- The instructions are the same as in Homework-0, 1, 2.

There are 6 questions for a total of 100 points.

---

1. (5 points) Solve the following recurrence relations. You may use the Master theorem wherever applicable.

   (a) $T(n) = 3T(n/3) + cn$, T(1) = c

   Solution: Rewriting as
   $$T(n) = 3T(n/3) + O(n)$$
   which then fits into the template for master theorem $T(n) = aT(n/b) + O(n^d)$. Here, $a = 3$, $b = 3$, $d = 1$. Clearly, here $a = b^d$ which is the steady state condition. So the time complexity is then given by, $T(n) = O(n^d \log n) = O(n \log n)$, for this case.

   (b) $T(n) = 3T(n/3) + cn^2$, T(1) = c

   Solution: Rewriting as
   $$T(n) = 3T(n/3) + O(n^2)$$
   which then fits into the template for master theorem $T(n) = aT(n/b) + O(n^d)$. Here, $a = 3$, $b = 3$, $d = 2$. This time, here $a < b^d$ which is the top heavy condition. So the time complexity is then given by, $T(n) = O(n^d) = O(n^2)$, for this case.

   (c) $T(n) = 3T(n-1) + 1$, T(1) = 1

   Solution:

   Claim 1. $T(n) = \frac{3^n - 1}{2}$ *for all* $n \geq 1$

   Proof. We can prove this claim by induction as follows:

   Base Case When $n = 1$, $T(1) = 1 = \frac{3^1 - 1}{2}$ the claim holds.

   Inductive Case Suppose the claim holds for some $n - 1 \geq 1$. We will show that this implies that claim is also true for n.
   $$\begin{aligned} T(n) &= 3T(n-1) + 1 \\ &= 3 \times \frac{3^{n-1} - 1}{2} + 1 \\ &= \frac{3^n - 1}{2} \end{aligned}$$
   which established the induction and proves the claim. □

   Hence we can conclude that $T(n) \in O(3^n)$

2. (20 points) Consider the following problem: You are given a pointer to the root r of a binary tree, where each vertex v has pointers $v.lc$ and $v.rc$ to the left and right child, and a value $Val(v) > 0$. The value NIL represents a null pointer, showing that v has no child of that type. You wish to find the path from r to some leaf that minimizes the total values of vertices along that path. Give an algorithm to find the minimum sum of vertices along such a path along with a proof of correctness and runtime analysis.

Solution:

Assumption We have assumed that the root will not be a null, and if it is null the algorithm will throw error as no leaf can be reached from there. Let us define $MS(r)$, as a function of a node r which return the minimum total sum of values from r to leaf, in the tree formed by taking r as root.

Now, to find the $MS(r)$, we will make a claim,

Claim 1. *For tree with height k k >0, and whose root has one of the child as null the $MS(r)$ would be just the sum of value of root node and the MS of its non null child*

Proof. W.L.O.G let us assume that the left child is null, So to prove our claim we just have to show that $MS(r) = Val(v) + MS(r.rc)$,
Now as the left child is null therefore the root has to take the right child to to reach the leaf.
Let the path followed by $r.rc$ to get the $MS(r.rc)$ be $P = \{r.rc, \ldots, leaf\}$ Now consider the path $P' = \{r, r.rc, \ldots, leaf\}$ the leaf of right subtree formed by taking the $r.rc$ as a root will also be the leaf of the original tree, and therefore P' is also one of the path to reach the leaf. The sum along the path P' would be

$$\begin{aligned} Sum(P') &= Val(r) + Val(r.rc) + \ldots + Val(leaf) \\ Sum(P') &= Val(r) + (Val(r.rc) + \ldots + Val(leaf)) \\ Sum(P') &= Val(r) + Sum(P) \\ Sum(P') &= Val(r) + MS(r.rc) \\ MS(r) &\leq Val(r) + MS(r.rc) \end{aligned} \tag{1}$$

The last equality comes from the fact that the $MS(r)$ is the minimum sum achievable and therefore it would be less than any other path possible to the leaf.
Now let us consider the path followed by r to get the $MS(r)$ be $P'' = \{r, r.rc, \ldots, leaf'\}$. Note that the second node is $r.rc$ as the left child is null and it has to take the right child to reach the leaf, and also let $P''' = \{r.rc, \ldots, leaf'\}$

$$\begin{aligned} MS(r) &= Sum(P'') \\ MS(r) &= Val(r) + Val(r.rc) + ...Val(leaf') \\ MS(r) &= Val(r) + (Val(r.rc) + ...Val(leaf')) \\ MS(r) &= Val(r) + Sum(P''') \\ MS(r) &\geq Val(r) + MS(r.rc) \end{aligned} \tag{2}$$

The last equality comes from the fact that the $MS(r.rc)$ is the minimum sum achievable and therefore it would be less than any other path possible to the leaf from r.rc and hence less than $Sum(P''')$. From equation (1) and (2) we get $MS(r) = Val(r) + MS(r.rc)$ as required. Similar argument can be followed when the right child is null.
This proves our claim. □

Claim 2. *For tree with height k k >0, and both of the child of the roots are not null then the MS(r) would be just the sum of value of root node and the minimum of the MS of both child*

Proof. To prove this claim we just have to show that $MS(r) = Val(v) + \min(MS(r.lc), MS(r.rc))$. Let the path followed by r.rc to get the MS(r.rc) be $P = \{r.rc, \ldots, leaf\}$ Now consider the path $P' = \{r, r.rc, \ldots, leaf\}$ the leaf of right subtree formed by taking the $r.rc$ as a root will also be the leaf of the original tree, and therefore P' is also one of the path to reach the leaf. The sum along the path P' would be

$$\begin{aligned} Sum(P') &= Val(r) + Val(r.rc) + \ldots + Val(leaf) \\ Sum(P') &= Val(r) + (Val(r.rc) + \ldots + Val(leaf)) \\ Sum(P') &= Val(r) + Sum(P) \\ Sum(P') &= Val(r) + MS(r.rc) \\ MS(r) &\leq Val(r) + MS(r.rc) \end{aligned} \tag{3}$$

The last equality comes from the fact that the MS(r) is the minimum sum achievable and therefore it would be less than any other path possible to the leaf. Similarly we can show that

$$MS(r) \leq Val(r) + MS(r.lc) \tag{4}$$

Combining (3) and (4) we get

$$MS(r) \leq Val(r) + min(MS(r.lc), MS(r.rc)) \tag{5}$$

Now let us consider the path followed by r to get the $MS(r)$ be $P'' = \{r, x, ..., leaf'\}$. Note that the second node would be either $r.lc$ or $r.rc$, w.l.o.g let us take $x = r.rc$ also let $P''' = \{r.rc, \ldots, leaf'\}$

$$\begin{aligned} MS(r) &= Sum(P'') \\ MS(r) &= Val(r) + Val(r.rc) + ...Val(leaf') \\ MS(r) &= Val(r) + (Val(r.rc) + ...Val(leaf')) \\ MS(r) &= Val(r) + Sum(P''') \\ MS(r) &\geq Val(r) + MS(r.rc) \end{aligned} \tag{6}$$

The last equality comes from the fact that the $MS(r.rc)$ is the minimum sum achievable and therefore it would be less than any other path possible to the leaf from $r.rc$ and hence less than $Sum(P''')$. Similarly if x is equal to $r.lc$ we get

$$MS(r) \geq Val(r) + MS(r.lc) \tag{7}$$

Combining (6) and (7) we get

$$MS(r) \geq Val(r) + \min(MS(r.lc), MS(r.rc)) \tag{8}$$

From (5) and (8) we get

$$MS(r) = Val(r) + \min(MS(r.lc), MS(r.rc)) \tag{9}$$

and hence proves our claim. □

Now using this above proven claims we will propose a algorithm which takes as input a vertex and returns the minimum sum possible along any root to leaf path of the subtree with that vertex as its root.

```
1: function MINSUM(v)
2:     if v = NIL then
3:         throw Error
```

```
4:    end if
5:    if v.lc = NIL and v.rc = NIL then
6:        return Val(v)
7:    end if
8:    if v.lc = NIL then
9:        return Val(v)+Minsum(v.rc)
10:   end if
11:   if v.rc = NIL then
12:       return Val(v)+Minsum(v.lc)
13:   end if
14:   return Val(v) + min(Minsum(v.lc), Minsum(v.rc))
15: end function
```

Proof of Correctness Now we will show that the proposed algorithm is correcy, For this we will make a claim.
Claim: The function as proposed above Minsum(V) takes a input node v and returns MS(v). We will prove this by induction

Induction Hypothesis $P(n)$: $Minsum(v)$ returns MS(v) where the tree formed by taking the v as a root has height k,for all $0 \leq k \leq n$

Base Case When $n = 0$, v is a leaf. In this case as there is only one leaf in the tree so the path will be from root to this vertex, but as it also the root node therefore the sum would be just Val(v). In our function above as $v.lc$ and $v.rc$ both will be NIL and therefore will pass the second if condition and return the value of itself i.e. $Val(v)$

Induction Step Suppose $P(n-1)$ holds, now we will show that then $P(n)$ also holds.

