and only if Set Partition accepts. This reduction clearly works in polynomial time.

We will prove that $(X, t) \in$ Subset Sum if and only if $X' \in$ Set Partition. Note that sum of members of X' is $2s - 2t$.

Proof: *($\implies$) If there exists a set of numbers in X that sum to t, then the remaining numbers in X sum to $s - t$. Therefore, there exists a partition of X' into two such that each partition sums to $s - t$.*

($\impliedby$) Let's say that there exists a partition of X' into two sets such that the sum over each set is $s - t$. One of these sets contains the number $s - 2t$. Removing this number, we get a set of numbers whose sum is t, and all of these numbers are in X. $\square$

Idea behind this proof is

- *Add in a new element to the set such that a subset with the appropriate sum also forms a partition.*

- *The new element added in might need to go in the subset that originally added to t, or it might have to go in the complement of that set.*

Hence we have showed that Set Partition problem is NP-complete which in turn means that the problem of splitting checks is NP-complete.

*(d) This problem is **NP-complete** problem. Let us the name this problem as Split-Checks-100. An assignment of each check to either Bonnie or Clyde represents a certificate, and we can check in polynomial time whether or not the total amounts given to Bonnie and Clyde differ by at most 100. Thus, the problem is in NP. Next, we'll show it's NP-hard by showing Set Partition $\leq_P$ Split-Checks-100. Let $S = s_1, s_2, \ldots, s_n$ be a set, and let $kS = ks_1, ks_2, \ldots, ks_n$. Choose $k = \frac{101}{min_{i,j} s_i - s_j}$. Then the difference between any two elements in kS is more than 100. If there exists $A \subseteq S$ such that $\sum_{x \in A} x = \sum_{x \in \overline{A}} x$, then we give Bonnie all the checks in kA and Clyde all the checks in $k(S - A) = k\overline{A}$, for a perfectly even split of the money, which means the difference is less than 100. On the other hand, suppose there exists no such A. Then for any way of splitting the elements of S into two sets, their sums will differ by at least the minimum difference of elements in S. Thus, any way of splitting the checks in kS will result in a difference of at least 101, so there is no way to split them so that the difference is at most 100. Therefore Set Partition $\leq_P$ Split-Checks-100, so the problem is NP-hard. Since we showed it is in NP, it is also NP-complete.*

36. (**Galactic Shortest Path**) This problem originates from the *Star Wars* movies. Luke, Leia, and friends are trying to make their way from the Death Star back to the hidden Rebel base. We can view the galaxy as an undirected graph $G = (V, E)$, where each node is a star system and an edge (u, v) indicates that one can travel directly between u and v. The Death Star is represented by a node s, the hidden Rebel base by a node t. Certain edges in this graph represent longer distances than others; thus each edge e has an integer length $\ell_e \geq 0$. Also, certain edges represent routes that are more heavily patrolled by evil Imperial spacecraft; so each edge e also has an integer risk $r_e \geq 0$, indicating the expected amount of damage incurred from special-effects-intensive space battles if one traverses this edge.

It would be safest to travel through the outer rim of the galaxy, from one quiet upstate star system to another; but then one's ship would run out of fuel long before getting to its destination. Alternately, it would be quickest to plunge through the cosmopolitan core of the galaxy; but then there would be far too many Imperial spacecraft to deal with. In general, for any path P from s to t, we can define its total length to be the sum of the lengths of all its edges; and we can define its total risk to be the sum of the risks of all its edges.

So Luke, Leia, and company are looking at a complex type of shortest path problem in this graph: they need to get from s to t along a path whose total length and total risk

are both reasonably small. In concrete terms, we can phrase the *Galactic Shortest-Path Problem* as follows: Given a setup as above, and integer bounds L and R, is there a path from s to t whose total length is at most L, and whose total risk is at most R?

Describe a poly-time algorithm or prove that Galactic Shortest Path is NP-complete.

Solution: This problem is **NP-complete** problem. Let us the name this problem as Galactic Shortest Path. Galactic Shortest Path is in NP. Given a path P we can add up the lengths and risks of its edges, and compare them to given bounds L and R. We will reduce this problem to Subset Sum problem. We will prove that Subset Sum $\leq_P$ Galactic Shortest Path.

