Collection of Interesting Problems in Algorithms

Sameer Kumar

November 2019

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:

$$\begin{aligned} T(n) &\leq 2T(\lceil n/2 \rceil) + n \\ &\leq 2T(n/2+1) + n \end{aligned}$$

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 $T(n+\alpha)$:

$$\begin{aligned} S(n) &\le 2S(n/2) + O(n) &&\implies && T(n+\alpha) \le 2T(n/2+\alpha) + O(n) \\ T(n) &\le 2T(n/2+1) + n &&\implies && T(n+\alpha) \le 2T((n+\alpha)/2+1) + n + \alpha \end{aligned}$$

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:

$$\begin{aligned} T(n) &\ge 2T(\lfloor n/2 \rfloor) + n \\ &\ge 2T(n/2-1) + n \end{aligned}$$

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 $S(n) \ge 2S(n/2) + O(n)$. To figure out the correct value of α, we compare two versions of the recurrence for the function $T(n+\alpha)$:

$$\begin{aligned} S(n) &\ge 2S(n/2) + O(n) &&\implies && T(n+\alpha) \ge 2T(n/2+\alpha) + O(n) \\ T(n) &\ge 2T(n/2-1) + n &&\implies && T(n+\alpha) \ge 2T((n+\alpha)/2-1) + n + \alpha \end{aligned}$$

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

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

$T(n)$ is the running time on a problem of size n. If the problem size is small enough, say $n \le c$ 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

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) \end{aligned}$$

$\Theta(n)$ 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 $n, n-1, n-2, \dots, 1$. 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 $T(n) = O(n \log n)$.

Note: 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

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

Now we can write a^2 as,

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

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,

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

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,

$$\begin{aligned} p(0) &= a_0 \\ p(1) &= a_2 + a_1 + a_0 \\ p(-1) &= a_2 - a_1 + a_0 \\ p(2) &= 4a_2 + 2a_1 + a_0 \\ p(-2) &= 4a_2 - 2a_1 + a_0. \end{aligned}$$

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

$$\begin{aligned} r(0) &= p(0)p(0) \\ &= a_0^2 \end{aligned}$$

$$\begin{aligned} r(1) &= p(1)p(1) \\ &= (a_2 + a_1 + a_0)^2 \\ &= a_2^2 + a_1^2 + a_0^2 + 2a_1 a_2 + 2a_2 a_0 + 2a_1 a_0. \end{aligned}$$

$$\begin{aligned} r(-1) &= p(-1)p(-1) \\ &= (a_2 - a_1 + a_0)^2 \\ &= a_2^2 + a_1^2 + a_0^2 - 2a_1 a_2 + 2a_2 a_0 - 2a_1 a_0 \end{aligned}$$

$$\begin{aligned}r(2) &= p(2)p(2)\\ &= (4a_2 + 2a_1 + a_0)^2\\ &= 16a_2^2 + 4a_1^2 + a_0^2 + 16a_1a_2 + 2a_2a_0 + 4a_1a_0\end{aligned}$$

$$\begin{aligned}r(-2) &= p(-2)p(-2)\\ &= (4a_2 - 2a_1 + a_0)^2\\ &= 16a_2^2 + 4a_1^2 + a_0^2 - 16a_1a_2 + 2a_2a_0 - 4a_1a_0.\end{aligned}$$

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

$$r(x) = r_4x^4 + r_3x^3 + r_2x^2 + r_1x + r_0$$

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

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

$$\begin{aligned}r_0 &= \frac{6r(0) - 4r(1) - 4r(2) + r(3) + r(4)}{24}\\ r_1 &= \frac{-2r(1) + 2r(2) + r(3) - r(4)}{12}\\ r_2 &= \frac{-30r(0) + 16r(1) + 16r(2) - r(3) - r(4)}{24}\\ r_3 &= \frac{8r(1) - 8r(2) - r(3) + r(4)}{12}\\ r_4 &= r(0).\end{aligned}$$

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,

$$\begin{aligned} r_4 &= a_2^2 \\ r_3 &= 2a_1a_2 \\ r_2 &= a_1^2 + 2a_2a_0 \\ r_1 &= 2a_1a_0 \\ r_0 &= a_0^2. \end{aligned}$$

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

$$\begin{aligned} a_2^2 &= r(0) \\ a_1a_2 &= \frac{8r(1) - 8r(2) - r(3) + r(4)}{24} \\ a_1^2 + 2a_2a_0 &= \frac{-30r(0) + 16r(1) + 16r(2) - r(3) - r(4)}{24} \\ a_1a_0 &= \frac{-2r(1) + 2r(2) + r(3) - r(4)}{24} \\ a_0^2 &= \frac{6r(0) - 4r(1) - 4r(2) + r(3) + r(4)}{24}. \end{aligned}$$

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.