There are two choice for the vertex v to go till a leaf. Either go left or right i.e either the leaf lies in the left subtree or in the right subtree. We will make cases on children of v.

1. **Case 1** Left Pointer is null
   As the height of tree ($n > 0$) and one of the pointer is null, the MS(v) would be equal to Val(v)+MS(v.rc) as shown in claim1 above.
   Now the height of right subtree is at max $n-1$ and as $P(n-1)$ holds therefore $Minsum(v.rc)$ will return MS(v.rc) and hence the MS(v) would be $Val(v) + Minsum(v.rc)$. This is exactly what is returned in the line 9 of the algorithm (the third if-condition)

2. **Case 2** Right Pointer is null
   As the height of tree ($n > 0$) and one of the pointer is null, the MS(v) would be equal to Val(v)+MS(v.lc) as shown in claim1 above.
   Following the similar argument as Case 1 the MS(v) in this case would be $Val(v)+Minsum(v.lc)$.
   This is exactly what is returned in the line 12(the fourth if-condition)

3. **Case 3** None of the child pointer is null
   As the height of tree ($n > 0$) and none of the pointer is null, the MS(v) would be equal to Val(v)+min(MS(v.lc),MS(v.rc)) as shown in claim2 above.
   Now the height of right subtree is at max $n-1$ and as $P(n-1)$ holds therefore $Minsum(v.rc)$ will return MS(v.rc), and similarly $Minsum(v.lc)$ will return MS(v.lc)

$$MS(v) = Val(v) + min(MS(v.lc), MS(v.rc))$$
$$\implies MS(v) = Val(v) + min(Minsum(v.lc), Minsum(v.rc))$$

This is exactly what is returned by the algorithm in line 14. Therefore $P(n)$ also holds.

Running Time Analysis Here n is the size of tree.

For $n \leq 1$ either of the first two if condition is fulfilled and O(1) time is taken to complete it.

For $n > 1$, none of the first two if condition is fulfilled, therefore the time taken to avoid those will be O(1) time.Now there are two case possible either one of the child is null or none of them is null. In both the case it will take T(L)+T(R)+O(1) time where L and R are the size of the left and right sub-tree respectively. So the recurrence relation formed will be

$$T(n) = \begin{cases} O(1) & n \leq 1 \\ T(L) + T(R) + O(1) & n > 1 \end{cases}$$

Claim 3. *$T(n) < cn$ for $n \geq 1$ for some $c >$ than hidden constant in O(1) terms.*

Proof. We shall prove the above claim using induction by using the claim as induction hypothesis.

Base Case $T(1) < c \cdot 1$ by choice of c.

Inductive Step Suppose for some $n > 1, T(k) < ck$ for all $1 \leq k < n$. Then,

$$\begin{aligned} T(n) &< cL + cR + c \quad \text{as } L, R < n \\ T(n) &< c(L + R + 1) \\ T(n) &< cn \end{aligned}$$

Hence the induction is established and claim is proved. □

Therefore algorithm runs in $O(n)$ time.

3. (20 points) Let S and T be sorted arrays each containing n elements. Design an algorithm to find the n^{th} smallest element out of the $2n$ elements in S and T . Discuss running time, and give proof of correctness.

Solution:

Introduction Let $C[1 \ldots 2n]$ denote the array obtained by merging 2 sorted input arrays A and B. The problem asks us to find $C[n]$. Trivially this will take $O(n)$ steps. But, we show that it is possible to find $C[n]$ without evaluating the array C using only $O(\log n)$ steps. The intuitive idea behind the algorithm for this question is as follows : We start by comparing the $\lfloor \frac{n}{2} \rfloor^{th}$ smallest elements of both the lists. Based on the result of comparison we reduce our search space in both the arrays by $\approx \frac{n}{2}$ elements and then recurse on the remaining lists each of size exactly $\lceil \frac{n}{2} \rceil$. We show later in the proof of correctness how such recursive strategy yields the nth smallest element. In the pseudocode that is sketched below we have clearly shown how the recursive calls are made to easily demonstrate the correctness, however, we note that a verbatim implementation of this pseudocode will be inefficient as it requires copying the subarrays for the recursive call. We show later in the running time analysis how we can use a pointer-based implementation to avoid copying

the subarrays and bring down the actual running time to $O(\log n)$. Another strategy is to imagine that we are not actually copying the subarrays but rather only constructing and passing around a light-weight "view" or "slice" of the original array in the backend (which can be implemented using a structure containing pointers to the begin and the end of the array in question).

```
1:  function NthElement(A[1...n], B[1...n])
2:      if n = 1 then
3:          return min (A[1], B[1])
4:      end if
5:      if A[⌊n/2⌋] ≥ B[⌊n/2⌋] then
6:          let A' ← A[1...⌈n/2⌉]
7:          let B' ← B[⌊n/2⌋ + 1...n]
8:      else
9:          let A' ← A[⌊n/2⌋ + 1...n]
10:         let B' ← B[1...⌈n/2⌉]
11:     end if
12:     return NthElement(A', B')
13: end function
```

Proof of Correctness Let $pos : \{A, B\} \times \{1 \dots n\} \mapsto \{1 \dots 2n\}$ denote a finite domain mapping function which determines the position to which an position in input array is mapped to in the merged array C. For example, $pos(A, 3)$ represents the index in C to which the element $A[3]$ gets mapped to. Note that pos is not a unique mapping if C contains duplicate elements. In such cases we are only interested in studying a particular mapping function. Now, we can observe

1. $pos(X, i) < pos(X, j)$ if $i < j$. Where X is A, B. If $X[i] < X[j]$ then the statement holds true because C is a sorted array by construction. If $X[i] = X[j]$ then we define pos to be such that the claim holds (simply by swapping).

Now, to show that the algorithm is correct we need to show that

Claim 1. *`FindNth` (A, B) returns the n^{th} smallest element for any 2 sorted arrays A, B of size n each, for all $n \geq 1$.*

We shall prove this claim by using proof by induction by proving the following inductive hypothesis.

Inductive Hypothesis $P(n)$: `FindNth` (A, B) returns the k^{th} smallest element for any 2 sorted arrays A, B of size k each, for all $1 \leq k \leq n$.

Base Case When $n = 1$, the 1^{st} smallest element is simply the minimum of the first element of both the arrays. Indeed, as we see from lines 2-3 of our algorithm the algorithm returns $\min(A[1], B[1])$. So for $n = 1$ the hypothesis holds.

Inductive Step Suppose we have shown that $P(n-1)$ holds for some $n > 1$. We will now show that this implies that $P(n)$ also holds. If A, B are of size $< n$ then the hypothesis holds by assumption of $P(n-1)$ so we will now assume that $|A| = |B| = n$. We will make cases on the values of $A[\lfloor \frac{n}{2} \rfloor]$ and $B[\lfloor \frac{n}{2} \rfloor]$ where $\lfloor . \rfloor$ denotes the floor function.

1. **Case 1** : $A[\lfloor \frac{n}{2} \rfloor] \geq B[\lfloor \frac{n}{2} \rfloor]$. First we can impose the constraint $pos(A, \lfloor \frac{n}{2} \rfloor) > pos(B, \lfloor \frac{n}{2} \rfloor)$ on our mapping function and then claim that

Claim 2. *$pos(B, i) < n$ for all $i \leq \lfloor \frac{n}{2} \rfloor$*

Proof. Since we already have $pos(B, i) < pos(B, j)$ when $i < j$ by definition of pos we only need to show that $pos(B, \lfloor \frac{n}{2} \rfloor) < n$ to prove the claim. To see this then observe that $B[\lfloor \frac{n}{2} \rfloor]$ is greater than $\lfloor \frac{n}{2} \rfloor - 1$ elements of B and atmost $\lfloor \frac{n}{2} \rfloor - 1$ elements of A, namely, $A[1 \dots \lfloor \frac{n}{2} \rfloor - 1]$. Therefore, in total, it comes after atmost $2\lfloor \frac{n}{2} \rfloor - 2$ elements in C. Therefore we have,

$$pos(B, \lfloor \frac{n}{2} \rfloor) \leq 2\lfloor \frac{n}{2} \rfloor - 1 < n \quad \text{as} \quad 2\lfloor \frac{n}{2} \rfloor - 1 = \begin{cases} n-1 & \text{n is even} \\ n-2 & \text{n is odd} \end{cases}$$