Consider an instance of Subset Sum specified by the numbers $w_1, \ldots, w_n$ and a bound W; we want to know that if there is a subset S of these numbers such that they add up exactly to W. We define the instance of Galactic Shortest Path as graph G with vertices $v_0, v_1, \ldots, v_n$. Let there be two edges e_i and e_i' from v_{i-1} to v_i for all $i \in (1, n)$. Now any path from v_0 to v_n in the graph G goes through edges either e_i or e_i'. This corresponds to making n independent binary choices much like binary choices one has in Subset Sum i.e., we have to make a choice whether this element should be in our desired subset. Here choosing e_i will represent putting w_i into our set S and e_i' will represent leaving it out.

Let $W_0 = \sum_i w_i$ sum of all elements in the set. Then a subset S adds up to W iff its complement adds up to $W_0 - W$. We give e_i length equal to w_i and risk 0 and e_i' length equal to 0 and risk w_i. We set the bounds $L = W$ and risk $R = W_0 - W$. We now claim that there is a solution to the Subset Sum instance iff there is a valid path in G. For if there is a set S adding up to W, then in G we use the edge e_i for $i \in S$ and e_j' for $j \notin S$. This path has length W and risk $W_0 - W$, so its meets the given bounds. Conversely, if there is path P meeting the given bounds, then consider the set $S = \{w_i | e_i \in P\}$. S adds up to at most W and its complement adds up to at most $W_0 - W$. But since the two sets together add up to exactly W_0, it must be that S adds up to exactly W and its complement to exactly $W_0 - W$. Thus, S is valid solution to the Subset Sum instance.

37. A clique in an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ of vertices, each pair of which is connected by an edge in E. (In other words, a clique is a *complete* subgraph of G). The size of a clique is the number of vertices it contains. The Clique problems asks to decide whether a clique of a given size k exists in the graph. Show that Clique is NP-complete.

38. Given an integer $m \times n$ matrix A and an integer m-vector b, the 0-1 integer programming problem asks whether there exists an integer n-vector x with elements in the set $\{0,1\}$such that $Ax = b$. Prove that 0-1 integer programming is NP-complete. (Hint: Reduce from 3-CNF-SAT.)

7 Randomized Algorithms and Approximation Algorithms

39. (**Hat-check**) Each of n customers gives a hat to a hat-check person at a restaurant. The hat-check person gives the hats back to the customers in a uniformly random order. What is the expected number of customers who get back their own hat?

    Solution: Let X be the number of customers who get back their own hat and X_i be the indicator random variable that customer i gets his hat back. The probability that an individual gets his hat back is $1/n$. Then we have

    $$E[X] = E\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n} \frac{1}{n} = 1.$$

40. (**Streaks**) Suppose you flip a fair coin n times. What is the longest streak of consecutive heads that you expect to see?

41. (**Happy Holidays**) This time of year is here, you known, holidays, and everyone is facing various (joyful) challenges.

    (a) For Kevin (character in movie "Home Alone"), he is handed n pieces of candy with weights $W[1, \ldots, n]$ (in ounces) that he needs to load into boxes. The goal is to load the candy into as many boxes as possible, so that each box contains at least L ounces of candy. Help Kevin solve this by describing an efficient 2-approximation algorithm for this problem. Prove that the approximation ratio of your algorithm is 2. [Hint: First consider the case where every piece of candy weighs less than L ounces.]

    (b) For Juliet, this means that she needs to fulfill Romeo's wishlist - n Nintendo® Switch games. She happened to notice that in a personal care and beauty store AROHPES, her favorite lip balm includes a 99% discount code for one of the n games Romeo wants. However, the code is hidden at the bottom of the package, which Juliet cannot tell before purchasing and opening it, and the code comes in random so that it is equally likely to be good for any of the n games. She intends to purchase sufficient lip balms, which she likes and needs anyways, so she gets discount codes for all n games. How many lip balms does she need to buy *in expectation* before getting a code for each game?

    (c) Continuing from Part (b), it's 7 days till Thanksgiving now. Due to popularity, AROHPES limits the number of this particular lip balms one can purchase in a day to 6. She convinced Romeo to reduce his demand to 10 games. Can Juliet succeed in collecting all discount codes by Thanksgiving? [Hint: look up Markov's inequality]

Solution:

(a) *Let OPT denote the maximum number of bins possible by the optimal algorithm for the weight vector W. Assign an empty box as the current box and consider the following greedy algorithm:*

- *Consider n candies in some fixed order and the associated weights are given by $W[1], W[2], \ldots, W[n]$.*
- *Assign candy i to jth current box if the weight of jth current box is less than L, otherwise assign to a new empty box which becomes the $(j+1)$th current box.*

Above algorithm can be summarised as first-fit heuristic where we take each object in turn and places it into the first bin that can accommodate it. Let P denote the maximum number of boxes possible by the greedy algorithm for weight vector W.