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}
    $L[idx] = B[q]$
    $q = q + 1$
    $r_{AB} = r_{AB} + (mid - p + 1)$
    **for** $i = p$ to $mid$
      **print** *Inversion pair* $B[q]$, $A[i]$
  $idx = idx + 1$
**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 $r_{AB}$
```

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,

```
**Algorithm** *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 $A[p] \geq 2 * B[q]$
{Yes we have inversion pairs count them and print them}
    $q = q + 1$
    $r_{AB} = r_{AB} + (mid - p + 1)$
    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. *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. □

Lemma 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 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 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 2 and lemma 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

$$\begin{aligned} T(n) &= T(n/2) + O(n) \\ &= T(n/2) + n. \end{aligned}$$

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.

2 Data Structures, Amortized Analysis

1. Suppose we are maintaining a data structure under a series of n operations. Let $f(i)$ denote the actual running time of the i^{th} 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 i^{th} 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	2^0	0
18	2^1	1
19	2^0	0
20	2^0	0
21	2^0	0
22	2^1	1
23	2^0	0
24	2^3	3

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

$$\begin{aligned}\sum_{i=1}^{n} f(i) &= \sum_{i=1}^{m} 0 + \frac{n}{4}\sum_{i=1}^{m}\frac{i}{2^{i-1}}\\ &= \frac{n}{2}\sum_{i=1}^{m}\frac{i}{2^{i}}\\ &\le \frac{n}{2}\sum_{i=1}^{\infty}\frac{i}{2^{i}}.\end{aligned}$$

We will now evaluate the sum running to infinity.

$$\begin{aligned}S &= \sum_{i=1}^{\infty}\frac{i}{2^{i}}\\ &= \frac{1}{2}+\frac{2}{4}+\frac{3}{16}+\frac{4}{32}+\frac{5}{64}\dots\\ S/2 &= \frac{1}{4}+\frac{2}{8}+\frac{3}{32}+\frac{4}{64}+\frac{5}{128}\dots\\ S-S/2 &= \frac{1}{4}+\frac{1}{8}+\frac{1}{32}+\frac{1}{64}+\frac{1}{128}\dots\\ S/2 &= \sum_{i=1}^{\infty}\frac{1}{2^{i}}\\ S/2 &= 1\\ S &= \sum_{i=1}^{\infty}\frac{i}{2^{i}} = 2.\end{aligned}$$

Hence the total running time will be

$$\begin{aligned}\sum_{i=1}^{n} f(i) &\le \frac{n}{2}\sum_{i=1}^{\infty}\frac{i}{2^{i}}\\ &\le \frac{n}{2}*2\\ &\le n.\end{aligned}$$

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 i^{th} 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

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

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

1. (**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 $T[1,\ldots,n]$. 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 `RECURSION`, your algorithm should return 1, for the substring `R`.
   - Given the input string `REDIVIDE`, your algorithm should return 3, for the substring `EDI`. (The forward and backward substrings must not overlap!)
   - Given the input string `DYNAMICPROGRAMMINGMANYTIMES`, 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 $T[1,\ldots,n]$ 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.,

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

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

- **Case 1:** If the length of the string $T[i,\ldots,j]$ 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.

2. (**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 1: RFStops(D)
A ← 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, $1, \dots, j$: 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 $1, \dots, j$, then we have

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

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 2: RFStopsMinCost(D, C)
Initialize $M[1] = c_1$
for $j = 2, \dots, n$
  $M[j] \leftarrow \infty$
  $k \leftarrow j + 1$
  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 $1, \dots, j$ 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.

3. (**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 $L[1, \dots, n]$ and $R[1, \dots, n]$, 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.

Claim: 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 $R[1, \ldots, n]$).

Proof: 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 1: 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.

4 Graph Theory

1. (**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).

2. (**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.

3. (**Component graph**) Given a directed graph $G = (V, E)$, we define another graph $G^{\text{SCC}} = (V^{\text{SCC}}, E^{\text{SCC}})$ 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 $(v_i, v_j) \in E^{\text{SCC}}$ 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 $O(V + E)$.

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 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 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.

4. (**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.

5. (**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.