□

Claim 3. *$pos(A, i) > n$ for all $i \geq \lceil \frac{n}{2} \rceil + 1$*

Proof. Since we already have that $pos(A, i) > pos(A, j)$ if $i > j$ by definition of pos we only need to show that $pos(A, \lceil \frac{n}{2} \rceil + 1) > n$ to prove the claim. To see this then observe that $A[\lceil \frac{n}{2} \rceil + 1]$ is greater than $\lceil \frac{n}{2} \rceil$ elements of A and atleast $\lfloor \frac{n}{2} \rfloor$ elements of B, namely, $B[1 \dots \lfloor \frac{n}{2} \rfloor]$. Therefore, in total, we have that $A[\lceil \frac{n}{2} \rceil + 1]$ is greater than atleast $\lceil \frac{n}{2} \rceil + \lfloor \frac{n}{2} \rfloor = n$ elements of C. Hence $pos(A, \lceil \frac{n}{2} \rceil + 1) > n$. □

The above 2 claims prove that the n^{th} smallest element ($C[n]$) belongs to $(A' = A[1 \dots \lceil \frac{n}{2} \rceil]) \bigcup (B' = B[\lfloor \frac{n}{2} \rfloor + 1 \dots n])$. Note that both A', B' are of size $\lceil \frac{n}{2} \rceil$ each. Now we may claim that

Claim 4. *$C[n] = C'[\lceil \frac{n}{2} \rceil]$ where C' denotes the merge sorted array from A', B'.*

Proof. Note that $C[n] = C[\lfloor \frac{n}{2} \rfloor + \lceil \frac{n}{2} \rceil]$ and we have already discarded $\lfloor \frac{n}{2} \rfloor$ elements smaller than $C[n]$ from our search space, namely, $B[1 \dots \lfloor \frac{n}{2} \rfloor]$. Therefore, the $C[n]$ is now the $\lceil \frac{n}{2} \rceil^{th}$ smallest element in the remaining search space, i.e., C'. □

Now in our algorithm, for this case, we return the value `NthElement` (A', B'). As both A' and B' are of size $\lceil \frac{n}{2} \rceil$ and sorted, so by our assumption that $P(n-1)$ holds we can conclude that it will return the $\lceil \frac{n}{2} \rceil^{th}$ smallest element from A', B'. But as we have proved above $\lceil \frac{n}{2} \rceil^{th}$ smallest element of A', B' is actually the n^{th} smallest element of A, B therefore we have that our algorithm correctly returns the n^{th} smallest element of A, B. Hence for this case we have shown that our inductive hypothesis holds.

2. **Case 2** : $A[\lfloor \frac{n}{2} \rfloor] < B[\lfloor \frac{n}{2} \rfloor]$. We see that this case is exactly symmetric to the previous case with the roles of A, B swapped. Therefore we can apply the same argument for this case and by symmetricity conclude that the hypothesis also holds for this case as well.

So, we have shown that for both the mutually exclusive and exhaustive cases the $P(n)$ holds therefore the inductive step is complete and hence the induction is established. This completes the proof of correctness for our algorithm as we have shown that the algorithm correctly returns the n^{th} smallest element for all valid inputs.

Running Time Analysis First a word about subarrays A', B' in the algorithm. We note that although in the pseudocode these are represented as actual subarrays, in implementation we can represent these as "view" of original arrays instead. A "view" is essentially a pair of pointers, start pointer and end pointer, that characterizes a contiguous subarray completely. Representing a subarray as a "view" has the advantage that constructing A', B' takes only $O(1)$ time as we just need to create 2 new pointers and not copy the entire array. Therefore we only need to do $O(1)$ extra work per call to `FindNth` function in checking the if conditions (as memory access is $O(1)$) and

if required constructing these views. Now, we note that A', B' are of size $\lceil \frac{n}{2} \rceil$ each so the recursive call to `FindNth` involves recursing on problem instance of size $\lceil \frac{n}{2} \rceil$. So if we denote by $T(n)$ the time taken by algorithm when $|A| = |B| = n$ then we can write

$$T(n) = \begin{cases} O(1) & n = 1 \\ T(\lceil \frac{n}{2} \rceil) + O(1) & n > 1 \end{cases}$$

Now, we can write $\lceil \frac{n}{2} \rceil \leq c\frac{n}{2}$ for all $n > n_0 > 1$ where $1 < c < 2$ is constant whose exact value depends on chosen value of n_0 and in the limit $\lim_{n_0 \to \infty} c = 1$. For example, if we fix $n_0 = 3$, then we have $\lceil \frac{n}{2} \rceil \leq \frac{n}{2} + 1 \leq \frac{n}{2} + \frac{n}{4} \leq \frac{3n}{4}$. So, here $c = 3/2$. So we can write

$$T(n) \leq T(\frac{n}{2/c}) + O(1) \quad (\text{where } n > n_0 > 1)$$

which then fits into the template for master theorem $T(n) = aT(n/b) + O(n^d)$. Here, $a = 1$, $1 < b = 2/c < 2$, $d = 0$. Clearly, here $a = b^d$ which is the steady state condition. So the time complexity is then given by, $T(n) = O(n^d \log n) = O(\log n)$, for our case. Hence, the final runtime complexity of our algorithm is logarithmic in input size.

$$T(n) = O(\log n)$$

We shall further remark that while this is logarithmic in input size it is limited by the fact that if input of size n needs to be read then in practice it might not be better than its counter part with linear runtime. However, if the sequences A and B are mathematically representable such that the values $A[i], B[i]$ are mathematically computed rather than read from user than this algorithm will begin to outperform its linear counter parts by a large margin. Because this algorithm only requires $O(\log n)$ computations of $A[i]$ so if computing $A[i]$ requires $O(f(n))$ time in general then the algorithm will find the median in $O(f(n) \log n)$ whereas the simpler linear time algorithms will still take $O(nf(n))$.

4. An array $A[1...n]$ is said to have a majority element if more than half (i.e., $> n/2$) of its entries are the same. Given an array, the task is to design an efficient algorithm to tell whether the array has a majority element, and, if so, to find that element. The elements of the array are not necessarily from some ordered domain like the integers, and so there can be no comparisons of the form "is $A[i] \geq A[j]$?" (Think of the array elements as GIF files, say.) However you can answer questions of the form: "is $A[i] = A[j]$?" in constant time.

   (a) (10 points) Show how to solve this problem in $O(n \log n)$ time. Provide a runtime analysis and proof of correctness.

   (*Hint: Split the array A into two arrays A1 and A2 of half the size. Does knowing the majority elements of A1 and A2 help you figure out the majority element of A? If so, you can use a divide-and-conquer approach.*)

   Solution: In this part we will give approach which solves the problem in $O(nlogn)$ time. First we will split the given array into two small arrays A_1 and A_2 where A_1 is the left half of A of size $\lfloor \frac{n}{2} \rfloor$ and A_2 is right half of A of size $\lceil \frac{n}{2} \rceil$. Now we will make a claim and prove it that
   Claim 1. *If there is a majority element in array A then it has to be majority element of either A_1 or A_2*

   Proof. Let us assume that the claim is false and there exist a array A which has majority element m, while A_1 and A_2 formed from A either does not have majority element or if they

have then it is not m.
As m is the majority element let the frequency of m in array be f.
Now let as assume that the after the split A_1 and A_2 has f_1 and f_2 entries of m respectively.
As we have assumed that m is not the majority element of both A_1 and A_2 therfore

$$\frac{f_1}{\lfloor \frac{n}{2} \rfloor} \leq \frac{1}{2}$$

$$f_1 \leq \frac{\lfloor \frac{n}{2} \rfloor}{2} \tag{1}$$

Similarly

$$f_2 \leq \frac{\lceil \frac{n}{2} \rceil}{2} \tag{2}$$

Adding (1) and (2) gives

$$f_2 + f_1 \leq \frac{\lceil \frac{n}{2} \rceil}{2} + \frac{\lfloor \frac{n}{2} \rfloor}{2}$$

$$f_2 + f_1 \leq \frac{n}{2} \tag{3}$$

As the array is splitted therfore every element in A goes either to A_1 or A_2 therfore $f_1 + f_2 = f$ using this in (3)

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

This is a contradiction as to the fact that m is the majority element of A,(as if majority then $(f > n/2)$. Hence our assumption was wrong and this proves our proposed claim. □

Our algorithm `Majority_Element` uses this claim and a helper function `Check_Majority` to output the maximum element of present in the array in $O(nlogn)$ time.

```
1: function CHECK_MAJORITY(A[1...n], m)
2:     let num ← 0
3:     for i ∈ {1...n} do
4:         if A[i] = m then
5:             num ← num + 1
6:         end if
7:     end for
8:     if num > n/2 then
9:         return True
10:    else
11:        return False
12:    end if
13: end function
```

```
1: function MAJORITY_ELEMENT(A[1...n])
2:     if n = 1 then
3:         return (True, A[1])
4:     end if
5:     let A_1 ← A[1...⌊n/2⌋]
6:     let A_2 ← A[⌊n/2⌋ + 1...n]
7:     let (b_1, m_1) ← MAJORITY_ELEMENT(A_1)
8:     let (b_2, m_2) ← MAJORITY_ELEMENT(A_2)
9:     if b_1 = True then
10:        if CHECK_MAJORITY(A, m_1) then
```

```
11:             return (True, m_1)
12:         end if
13:     end if
14:     if b_2 = True then
15:         if CHECK_MAJORITY(A, m_2) then
16:             return (true, m_2)
17:         end if
18:     end if
19:     return (False, --)
20: end function
```

Proof of Correctness

First about the helper function `Check_Majority`, this function takes two input a array A and element m, and returns if it is the majority element or not in the array. The functionality of it is trivial it just calculates the frequency of element m by iterating over the whole array and if the frequency is greater than half of the size of array then it returns true otherwise false.