Claim: *Suppose every candy weighs less than L ounces. Then greedy algorithm (first-fit heuristic) is a 2-approximation algorithm for all weight vector W of size n. That is $\frac{OPT}{P} \leq 2$.*

Proof: *Some trivial observations:*

- *$OPT < n$, because if all the candies weighed more than L then we would keep one candy in one bin. Which means that we need only n boxes. Here we assumed that each candy weighs less than L hence OPT is strictly less than L.*
- *$OPT \leq \frac{1}{L} \sum_{i=1}^{n} W[i]$, because if we were to have even more bin extra than $\frac{1}{L} \sum_{i=1}^{n} W[i]$ then we would end up with a bin with total weight less than L.*

Based on greedy algorithm we have P bins. Weight of each bin is some value between L and 2L, because weight of all candies is less than L. So as soon as we add a candy to i^{th} bin with weight less than L, total weight of that bin can only be slightly more than L. That is $b_j < 2L$ and $\sum_{i=1}^{P} b_j < 2LP$ where b_j denotes the weight in each box j. Since $\sum_{i=1}^{P} b_j = \sum_{i=1}^{n} W[i]$, we have $\sum_{i=1}^{n} W[i] \leq 2LP$ which implies $OPT \leq 2P$. $\square$

Claim: *Suppose there n_0 candies with weights less than L and there are $n - n_0$ weights greater than L. Then greedy algorithm is a 2-approximation algorithm for all weight vector W of size n. That is $\frac{OPT}{P} \leq 2$.*

Proof: *We will modify our greedy algorithm. First we will arrange $n - n_0$ candies in individual bins because it is the optimal way of arranging them. For the remaining n_0 candies use the above greedy algorithm. Let the output of greedy algorithm for n_0 weights be P_0. We obtain $OPT_0 \leq 2P_0$. Let output of our modified greedy algorithm be P. Then $P = P_0 + n - n_0$. We have*

$$\begin{aligned}
OPT &= OPT_0 + n - n_0 \\
&\leq 2P_0 + n - n_0 \\
&= P + P_0 \\
&\leq 2P. \qquad \square
\end{aligned}$$

(b) *We can represent this problem as a finite state machine problem to get a better understanding of the underlying structure. Consider we have n states and at each state i we*

have a probability of $(n - i)/n$ to go the next state and probability of i/n for being in the same state. Here these states represent the events where we have i discount codes for i different games and these probabilities tells us chance to transition to a next state where we get a coupon for next unique game or get stuck in the same state.

This can be understood as follows:

- *Assume we went to the store and bought first lip balm then we would get a discount code for one of the n games.*

- *Next if we buy another lip balm then with probability of $(n - 1)/n$ we can get code for another game or with probability of $1/n$ we will get code for the same game. This can be treated as geometric distribution i.e. the probability distribution of the number X_1 of Bernoulli trials needed to get one success. And we know that on average we would need $1/p$ trails where p is the probability of success. Hence we would need to buy (in the order of) $\mathbb{E}[X_1] = n/(n - 1)$ lip balms on an average to get code for another unique game.*

- *Using argument provided above we can see that if we were to buy lip balms for to get a discount code for i^{th} game (represented by random variable X_i which admits geometric distribution) then on an average we would need to buy $\mathbb{E}[X_i] = n/(n - i)$ lip balms. We can see that as i increase it becomes harder and harder to get a new coupon.*

- *Let X_i represent i^{th} random variable which follows geometric distribution. It represents the number of Bernoulli trials (no. of lip balms that must be bought) needed to get one success i.e. getting discount code for i^{th} game. Let X represents no of lip balms that are required to bought to get discount codes for all n games. Then*

$$
\begin{aligned}
\mathbb{E}[X] &= \mathbb{E}[\sum_{i=1}^{n} X_i] \\
&= \sum_{i=1}^{n} \mathbb{E}[X_i] \\
&= 1 + \sum_{i=2}^{n} \mathbb{E}[X_i] \\
&= 1 + n \sum_{i=1}^{n-1} \frac{1}{n - i}
\end{aligned}
$$

- *Hence to get discount coupons for all the game we need to buy $1 + n \sum_{i=1}^{n-1} \frac{1}{n-i}$ lip balms on an average.*

If we expand the summation in the above equation we see that it is Harmonic series and it grows in terms of $O(\log n)$. Hence we need to buy $O(n \log n)$ lip balms to get discount codes for all the games. Juliet needs to buy $n \log n$ lip balms in expectation before getting a code for each game.