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

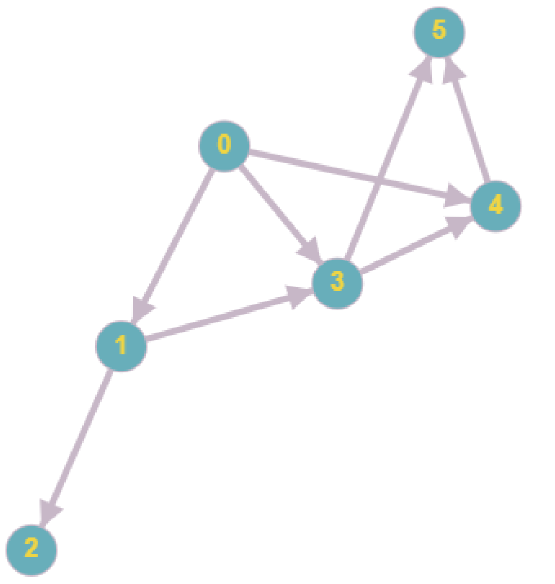

Figure 1: Example of a DAG that is not semi-connected graph

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 $v_1, v_2, \ldots, v_k$.

$\Longrightarrow$: 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.

$\Longleftarrow$: 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.

Observation: 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.

```
**Algorithm** *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 problem 1, 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.

```
Algorithm 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 $O(V + E)$.

6. (**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. □

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.

7. (**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 $x \to y$ has more weight than the actual edge present between them in original graph (assuming it is present, if not do nothing) G, i.e. $w''(x \to y) > w(x \to y)$ then we will replace the w'' with w. Also, note that by construction $w''(x \to y) = dist'(x \to y)$.

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.

8. (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: $G = T$.

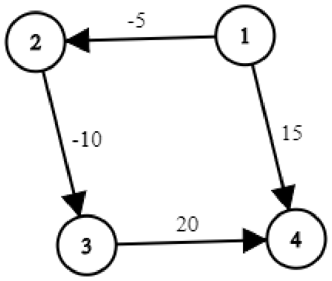

Figure 2: Counter example for modified Dijkstra's algorithm

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*** Note: 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 $G = T$.

9. (**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) **Algorithm 3:** *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.

10. (**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 2: MaxSpanTree(G, {w_e})
T ← φ // Kruskal's Algorithm
```

```
**Sort** $m$ edges so that $w(e_1) \geq w(e_2) \geq \dots w(e_m)$
**for** $v \in V$
  *MakeSet(v)*
**for** $i = 1, \dots, n$
  $(u, v) \leftarrow e_i$ //$i^{th}$ 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 $O(m \log n)$.

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

1. 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.

2. 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 $v \in S'$.

3. (**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 3: 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 $v \in V$
  $m \leftarrow \sum_{\text{e into v}} f(e) - \sum_{\text{e out of v}} f(e)$
    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 $c^{'} : E^{'} \to \mathcal{R}$ as follows:

- For each vertex $v \in V$, we set $c^{'}(s^{'} \to v) = \sum_{u \in V} d(u \to v)$ and $c^{'}(v \to t^{'}) = \sum_{w \in V} d(v \to w)$.

- For each edge $u \to v \in E$, we set $c'(v \to u) = d(u \to v)$.
- Finally, we set $c'(s \to t) = \infty$.

Intuitively, we construct G' by replacing any edge $u \to v$ in G with three edges: an edge $v \to u$ with capacity $d(u \to v)$, an edge $s' \to v$ with capacity $d(u \to v)$, and an edge $u \to t'$ with capacity $d(u \to v)$. 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 $f' : E' \to \mathcal{R}$:

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

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

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

Flow Value Lemma:

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

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.

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

Proof:

$$\begin{aligned} v(f) &= \sum_{\text{e out of A}} f(e) - \sum_{\text{e into A}} f(e) \\ v(f) &\geq \sum_{\text{e out of A}} d(e) - \sum_{\text{e into A}} d(e) \\ v(f) &\geq \sum_{\text{e out of A}} d(e) \\ v(f) &\geq dem(A,B). \end{aligned}$$

First and second line comes due to the fact that $d(e) \leq f(e)$. Third line comes from $\sum_{\text{e into A}} d(e) \leq \sum_{\text{e into A}} f(e)$, 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.
Proof: 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.

4. (**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 1: *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. $\square$

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 $\geq k \geq |C|$. $\square$

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

5. (**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 $O(mn)$.

6. (**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 $i \rightarrow j$ 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: ($\Longrightarrow$) 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 $i \to j$ 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 $G^{'}$.

7. (**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!?

6 NP, NPC, Complexity Theory

1. (**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.

Note: 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.