Claim 2. *Our main function* `Majority_Element`(A) *takes as input array A and returns a tuple of boolean value and an element. If the majority element exists then boolean value is true and the element returned is the majority element, otherwise the boolean value is false.*

We will prove this claim by induction.

Induction Hypothesis $P(n) : Majority\_element(A)$ takes a array of size k return a tuple of boolean value and a element. If the majority element exists then the boolean value is true and the element returned is the majority element, otherwise the boolean value is false, for all k such that $1 \leq k \leq n$

Base Step $P(1)$: This is trivial case as the element itself would be majority ($1 > 1/2$) and therefore majority element exist and is the first element, this is exactly what is returned by our function at line 3.

Induction Step Suppose we have shown that $P(n-1)$ holds for some $n > 1$ We will now show that this implies that $P(n)$ also holds. We will make cases on the existence of majority element in the array.

1. **Case 1** Majority element exists.
   As the size of both the array $A_1$and A_2 is less than n as ($\lfloor \frac{n}{2} \rfloor < n$ and $\lceil \frac{n}{2} \rceil < n$ for $n > 1$) therefore we can run the the *majoirty_element* function over the two arrays $A_1$and A_2 and they will work as stated above in the claim as $P(n-1)$ holds. Let the output of the algorithm for these 2 cases be (b_1, m_1) and (b_2, m_2) respectively. As shown in the very first claim that if there is majority element of A then it is either the majority element of A_1 or A_2 so at least one of b_1 or b_2 must be True. So we will check both the m_1 and m_2 (if they are majority of their array depending upon the value of b_1 and b_2) in the original array using our helper function and which one of them returns true then that is our majority element. (line 11 and line 16). Note that one of them will definitely pass the majority check. Here we also clarify that if m_1 comes out to be majority element then there is no need to check for m_2 as there cannot be two majority element of array.

2. **Case 2** Majority Element does not exist
   The first if condition will not be satisfied under this case as array of size 1 has a majority element. The next if condition will also not be satified either at outer if(line 9) or inner if(line 10). Line 10 will not be satisfied definitely as there is no majority element in the

array A and therefore no element m_1 can have frequency greater than half of the size of array and therefore the check majority function will always return false for any value of m in this case. Similarly next nested if conditions(line 14-18) will also be not satisfied and therefore our function will continue to line 20 where it will return a tuple with boolean value False.

Hence, we have verified the hypothesis for both the cases and therefore concluded that $P(n)$ holds which completes the inductive step of the argument and our hypothesis is established.

Running Time Analysis Check Majority has a for loop and some $O(1)$ steps. The for loop just iterate over the whole array A. So the time time complexity of Check majority function will be $O(n)$ time where n is the size of array.
Now for our main function majority_element, for $n = 1$ it runs in O(1) time. For $n > 1$ it takes some $O(1)$ steps in starting then recursively calls itself on two smaller arrays. In the end it also calls the helper function Check_majority two time which takes $O(n)$ time. So in total each call to this function does potentially $O(n)$ extra work. The time complexity can therefore be written as

$$T(n) = \begin{cases} O(1) & n = 1 \\ T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + O(n) & n > 1 \end{cases}$$

Claim 3. *$T(n) < cn \log_b n$ for $n \geq 2$ for some $c >$ than hidden constant in O(1) terms and where $b = \sqrt{2}$.*

Proof. We shall prove the above claim using induction by using the claim as induction hypothesis.

Base Case $T(2) = 2O(1) + O(2) = 4O(1) < c \cdot (2 \log_b 2)$ by choice of c.
$T(3) = O(1) + O(2) + O(3) = 6O(1) < c \cdot (3 \log_b 3)$ by choice of c.

Inductive Step Suppose for some $n > 3, T(k) < c \cdot k log_b k$ for all $1 \leq k < n$. Then,

$$T(n) < c \cdot \lceil \frac{n}{2} \rceil \log_b(\lceil \frac{n}{2} \rceil) + c \cdot \lfloor \frac{n}{2} \rfloor log_b(\lfloor \frac{n}{2} \rfloor) + c \cdot n \quad (\text{as } \lceil \frac{n}{2} \rceil, \lfloor \frac{n}{2} \rfloor < n)$$
$$T(n) < c \cdot \lceil \frac{n}{2} \rceil \log_b(\lceil \frac{n}{2} \rceil) + c \cdot \lfloor \frac{n}{2} \rfloor \log_b(\lceil \frac{n}{2} \rceil) + c \cdot n$$
$$T(n) < c \cdot n \log_b(\lceil \frac{n}{2} \rceil) + c \cdot n \tag{4}$$

We can observe for $b = \sqrt{2}$ and $n > 3$,

$$b \cdot \lceil \frac{n}{2} \rceil \leq n$$

where equality holds when b=2 and n is even. Putting this in eq(4) we get,

$$T(n) < c \cdot n \log_b(\frac{n}{b}) + c \cdot n$$
$$T(n) < c \cdot n \log_b(n) - c \cdot n \log_b(b) + c \cdot n$$
$$T(n) < c \cdot n \log_b(n) - c \cdot n + c \cdot n$$
$$T(n) < c \cdot n \log_b(n)$$

Hence the induction is established and claim is proved.
Using the above claim, we showed that the time complexity of our function Majority_element is $O(nlogn)$. □

(b) (10 points) Design a linear time algorithm. Provide a runtime analysis and proof of correctness. (*Hint: Here is another divide-and-conquer approach:*

- *Pair up the elements of A arbitrarily, to get $n/2$ pairs (say n is even)*
- *Look at each pair: if the two elements are different, discard both of them; if they are the same, keep just one of them*
- *Show that after this procedure there are at most $n/2$ elements left, and that if A has a majority element then it's a majority in the remaining set as well)*

Solution: In this part we will give approach that runs in $O(n)$ time. There are two cases possible for n.

Case 1 n is even.
Pair all the elements at odd index with consecutive next element, and create a empty array A_1, now look at each pair if they are same then insert one of them in A_1 and if different discard both of them. Now we will make a claim
Claim 1. *If there is a majority element in A then it is also the majority element of A_1.*

Proof. Let us assume that our claim is false and there exist A such that it has a majority element, say m and A_1 formed from A either does not have majority element and if it has then it is not m. Now, As m is the majority element let the frequency of m in array be f. As f is greater than $n/2$ we can also write it as.

$$f = \frac{n}{2} + 1 + p \quad (\text{where } p \geq 0) \tag{1}$$

Now let us assume that $2k$ elements having value m, got paired among themselves with $0 \leq 2k \leq f$ let them be P_1. Now in the remaining $n/2 - k$ pairs, $f - 2k$ pairs will be discarded as the pair have one element with value m and other with diff value let them be P_2. So the remaining pairs whose both element are not equal to m will be $n/2 - k - (f - 2k)$ which on simplifying gives $n/2 - f + k$ pairs let them be P_3 .
Now the array A_1 will have elements only from P_1 and P_3. All the pair P_1 will contribute as all the pairs have element with value m as both the element. Therefore the frequency of element m will be exactly k in the array A_1. Now as the array A_1 is formed from only P_1 and P_3. Therefore

$$|A_1| \leq |P_1| + |P_3|$$

where equality holds when all of the pairs of P_3 have second element same as the first element in that particular pair.