(c) From the question we can conclude before the holidays start we can buy only 42 lip balms. No. of games that Romeo demands is 10 i.e. $n = 10$. Using the same definition for X as mentioned above, we are interested in looking at $Pr\{X \geq 42\}$ i.e. how likely it is that we would need to buy more than 42 lip balms to get discount codes for 10 games. Using Markov's inequality we can compute a upper bound for this probability as follows

$$Pr\{X \geq 42\} \leq \frac{\mathbb{E}[X]}{42}.$$

From above we know $\mathbb{E}[X] = n \log n$ where n is no. of games which 10 so $\mathbb{E}[X] = 10 \log 10 \approx 33.22$. Now we can calculate

$$Pr\{X \geq 42\} \leq 0.7909$$

which means that with probability of 79% we would need to buy more than 42 lip balms to get discount codes for all games. Since this value is very high Juliet may not succeed in collecting all discount codes by Thanksgiving. Note: Markov bounds are not tight bounds.

42. (**3-Coloring**) Suppose you are given a graph $G = (V, E)$, and we want to color each node with one of three colors, even if we aren't necessarily able to give different colors to every pair of adjacent nodes. We say an edge (u, v) is satisfied if the colors assigned to u and v are different. Consider a coloring scheme that maximizes the number of satisfied edges, and let c^* denote this number. Give a poly-time algorithm that produces a coloring that satisfies at least $\frac{2}{3}c^*$ edges. If you want to use an randomized algorithm, the *expected* number of edges it satisfies should be at least $\frac{2}{3}c^*$.

Solution: *Let's look at some definitions that are useful for solving this problem. Suppose we have sample space S and an event A. Then the **indicator random variable** $X_A = I\{A\}$ associated with event A is defined as*

$$I\{A\} = \begin{cases} 1 & \text{if A occurs} \\ 0 & \text{if A does not occur} \end{cases}$$

and the immediate consequence of this would be that for random variable X_A, the expected value of it $E[X_A]$ would be equal to probability of the event A occurring i.e., $Pr\{A\}$.

Now consider an algorithm that picks a color for each vertex uniformly and independently at random. Then for each edge $e \in E$, the probability that e is satisfied is 2/3. Define the indicator random variables X_e for $e \in E$ which be equal to 1 iff edge e is satisfied otherwise zero. Let X be random variable defined as total number of edges that will be satisfied by the

algorithm that we have picked, then

$$\mathbb{E}[X] = \mathbb{E}\left[\sum_{e \in E} X_e\right]$$
$$= \sum_{e \in E} \mathbb{E}[X_e]$$
$$= \sum_{e \in E} Pr\{e \text{ is satisfied}\}.$$

Now, noting that the optimal number of satisfied edges can be no more than the total number of edges, i.e. $c^* \leq |E|$, we have for our algorithm $\mathbb{E}[X] = \frac{2}{3}|E| \geq \frac{2}{3}c^*$.

43. (**Errors in randomized algorithms**) Suppose you want to write a computer program C to compute a Boolean function $f : \{0,1\}^n \to \{0,1\}$, mapping n bits to 1 bit. If C is a deterministic algorithm, then "C successfully computes f" has a clear meaning that that $C(x) = f(x)$ for all inputs $x \in \{0,1\}^n$. But what if C is a randomized algorithm?

    (a) The best thing is if C is a *zero-error* algorithm with failure probability p. Namely

    - on every input x, the output of $C(x)$ is either $f(x)$ or $\perp$ (denoting failure).
    - on every input x we have $Pr[C(x) = \perp] \leq p$ (NB. the probability is only over the internal randomness of C, not the random choice of x.).

      i) If you have a zero-error algorithm C for f with failure probability 90%, show how to convert it to a zero-error algorithm C' with failure probability at most 2^{-500}. The "slowdown" should only be a factor of a few thousand.
      ii) Alternatively, show how to convert C to an algorithm C'' for f which: (i) always outputs the correct answer, meaning $C''(x) = f(x)$ for all x; (ii) has expected running time only a few powers of 2 worse than that of C. (Hint: look up the mean of a geometric random variable.)

    (b) The second best thing is if C is a one-sided error algorithm for f, with failure probability p. There are two kinds of such algorithms, "no-false-positives" and "no-false-negatives". For simplicity, let's just consider "no false-negatives" (the other case is symmetric);

    - on every input x, the output $C(x)$ is either 0 or 1;
    - on every input x such that $f(x) = 1$, the output $C(x)$ is also 1;
    - on every input x such that $f(x) = 0$, we have $Pr[C(x) = 1] \leq p$.

    Show how to convert a no-false-negatives algorithm C for f with failure probability 90% to another no-false-negatives algorithm C' for f with failure probability at most 2^{-500}. The "slowdown" should only be a factor of a few thousand.

    (c) The third possibility (which is rare in practice) is if C is a two-sided error algorithm for f, with failure probability p. Namely,

    - on every input x, the output $C(x)$ is either 0 or 1.
    - on every input x, we have $Pr[C(x) \neq f(x)] \leq p$.

If you have a two-sided error algorithm C for f with failure probability 40%, show how to convert it to a two-sided error algorithm C' for f with failure probability at most 2^{-500}. The "slowdown" should only be a factor of a few dozen thousand. (Hint: look up the Chernoff bound.)

Solution:

(a) (I) *Let $C'(x)$ be an algorithm where we run $C(x)$ n times such that if anyone of run gives us a valid output then we can conclude that $C'(x) = C(x)$ equals to the value $f(x)$. Also note that this algorithm we fail only when we end getting output for all the runs equal to $\perp$. The probability of this happening will be p^n where p is probability of the failure of $C(x)$ for one run. This can be return as $Pr\{C(x) = \perp\} \leq p$ where $p = 0.9$. Also, from the question we know that we want $Pr\{C'(x) = \perp$ for all n runs$\} \leq p^n$ and we want this value to be 2^{-500}. Equating $p^n = 2^{-500}$ gives us the value of n to be ≈ 3290. This tell us that by repeating the same random algorithm for few thousand times we can reduce chances of the failure of the algorithm to near zero chance.*

(II) *Here we have converted the problem into the probability distribution of the number of Bernoulli trials needed to get one success i.e. how many times we have to run our algorithm $C(x)$ (Bernoulli trial) to get valid output $C(x) \neq \perp$ or $C''(x) = C(x) = f(x)$ (one success). The distribution we get is nothing but a geometric distribution. From basic probability theory we know the expected number of failures before the first success of a geometrically distributed random variable X is $1/p$ and the variance is $(1-p)/p^2$ where p is the probability of success.*

Using the above identity for any given input $x \in \{0,1\}^n$ the expected number of runs for seeing first success would be $\frac{1}{1-p} = 10$, where $p = 0.9$. In total we can have 2^n possible inputs, hence if we want our algorithm to work on all inputs we have to run $C(x)$ algorithm in the order of $O(2^n)$ and expected runtime would be runtime of $C(x)$ multiplied with few powers of 2. Our algorithm $C''(x)$ will be running $C(x)$ in the order of $O(2^n)$.

Above arguments comes from the following reasoning. If we define a random variable X be the case where our algorithm works for all possible inputs then this can written as sum of random variables X_i which will tell us how many trials are required for the algorithm to work for i^{th} input i.e. when we see the first success (similar to what we did in previous problem). Using this definition we can write

$$E[X] = E[\sum_i X_i]$$

$$= \sum_i E[X_i]$$

$$= \sum_i \frac{1}{1-p}$$

$$= \frac{2^n}{1-p} = 10 * 2^n$$

(c) *Here our algorithm $C'(x)$ would be running no-false-negative algorithm $C(x)$ n times. If we get output zero in one or more runs then we can conclude for sure that $f(x) = 0$ because we can get output of $C(x)$ to be zero only when $f(x) = 0$. Finally, if we get output to be one for all the n runs then we can conclude that $f(x) = 1$.*

But we know that when $f(x) = 0$, we have non zero error probability i.e. $Pr\{C(x) = 1\} \le p$. Hence when we saw all the outputs of n runs to be equal to one then it could be possible that $f(x)$ was zero. We want to reduce this error and we notice that if $f(x) = 0$ and for all runs we saw output to be one then probability of this happening will be equal to p^n, where p is 0.9. According to the problem we want this to be equal to 2^{-500} (extremely low value; we can assumed that occurrence of this event has zero chance), equating the values will give n to be ≈ 3290.