$$|A_1| \leq k + (\frac{n}{2} - f + k)$$

Putting the value of f from equation(1)

$$|A_1| \leq k + (\frac{n}{2} - (\frac{n}{2} + 1 + p) + k)$$

$$|A_1| \leq 2k - 1 - p \tag{2}$$

Now as we have assumed that m is not the majority element of A_1 then this implies

$$k \leq \frac{|A_1|}{2}$$

Now putting the value of A_1 from eq(2) we get

$$\begin{aligned} k &\leq \frac{2k-1-p}{2} \\ 2k &\leq 2k-1-p \\ 1+p &\leq 0 \end{aligned}$$

which is a contradiction as p≥0, and therefore $(1+p) > 0$. Hence our assumption was wrong and this proves our claim. □

Claim 2. *Size of A_1 is at max n/2.*

Proof. This can be proved trivially. As at most n/2 pairs are formed and each pair will either contribute 1 or 0 element in A_1 therefore at max n/2 elements can be there in A_1 □

Case2 When n is odd.
Create A' a sub-array of A from element at index 2 to end. Pair all the elements at odd index of A' with their consecutive next element, and create a empty array A_1, now look at each pair if they are same then insert one of them in A_1 and if different discard both of them. Now we will make a claim
Claim 3. *If there is a majority element in A then it is either the first element or the majority element of A_1.*

Proof. Let us assume that the claim is wrong, and there exist a array A with majority element m, where neither the first element is m, nor the majority element of A_1, if exists, is m.
Then, $A' = A[2 \dots n]$ now A' is of even size, and A_1 can be considered to be formed by pairing and discarding as in the case 1. Now as the first element is not m therefore let the frequency of m in A is equal to f, and as m is the majority element of A. Therefore

$$\begin{aligned} & f > n/2 \\ \implies & f > (n-1)/2 \end{aligned}$$

and hence m is also the majority element of A', and using the claim proved in case 1 we can argue that then m will also be the majority element of A_1 but this is a contradiction. This proves our claim □

Claim 4. *Size of A_1 is at max n/2.*

Proof. This can be proved trivially. As at most $(n-1)/2$ pairs are formed and each pair will either contribute 1 or 0 element in A_1 therefore at max (n-1)/2 elements can be there in A_1 and hence $|A_1| < n/2$ □

Now we will propose a Majority_element2 algorithm which uses the above proved claims and a helper function Check_Majority and runs in $O(n)$ time

```
1: function CHECK_MAJORITY(A[1...n], m)
2:     let num ← 0
3:     for i ∈ {1...n} do
4:         if A[i] = m then
5:             num ← num + 1
6:         end if
7:     end for
```

```
    if num > n/2 then
        return True
    else
        return False
    end if
end function
function MAJORITY_ELEMENT2(A)
    let n ← |A|
    if n = 1 then
        return (True, A[1])
    end if
    let A' ← []
    if n%2 = 1 then
        if CHECK_MAJORITY(A, A[1]) then
            return (True, A[1])
        end if
        A' = A[2, ...n]
    else
        A' = A[1, ...n]
    end if
    let n' ← |A'|
    let A_1 ← empty list
    let i ← 1
    while i < n' do
        if A'[i] == A'[i + 1] then
            A_1.append(A'[i])
        end if
        i ← i + 2
    end while
    let (b_1, m_1) ← MAJORITY_ELEMENT2(A_1)
    if b_1 then
        if CHECK_MAJORITY(A, m_1) then
            return (True, m_1)
        end if
    end if
    return (False, --)
end function
```

Proof of Correctness First about the helper function `Check_Majority`, this function takes two input a array A and element m, and returns if it is the majority element or not in the array. The functionality of it is trivial it just calculates the frequency of element m by iterating over the whole array and if the frequency is greater than half of the size of array then it returns true otherwise false.

Claim 5. *Our main function* `Majority_Element2`(A) *takes as input array A and returns a tuple of boolean value and an element. If the majority element exists then boolean value is true and the element returned is the majority element, otherwise the boolean value is false.*

We will prove this claim by induction.

Induction Hypothesis $P(n) : Majority\_element2(A)$ takes a array of size k return a tuple of boolean value and a element. If the majority element exists then the boolean value is true

and the element returned is the majority element, otherwise the boolean value is false, for all k such that $1 \leq k \leq n$

Base Step $P(1)$: This is trivial case as the element itself would be majority ($1 > 1/2$) and therefore majority element exist and is the first element, this is exactly what is returned by our function at line 4.

Induction Step Suppose we have shown that $P(n-1)$ holds for some $n > 1$. We will now show that this implies that $P(n)$ also holds. There are two cases possible

1. **Case 1** When majority element exists:
   There are further two subcases of this case
   (a) **Case 1.1** When n is even
   As n is even, A' is similar to A (line 13). All the pairing ,selecting and discarding is done from (line 15-23) and A_1 is formed, Now as shown in the above claim (Claim 1) that if there is a majority element of A then it is also the majority element of A_1. As the size of A_1 is less than $n/2$ (Claim 2) and $P(n-1)$ holds the majority element of A_1 can be found by using Majority_element2 and we have to just return it. This will be exactly done in our algorithm as both the if condition will be satisfied and the tuple will be returned in line 27.
   (b) **Case 1.2** When n is odd
   As n is even, therefore first the first element is checked if it is majority element or not and if yes then it is returned (line 8-10). Now if this is not the case then the A' is formed by taking the sub-array starting from element at index 2 till the end. All the pairing ,selecting and discarding is done from (line 15-23) and A_1 is formed.
   Now as shown in the above claim (Claim 3) that if there is a majority element of A then it is either the first element or the majority element of A_1. First one was not case so it is surely that the majority element is the majority element of A_1. As the size of A_1 is less than $n/2$ (Claim 4) and $P(n-1)$ holds the majority element of A_1 can be found by using Majority_element2 and we have to just return it. This will be exactly done in our algorithm as both the if condition will be satisfied and the tuple will be returned in line 27.
2. **Case 2** When majority element does not exists:
   As the majority element does not exist therefore $n > 1$. Now we can see that before both the return condition(line 9 and line 27) there is a if condition of check majority. And as the majority element does not exist then no element can have frequency more than half and therefore it will always return false. So our algorithm will not end in between and will come to line 30 and here it will return false, exactly as it should be as no majority element exists.

Running Time Analysis
Check Majority has a for loop and some $O(1)$ steps. The for loop just iterate over the whole array A. So the time time complexity of Check majority function will be $O(n)$ time where n is the size of array.
Now for our main function majority_element, for $n = 1$ it runs in $O(1)$ time. for $n > 1$ there are two case:
Case 1 When n is odd: It completes line 1-6 in $O(1)$ time, Then the line $7-14$ will take $O(n)$ time to call the helper function once and copying the array in A'. Line 15-17 will take $O(1)$ time. While loop will be completed in $O(n)$ time as it iterates $n/2$ time and in each loop it

takes $O(1)$ time. Line 24 will take T($\lfloor n/2 \rfloor$) time. line 25-30 will take $O(n)$ time.
Case 2 When n is even, It completes line 1-6 in $O(1)$ time Then the line 7-14 will take $O(n)$ time in copying the array in A'. Line 15-17 will take $O(1)$ time. While loop will be completed in $O(1)$ time as it iterates $n/2$ time and in each loop it takes $O(1)$ time. Line 24 will take T($\lfloor n/2 \rfloor$) time. line 25-30 will take $O(n)$ time
So the time complexity can be written as

$$T(n) = \begin{cases} O(1) & n = 1 \\ T(\lfloor \frac{n}{2} \rfloor) + O(n) & n > 1 \end{cases}$$

Now as $\lfloor \frac{n}{2} \rfloor \leq \frac{n}{2}$ we can write

$$T(n) \leq T(\frac{n}{2}) + O(n) \quad \text{for } n > 1$$

Now this fits well in our template of master theorem with $T(n) = aT(n/b) + O(n^d)$. Here a=1,b=2 and d=1. Clearly we have $a < b^d$ which is top heavy situation. So the time complexity will be given by$T(n) = O(n^d) = O(n)$ for our case. Hence the final runtime complexity is linear for the input size.

5. Consider the following algorithm for deciding whether an undirected graph has a *Hamiltonian Path* from x to y, i.e., a simple path in the graph from x to y going through all the nodes in G exactly once. ($N(x)$ is the set of neighbors of x, i.e. nodes directly connected to x in G).

```
HamPath(graph G, node x, node y)
  - If x = y is the only node in G return True.
  - If no node in G is connected to x, return False.
  - For each z ∈ N(x) do:
      - If HamPath(G − {x}, z, y), return True.
  - return False
```

(a) (7 points) Give proof of correctness of the above backtracking algorithm.

Solution:

Notation For this problem we shall use $N(v)$ to denote the set of neighbour vertices of vertex $v \in G$. Further we write $G - v$ to denote the induced subgraph of G formed after removing v and its incident edges. We use $|G|$ to denote the number of vertices in the graph $|G| = |V(G)|$. Also, we shall define the predicate `ExistsHamiltonian`(G, u, v) to denote the predicate : there exists a hamiltonian path between vertices u and v in G.