(d) *Here our algorithm $C'(x)$ would be running the $C(x)$ n time and observe which values occurs the most i.e. what is output of $C(x)$ that occurs more than 50% of the time. That is if we see the output of $C(x)$ is zero for more than 50% of the time then we would assume that $C(x) = f(x) = 0$. Similarly we can argue for when output of $C(x)$ is one most of the time.*

If we treat each run X_i as an independent Bernoulli trial, where we are flipping a coin with one face saying $C(x) = f(x)$ (heads) and other face saying $C(x) \ne f(x)$ (tails). We see heads with a probability of $Pr\{C(x) = f(x)\} = 1 - p = 0.6$ and tails with a probability of $Pr\{C(x) \ne f(x)\} = p = 0.4$. Let sum of these Bernoulli trails be X. Then we can write the probability of simultaneous occurrence of more than n/2 of the events $C(x) = f(x)$ has an exact value

$$Pr\{X > \frac{n}{2}\} = \sum_{i=\lfloor \frac{n}{2} \rfloor + 1}^{n} \binom{n}{i} (1-p)^i p^{n-i}.$$

A lower bound on this probability can be calculated based on Chernoff's inequality

$$Pr\{X > \frac{n}{2}\} \ge 1 - e^{-\frac{n}{2(1-p)}((1-p)-\frac{1}{2})^2}.$$

According to the question we want this to be equal to $1 - 2^{-500}$. Now equating both values gives us

$$1 - e^{-\frac{n}{2(1-p)}((1-p)-\frac{1}{2})^2} = 1 - 2^{-500}$$

$$e^{-\frac{n}{2(1-p)}((1-p)-\frac{1}{2})^2} = 2^{-500}$$

$$n = 41588.831$$

$$\approx 41589.$$

This tell us what our $C'(x)$ would be, run $C(x)$ few dozen thousands of time and observe the value that we get the most i.e. take the majority as our final output. And this will be correct most of times and wrong with probability of 2^{-500} which is extremely low. We can assume that event where $C(x) \ne f(x)$ would be of zero chance for all practical purposes.

8 Computational Geometry

44. Suppose you are given a set L of n line segments in the plane, where each segment has one endpoint on the vertical line $x = 0$ and one endpoint on the vertical line $x = 1$, and all $2n$ endpoints are distinct. Describe and analyze an algorithm to compute the largest subset of L in which no pair of segments intersects.

Solution: Here I have provided a solution based on Computational Geometry Theory. This toy problem has been used to introduce the paradigm of computational geometry algorithms. (Sweeping line algorithms can be used to solve many problems in this area). This algorithm might be an overkill but demonstrates its robustness.

Algorithm based on Computational Geometry Theory:
We will develop first a general algorithm which will work for this simplified case also. This algorithm is based on the sweeping line algorithm. A line is swept across the plane and when ever this line interacts with other objects in the 2D plane we study its interactions to develop a an algorithm. From the problem description we notice that there are no vertical lines in the give set of lines L. Also, all the endpoints are distinct i.e. no segment has an endpoint that is part of another segment.

First we will develop an idea for doing line segment intersection testing. Notice that 2D space when we dot product between two vectors we get the signed area of the parallelogram. If the cross product between two vectors v_1 and v_2 is positive indicates that v_1 is in clockwise direction from v_2 with respect to the origin and if the cross product is negative then then v_1 is counterclockwise from v_2. We use this idea to determine whether two line segments intersect, we check whether each segment straddles the line containing the other. A segment $l_1 = (p_1, p_2)$ straddles a line $l_2 = (p_3, p_4)$ if point p_1 lies on one side of the line l_2 and point p_2 lies on the other side. A boundary case arises if p_1 or p_2 lies directly on the line. Two line segments intersect if and only if either (or both) of the following conditions holds:

- *Each segment straddles the line containing the other.*

- *An endpoint of one segment lies on the other segment. (This condition comes from the boundary case.)*

If l_1 and l_2 intersect then value of the cross product $(p_3 - p_1) \times (p_2 - p_1)$ and $(p_4 - p_1) \times (p_2 - p_1)$ will differ in sign. This gives us a brute force algorithm (trivial) of running time $O(n^2)$, just apply segment intersection to all pairs of segments. Here n is total no. of line segments in set L.