Lemma 1. *For any graph G and vertices (not necessarily distinct) u and v :* *`ExistsHamiltonian`(G, u, v) if and only if there exists some $x \in N(u)$ such that* *`ExistsHamiltonian`$(G - u, x, v)$.*

Proof. To prove the if and only if statement we will prove the implication from both the sides.

($\implies$) `ExistsHamiltonian`(G, u, v) implies there exists some $x \in N(u)$ such that `ExistsHamiltonian`$(G - u, x, v)$

Let $P = (u, x_1, x_2, \ldots, x_k, v)$ denote the hamiltonian path that exists between u, v. Then we see that $x_1 \in N(u)$. So, we consider the path $P' = (x_1, x_2, \ldots, x_k, v)$. We claim that P' is a valid path on the graph $G - u$ as well. To see this note that $G - u$ is formed by removing the vertex u and its incident edges from G. Secondly, by definition of hamiltonian path it visits the vertices of G only once so this means that $u \notin \{x_1, x_2, \ldots, x_k\}$ and therefore we have that the path P' does not pass through u and is therefore a valid path in $G - u$. Note that P' is a hamiltonian path between x and v in $G - u$ by construction as it passes through all the vertices of $G - u$ as P was a hamiltonian path.

($\Longleftarrow$) There exists some $x \in N(u)$ such that `ExistsHamiltonian`$(G - u, x, v)$ implies `ExistsHamiltonian`(G, u, v)
Let $P' = (x, x_1, x_2, \ldots, x_k, v)$ denote the hamiltonian path between x, v in $G - u$. Note that P' is also a valid path in G as $G - u$ is but a subgraph of G. Now, consider $P = u \bigcup P' = (u, x, x_1, x_2, \ldots, x_k, v)$. We claim that P is then a hamiltonian path in G between u, v. To see this we note that since P' was a hamiltonian path between x and v in $G - u$ it passes through all the vertices in $G - u$ exactly once. We augmented P' with insertion of vertex u only so we have that P also passes through all the vertices of $G - u$ plus the vertex u exactly once. Since $G - u \bigcup u = G$ we have that P is a hamiltonian path in G.

□

Now to prove the correctness of the algorithm we need to show that

Claim 1. *`HamPath` returns the correct answer for any (finite) graph G and vertices $x, y \in G$. That is, `HamPath` (G, x, y) = `ExistsHamiltonian` (G, x, y)*
To prove this claim we will use the technique of proof by induction by establishing the following inductive hypothesis.

Inductive Hypothesis $P(n)$: `HamPath` returns the correct answer for any graph G and vertices $x, y \in G$ if $|G| \leq n$.

Base Case When $n = 1$ we necessarily have that $x = y$ and the trivial path $P = \{x = y\}$ is the hamiltonian path for this G. Therefore there always exists the hamiltonian path for this case. Indeed, the algorithm returns `true` for this case (line 1).

Inductive Case Suppose that we have shown that $P(n-1)$ holds true. We will now show that this implies that $P(n)$ also holds.
So, let G be any graph such that $|G| = n$ and $x, y \in G$ be any 2 vertices. We will now make 2 cases here

1. **Case 1 :** $N(x) = \Phi$. In this case no hamiltonian path between x, y can exist in G where $|G| \geq 2$. This is because x has no outgoing edges so any path starting x will terminate at x itself and therefore will not be able to span all the vertices of G. Note that our algorithm `HamPath` also returns False in this case as stated in line 2 of the algorithm. So the hypothesis holds for this case.
2. **Case 2 :** $N(x) \neq \Phi$. From the lemma that we proved above we know that `ExistsHamiltonian` (G, x, y) if and only if `ExistsHamiltonian` $(G - x, z, y)$ for some $z \in N(x)$. Now in our algorithm the loop in the lines 3-4 checks for each $z \in N(x)$ the value of `HamPath` $(G - x, z, y)$. If $|G| = n$ then $|G - x| = n - 1$ therefore by our assumption that $P(n-1)$ holds we have that `HamPath` $(G - x, z, y)$ = `ExistsHamiltonian` $(G - x, z, y)$

for all $z \in N(x)$. Then by the lemma we now have that algorithm correctly computes the value of `HamPath` (G, x, y) for this case also.

Hence for both the cases we have shown that $P(n)$ holds and therefore the inductive step is complete and the hypothesis is established. This completes our proof of correctness.

(b) (8 points) If every node of the graph G has degree (number of neighbors) at most 4, how long will this algorithm take at most? *(Hint: you can get a tighter bound than the most obvious one.)*

Solution: We will start by showing a weaker $O(4^n)$ bound and then progressively refine it to $O(3^n)$ and then to $O(3^{2n/3})$ which is the best we have been able to achieve. We can not show that $O(3^{2n/3})$ is the tightest bound but we can say that it is not *very* far away either as we produce a graph which has at least $O(20^{n/5})$ paths.

Claim 1. $T(n) \in O(4^n)$

Proof. To prove this we only need to observe that at any recursive call the algorithm recursively branches to all the neighbours of that vertex one by one. A vertex can have potentially 4 neighbours so during each recursive call the branching factor is at most 4. The recursion could be as much as n calls deep for a path of length n starting from vertex x so therefore the time taken by this algorithm $T(n)$ is bounded by $O(4^n)$ at the least. More formally, in each call the program only does $O(1)$ extra work checking if it is the only remaining vertex, or the goal. Then it enters the loop which can run at most 4 times. So,

$$T(n) \leq 4T(n-1) + O(1)$$

We can solve this recurrence using method similar to Q1 c) and obtain $T(n) \in O(4^n)$. □

Claim 2. $T(n) \in O(3^n)$

Proof. Actually we can observe that only the very first call to this algorithm can branch off to at most 4 neighbours, all the other remaining calls can only branch off to at most 3 neighbours only as one of their neighbour must have already been traversed before the recursive call to this vertex. So we can refine our recurrence,

$$T(n) \leq \begin{cases} 4T(n-1) + O(1) & n = |V(G)| \\ 3T(n-1) + O(1) & n < |V(G)| \end{cases}$$

Solving which will yield $O(4 \times 3^{n-1}) = O(3^n)$. So therefore $T(n) \in O(3^n)$ at least. □

Claim 3. $T(n) \in O(3^{2n/3})$

Proof. To refine the bound further we will consider the recursion tree of the algorithm. As discussed earlier at each call the algorithm does only $O(1)$ work (checking termination and deleting the node in a graph with bounded degree) so run time of this algorithm will be $O(f(n))$ if $f(n)$ is the number of nodes in the recursion tree. So we will now try to bound the number of nodes in the recursion tree of this graph to obtain a bound on run time as a consequence. Now, we have already shown in the previous claim how all nodes except the first will have only 1,2 or 3 incident edges when the recursive call is made on them, and notice that whenever we make a recursive call we delete all the incident edges of that vertex so at each step in the recursion we delete 1, 2 or 3 edges from the graph. Further, deleting 2 or 3 incident edges from a vertex means the vertex had that many neighbours respectively. So, moving along any path in the recursion tree we can estimate the branching caused by a node based on number of incident edges deleted – if we deleted k incident edges on that vertex then the corresponding node in recursion tree will cause branching of k and vice versa. Now we will prove a lemma regarding trees.

Lemma 2. *Let T be a tree rooted at r. Denote by b_v the branching at node v. Then there exists a path $P^* = (r, v_1, v_2, \ldots, v_k)$ from r to some leaf v_k in T such that if we define $n_P^* = 1 + b_r + b_r b_{v_1} + \ldots + (b_r b_{v_1} \ldots b_{v_k})$, then, $n_T \leq n_P^*$ where n_T denotes the number of nodes in T and h denotes height of T.*

Proof. We will prove this lemma by induction on height of tree h.

Base Case. When $h = 0$, then number of nodes is 1 so claim holds.

Inductive Case. Suppose the lemma is proved for upto $h-1$ for some $h > 0$. Then we will show that this implies that it also holds for height h.

$$n_T = n_r = 1 + \sum_{v \in N(r)} n_v$$
$$n_r \leq 1 + b_r \max_{v \in N(r)} n_v$$

Let $v_1 \in N(r)$ be the node for which n_{v_1} is maximum. Then the subtree rooted at v_1 will be of height atmost $h-1$. So by induction we can write

$$n_r \leq 1 + b_r(1 + b_{v_1} + \ldots + (b_{v_1} \ldots b_{v_k}))$$
$$n_r \leq 1 + b_r + b_r b_{v_1} + \ldots + (b_r b_{v_1} \ldots b_{v_k})$$

So therefore, $P^* = (r, v_1, v_2, \ldots, v_k)$ is the desired path. □

Now we state the following lemma which shall be useful for finding a bound:

Lemma 3. *If $(b_1, b_2, \ldots, b_n)$ is a sequence of positive integers, and $(a_1, a_2, \ldots, a_n)$ is the same sequence sorted in non-increasing order, we have $1 + b_1 + b_1 b_2 + \cdots + (b_1 b_2 \cdots b_k) \leq 1 + a_1 + a_1 a_2 + \cdots + (a_1 a_2 \cdots a_k)$.*

Proof. Consider any prefix products $a_1 a_2 \cdots a_i$ and $b_1 b_2 \cdot b_i$. Let the (non-increasing) sorted order of $b_1, \ldots, b_i$ be $b'_1, \ldots, b'_i$. Then we claim that $a_i \geq b'_i$. Suppose this wasn't the case, and hence there is an a_i such that $a_i < b'_i$. Then by the non-increasing property of the sequence b', we have $a_i < b'_j$ for all $j \leq i$. Hence there are at least i elements in a which are more than a_i, which is a contradiction since a_i by definition has at most $i-1$ elements more than it. Thus for each i, we have $a_i \geq b'_i$. Taking the product and noting that all a_i, b'_is are positive, and noting that $b'_1, \ldots, b'_i$ is merely a permutation of $b_1, \ldots, b_i$, we get that $a_1 a_2 \cdots a_i \geq b_1 b_2 \cdot b_i$. Taking the sum over all indices i from 1 to k, the lemma follows. □

Now denote by $\mathscr{P}$ the set of all paths in the recursion tree. And let $p \in \mathscr{P}$ be any one such path. If moving along the path p, discounting the very first vertex x, we encounter n_1 vertices with branching 1, n_2 vertices with branching 2 and n_3 vertices with branching 3, then we may conclude

1. $n_p \leq 1 + 3 + \ldots + 3^{n_3} + 3^{n_3} 2 + \ldots + 3^{n_3} 2^{n_2 - 1} + 3^{n_3} 2^{n_2} n_1$ by combining the results of lemmas 2 and 3.
2. We deleted a total of $n_1 + 2n_2 + 3n_3$ edges from the graph.

However, we know that in any graph G, $2|E| \leq \sum_{v \in G} d(v)$ which implies that $2|E| \leq 4n$ in our case as we max degree is 4, so $|E| \leq 2n$. And since we were able to delete $n_1 + 2n_2 + 3n_3$ edges, this implies that $n_1 + 2n_2 + 3n_3 \leq |E| \leq 2n$.

$$\begin{aligned}
f(n) &\leq \max_{p \in \mathscr{P}} n_p \\
&\leq \max_{p \in \mathscr{P}} 1 + 3 + \ldots + 3^{n_3} + 3^{n_3}2 + \ldots + 3^{n_3}2^{n_2-1} + 3^{n_3}2^{n_2}n_1 \\
&= \max_{p \in \mathscr{P}} \frac{3^{n_3}-1}{2} + 3^{n_3}(2^{n_2}-1) + 3^{n_3}2^{n_2}n_1 \\
&= \max_{p \in \mathscr{P}} (n_1+1)2^{n_2}3^{n_3} - \frac{3^{n_3}+1}{2} \\
&= \max_{p \in \mathscr{P}} 3^{n_3}((n_1+1)2^{n_2} - 1/2) - 1/2 \\
&\text{subject to } n_1 + 2n_2 + 3n_3 \leq 2n
\end{aligned}$$

So this implies that we need to solve the following optimization problem.

$$\begin{aligned}
f(n) &\leq 3^{n_3^*}((n_1^*+1)2^{n_2^*} - 1/2) - 1/2 \\
\text{where } n_1^*, n_2^*, n_3^* &= \arg\max_{n1,n2,n3} 3^{n_3}((n_1+1)2^{n_2} - 1/2) - 1/2 \\
&= \arg\max_{n1,n2,n3} n_3 \log 3 + \log((n_1+1)2^{n_2} - 1/2) \\
&\text{subject to } n_1 + 2n_2 + 3n_3 \leq 2n, n_i \geq 0 \quad \forall i
\end{aligned}$$

For this, we shall need the following claim:

Claim 4. *In an optimal real solution (n_1, n_2, n_3), we have $n_1 < d = 8$.*

Proof. Suppose this is not the case, and that $n_1' \geq 1$. Consider the solution $(n_1', n_2', n_3') = (n_1 - d, n_2, n_3 + d/3)$. This also satisfies the conditions.

Let $g(n_1, n_2, n_3) = 3^{n_3}((n_1+1)2^{n_2} - \frac{1}{2})$.

Then we have

$$\begin{aligned}
\frac{g(n_1', n_2', n_3')}{g(n_1, n_2, n_3)} &= \frac{3^{d/3}(n_1 + 1 - d - 1/2^{n_2+1})}{n_1 + 1 - 1/2^{n_2+1}} \\
&\geq \frac{3^{d/3}(n_1 - d + \frac{1}{2})}{n_1 + \frac{1}{2}} \\
&\geq \frac{3^{d/3}}{2d+1} \\
&> 1
\end{aligned}$$

where the last inequality follows from the fact that $3^{d/3} > 2d + 1$ for $d = 8$. □

Hence we can check for each possible value of n_1^* in $[0, 8)$. In each case, we need to optimize the function $3^{n_3}(a2^{n_2} - 1/2)$.

We shall need the following claim:

Claim 5. *In any optimal real solution (n_1, n_2, n_3), we have $n_2 < D = 11$.*

Proof. Suppose there is an optimal real solution with $n_2 \geq D$. Suppose the value of n_1 is some $a-1$ (for the sake of convenience).

Let $G(n_2, n_3) = 3^{n_3}(a2^{n_2} - 1/2)$.

Consider the tuple $(n_2', n_3') = (n_2 - D, n_3 + 2D/3)$.

We then have

$$\begin{aligned}
\frac{G(n_2', n_3')}{G(n_2, n_3)} &= \frac{3^{2D/3}(a2^{n_2-D} + 1/2)}{a2^{n_2} + 1/2} \\
&= \frac{3^{2D/3}}{2^D} \cdot \frac{1 + \frac{1}{a2^{n_2-D+1}}}{1 + \frac{1}{a2^{n_2+1}}} \\
&\geq \frac{3^{2D/3}}{2^D} \cdot \frac{1}{3/2} \\
&> 1
\end{aligned}$$

Where the last inequality follows from the fact that for $D = 11$, the final ratio is > 1. □

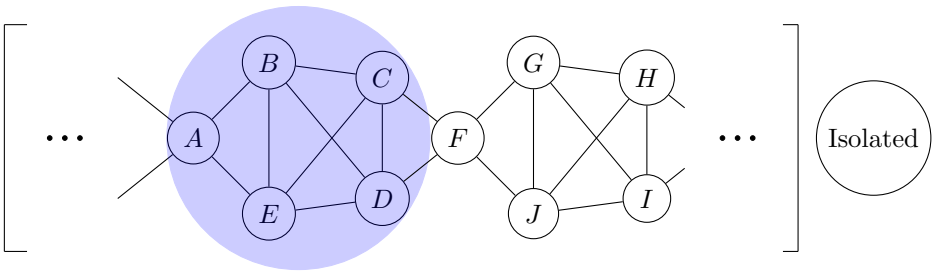
Hence, for any optimal solution, we must have $n_1^* = O(1), n_2^* = O(1)$ and $n_3^* = 2n/3 - O(1)$.