Sweeping line algorithm will give better performance. In sweeping, an imaginary vertical sweep line passes through the given set of geometric objects, usually from left to right. We treat the spatial dimension that the sweep line moves across, in this case the x-dimension, as a dimension of time. Sweeping provides a method for ordering geometric objects, usually by placing them into a dynamic data structure, and for taking advantage of relationships among them. The line-segment-intersection algorithm in this section considers all the line-segment

endpoints in left-to-right order and checks for an intersection each time it encounters an endpoint.

Ordering of segments: *Because we assume that there are no vertical segments, we know that any input segment intersecting a given vertical sweep line intersects it at a single point. Thus, we can order the segments that intersect a vertical sweep line according to the y coordinates of the points of intersection. To be more precise, consider two segments l_1 and l_2. We say that these segments are comparable at x if the vertical sweep line with x-coordinate x intersects both of them. We say that l_1 is above l_2 at x, written $l_1 \succeq_x l_1$, if l_1 and l_1 are comparable at x and the intersection of l_1 with the sweep line at x is higher than the intersection of l_2 with the same sweep line, or if l_1 and l_2 intersect at the sweep line. This give us an idea how our algorithm will work; by keeping an track of the ordering of the segments we can see observe when a change in ordering occurs then might be due to intersection of line segments.*

Moving sweep line: *Sweeping algorithms typically manage two sets of data:*

- *The **sweep-line status (T) or segment list (SL)** gives the relationships among the objects that the sweep line intersects.*

- *The **event-point schedule or event queue (EQ)** is a sequence of points, called event points, which we order from left to right according to their x-coordinates. As the sweep progresses from left to right, whenever the sweep line reaches the x-coordinate of an event point, the sweep halts, processes the event point, and then resumes. Changes to the sweep-line status occur only at event points.*

Each segment endpoint is an event point. We sort the segment endpoints by increasing x-coordinate and proceed from left to right. If two or more endpoints are covertical (in our case all have same left and right endpoint), i.e., they have the same x-coordinate, we break the tie by putting all the covertical left endpoints before the covertical right endpoints. Within a set of covertical left endpoints, we put those with lower y-coordinates first, and we do the same within a set of covertical right endpoints. When we encounter a segment's left endpoint, we insert the segment into the sweep-line status, and we delete the segment from the sweep-line status upon encountering its right endpoint.

The "interesting" x-coordinates are the ones where something happens: either a segment begins, or a segment ends, or two segments cross. Between any two neighboring event points, the segments (in that range of x) are in some order by y from bottom to top. This order (and which segments are there) does not change between a pair of neighboring interesting x coordinates. One of the three things can happen to the data structure segment list (a) A new segment is inserted into the list. (b) A segment is removed from the list. (c) Two neighboring segments in the list cross. Our goal is to compute the intersections among these segments. The key observation that leads to a good algorithm is that right before two segments cross, they must be neighbors in the segment list. So the key idea of the algorithm is that as our segment list evolves (as we process the interesting x coordinates from left to right), we only need to consider the possible intersections between segments that are neighbors, at some point in time, in the segment list.

Although a segment is defined by its two endpoints, it's going to be useful to also use a 'slope-intercept' representation for the line containing the segment. So segment l_i will have a

left and right endpoint as well as a pair (m_i, b_i) where m_i is the slope of the line and b_i is the y-intercept. Consider what we need for the SL data structure. The items being stored are a set of segments. The ordering used is the value of $m_i * x + b_i$, where x is the current x value in the ongoing sweep-line algorithm. Requirements for the SL data structure:

- *Insert: a new segment into the data structure.*
- *Delete: a segment from the data structure.*
- *Above: Find the successor of a segment in the data structure.*
- *Below: Find the predecessor of a segment in the data structure.*

All of these operations can be done in $O(\log n)$ time using any standard search tree data structure (e.g. splay trees, red-black-tree, or AVL trees). The ordering only changes at the interesting points, SL maintains the correct tree ordering. The EQ data structure must support insert, findmin, and deletemin. It's just a standard priority queue.

Now we can write the complete algorithm. Whenever the algorithm calls for CheckForIntersection(EQ, l_i, l_j), this means that the pair of segments l_i and l_j are tested for intersection. This function will always be called when l_i and l_j are neighbors in the SL. If they do intersect, CheckForIntersection adds the intersection event to the EQ.

> ### Algorithm: FindSegmentIntersections(L)
>
> *Copy L into L' Create an empty priority queue EQ*
> **for** *each segment in L*
>   *insert its start and end event points into EQ. Sort the endpoints of the segments in L from left to right, breaking ties by putting left endpoints before right endpoints and breaking further ties by putting points with lower y-coordinates first*
> *Create an empty balanced tree SL*
> **while** *EQ is not empty*
>   *E ← deletemin(EQ)*
>   **if** *E is start of line segment l* **then**
>   **Insert** *l into SL*
>   *CheckForIntersection(EQ, l, successor of l)*
>   *CheckForIntersection(EQ, l, predecessor of l)*
>   **else if** *E is the end of segment l* **then**
>   *CheckForIntersection(EQ, l, successor of l)*
>   **delete** *l from SL*
>   **else if** *E is cross event for segments l_i and l_j then* **then**
>   **print** *l_i and l_j intersect*
>   **remove** *l_i and l_j from SL*
>   *$L' \leftarrow L' \setminus \{l_i, l_i\}$*
>   *re-insert l_i and l_j into SL in the opposite order*
>   *CheckForIntersection(EQ, l_i, new neighbor of l_i)*
>   *CheckForIntersection(EQ, l_j, new neighbor of l_j)*
> **print** *Largest subset of L in which no pair of segments intersects is L'.*

It's easy to see that the running time of the algorithm is $O((n + k) \log n)$ using the data structures described. Because there are $O(n + k)$ events, and each one involves a constant number of operations on the SL and EQ data structures. Here k is the number of segment intersections found by the algorithm and n is the number of line segments/cardinality of the set L. Notice that sorting takes $O(n \log n)$ time, operations on the balanced tree take $O(\log n)$ time and the while loop is over $n + k$ event points and if we use the above described method based on slope-intercept line equation then checking for intersection using dot product will take constant time $O(1)$.

Correctness: *To show that FindSegmentIntersections is correct, we will prove that the call FindSegmentIntersections(L) returns TRUE (that is it will print out the intersections) if and only if there is an intersection among the segments in L. It is easy to see that FindSegmentIntersections returns TRUE (in last else if condition) only if it finds an intersection between two of the input segments. Hence, if it returns TRUE, there is an intersection.*

We also need to show the converse: that if there is an intersection, then FindSegmentIntersections returns TRUE. Let us suppose that there is at least one intersection. Let p be the leftmost intersection point, breaking ties by choosing the point with the lowest y-coordinate, and let l_i and l_j be the segments that intersect at p. Since no intersections occur to the left of p, the order given by SL is correct at all points to the left of p. Because no three segments intersect at the same point, l_i and l_j become consecutive in the SL at some sweep line z. Moreover, z is to the left of p or goes through p. Some segment endpoint q on sweep line z is the event point at which l_i and l_j become consecutive in the SL. If p is on sweep line z, then $q = p$. If p is not on sweep line z, then q is to the left of p. In either case, the order given by SL is correct just before encountering q. (Here is where we use the sorted order in which the algorithm processes event points. Because p is the lowest of the leftmost intersection points, even if p is on sweep line z and some other intersection point p' is on z, event point $q = p$ is processed before the other intersection p' can interfere with the SL. Moreover, even if p is the left endpoint of one segment, say l_i, and the right endpoint of the other segment, say l_j, because left endpoint events occur before right endpoint events, segment l_j is in SL upon first encountering segment l_i.) Either event point q is processed by FindSegmentIntersections or it is not processed.

If q is processed by FindSegmentIntersections, only two possible actions may occur:

- *Either l_i or l_j is inserted into SL , and the other segment is above (successor) or below (predecessor) it in the SL, 'if' detects this case in algorithm.*

- *Segments l_i and l_j are already in SL, and a segment between them in the SL is deleted, making l_i and l_j become consecutive, third 'else if' detects this case in the algorithm.*

In either case, we find the intersection p and FindSegmentIntersections returns TRUE. If event point q is not processed by FindSegmentIntersections, the procedure must have returned before processing all event points. This situation could have occurred only if FindSegmentIntersections had already found an intersection and returned TRUE.

Thus, if there is an intersection, FindSegmentIntersections returns TRUE. As we have already seen, if FindSegmentIntersections returns TRUE, there is an intersection. Therefore,

FindSegmentIntersections always returns a correct answer.

Similarly we can extend the proof for cases where three or more line segments intersect at the same point. This proof has been written with the help of the course textbook CLRS.