Hence, plugging in these values, we see that the upper bound on $f(n)$ is $O(3^{2n/3})$, and thus, we have

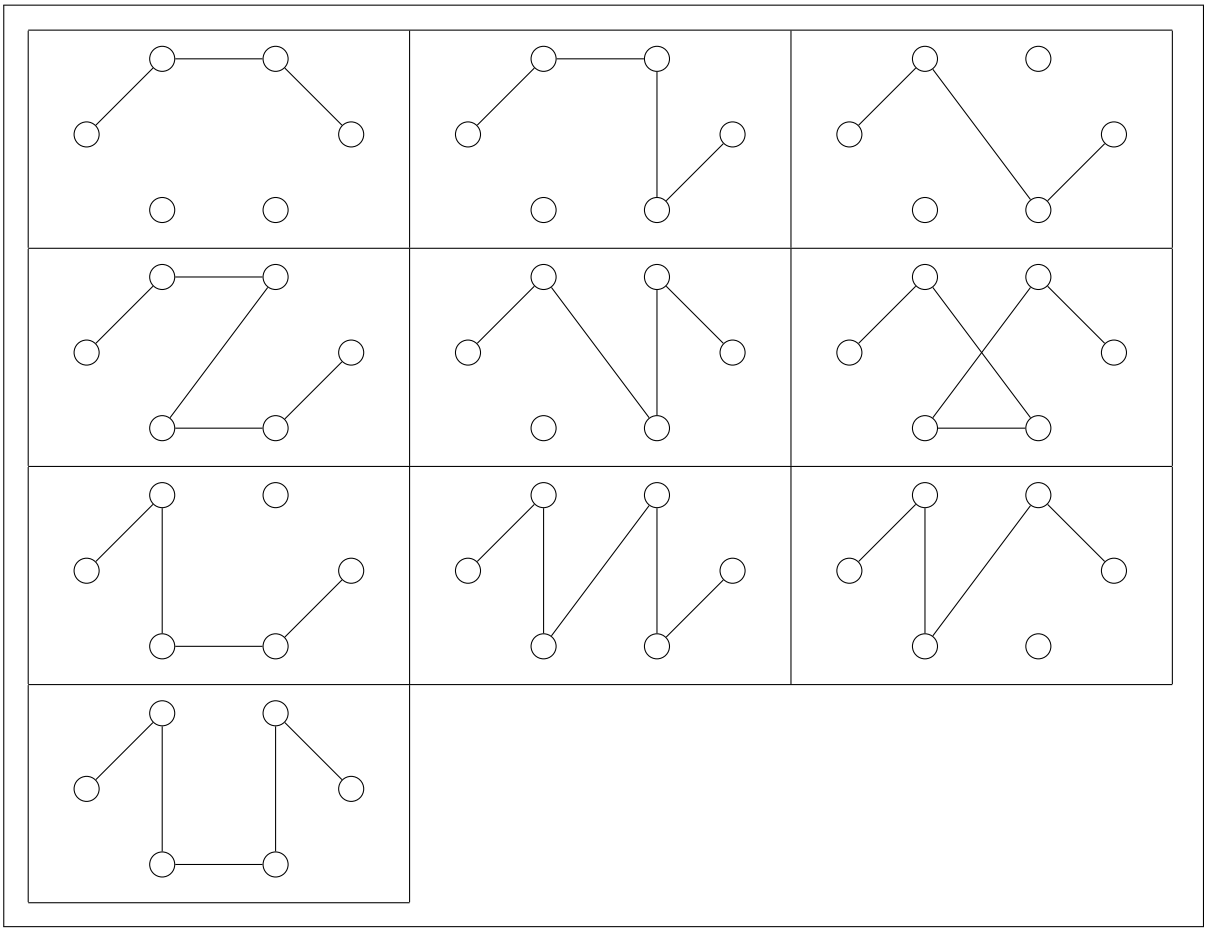
$$T(n) \in O(3^{2n/3}) \approx O(2.0800^n)$$

□

Finally we produce an example of graph with n vertices such that number of paths in this graph from a particular vertex to another particular vertex is $\Theta(1.8205^n)$. The graph is essentially a chain of pentagons arranged as shown, with an isolated rightmost vertex, so that the answer to the problem is no. In this case, note that all paths from the starting node to the last node will be traversed. Notice that each vertex has degree of at most 4 (exactly 4 for most vertices, but less for the vertices on the corner pentagons and the isolated vertex). To obtain an estimate on number of paths in this graph one can calculate that number of paths from the vertex marked A to vertex marked F is 20 going through the connecting pentagon. So there are 20 ways to reach F from A. So if there are n vertices, we have $(n-1)/5$ pentagons and number of ways to reach the last end of last pentagon from the tip of first pentagon will be $\Theta(20^{n/5}) = \Omega(1.8205^n)$.



The following are 10 paths (the remaining 10 are obtained by reflecting the paths over the horizontal axis).



6. (20 points) Design an $O(poly(n) \cdot 2^{n/4})$-time backtracking algorithm for the maximum independent set problem for graphs with bounded degree 3 (these are graphs where all vertices have degree at most 3). Give running time analysis and proof of correctness.

Solution:

Note that the following algorithm also works for graphs with vertices with degree more than 3, however the exhibited runtime bound matches with the requested runtime bound only for graphs with bounded degree 3.

We exhibit an algorithm that works in $O(n^2 2^{n'/4})$ time, where n' is defined as

$$\sum_{i \in \mathbb{N}} i \times \text{number of vertices with degree } (i+2)$$

Clearly for a graph with max degree 3, we have $n' \leq n$, so the algorithm has runtime in $O(n^2 2^{n/4})$.

Note: The following algorithm can be optimized in a real-world implementation using backtracking and updating the graph instead of creating a new graph each time, and maintaining a heap to find the vertex with the least degree. However, the complexity still remains the same and there is no real complexity gain.

Denote by $N(v)$ the neighbours of a vertex v for the sake of brevity. Edges are denoted as ordered pairs $\{u, v\}$ and not as ordered pairs (u, v) since the graph is undirected.

Algorithm

function MaximumIndependentSet(Graph $G = (V, E)$)
  if G is empty **then**
    return $\{\}$
  end if
  let v be the vertex with the least degree in G
  if $deg(v) = 0$ **then**
    return $\{v\} \cup$ MaximumIndependentSet$(G \setminus \{v\})$
  else if $deg(v) = 1$ **then**
    let u be the (only) neighbour of v
    return $\{v\} \cup$ MaximumIndependentSet$(G \setminus \{v, u\})$
  else if $deg(v) = 2$ **then**
    let u_1, u_2 be the (only) neighbours of v.
    if there is an edge between u_1 and u_2 in G **then**
      return $\{v\} \cup$ MaximumIndependentSet$(G \setminus \{v, u_1, u_2\})$
    else
      let v' be a new vertex not in G.
      let $E_o = \{\{u_1, u\} \mid u \in N(u_1)\} \cup \{\{u, u_2\} \mid u \in N(u_2)\}$ ▷ Edges incident to at least one vertex in the set $\{u_1, u_2\}$. Recall that we represent edges as unordered pairs.
      let $E_n = \{\{v', u\} \mid u \in N(u_1) \cup N(u_2) \setminus \{v\}\}$ ▷ Edges that will join v' to all neighbours of u_1, u_2 except v
      let $G' = ((V \cup \{v'\}) \setminus \{v, u_1, u_2\}, (E \cup E_n) \setminus E_o)$ ▷ Merge v, u_1, u_2 into 'super-node' v'
      let $S =$ MaximumIndependentSet(G').
      if v' is in S **then**
        return $\{u_1, u_2\} \cup S \setminus \{v'\}$
      else
        return $\{v\} \cup S$
      end if
    end if
  else ▷ In this case, $deg(v) \geq 3$
    let $S_1 = \{v\} \cup$ MaximumIndependentSet$(G \setminus (\{v\} \cup N(v)))$
    let $S_2 =$ MaximumIndependentSet$(G \setminus \{v\})$
    return the larger of S_1, S_2.
  end if
end function

Proof of Correctness We prove the correctness of the algorithm using induction on the number of vertices in the graph.

Inductive hypothesis: MaximumIndependentSet(G) returns a maximum independent set of G for all graphs G with number of vertices less than n for some $n > 0$.

Base Case: In the case the graph is empty, the maximum independent set of the graph is the empty set, and hence the algorithm returns the correct answer in the case that the graph is empty.

Inductive step:

Firstly we claim the following:

Claim 1. *For any graph G, and a vertex v in it, if there is a maximum independent set in G that*

contains exactly one vertex in $\{v\} \cup N(v)$, then there is a maximum independent set in G containing v.

Proof. Suppose that there exists a maximum independent set S in G that contains exactly one vertex out of the vertices in $\{v\} \cup N(v)$. If that vertex is v, we are trivially done. Now assume that S contains a vertex in $N(v)$, say u. Consider the set $S \setminus \{u\} \cup \{v\}$. Note that since there is no neighbour of v in this set (which is because u was the only neighbour of v in S), this set is also a valid independent set by the validity of S and this observation. Since the size of this set is the same as that of S, this set is also a maximum independent set in G, and we have shown the claim, as needed. □

Now, we make cases on the vertex v with the least degree in the graph.

1. $deg(v) = 0$.

   In this case, since this vertex is isolated, we claim that this vertex will always be in any maximum independent set of G. Indeed, suppose S is any maximum independent set of G that doesn't contain v. Then for any vertex u in S, since $deg(v) = 0$, u is not adjacent to any vertex in S. Hence we can add u to S to get a bigger independent set, which is a contradiction. By the inductive hypothesis, the recursive call gives the rest of the independent set, as needed.

2. $deg(v) = 1$.

   We claim that there is always at least one maximum independent set of G that contains v. Let u be the only neighbour of v. Consider any maximum independent set S of G.

   (a) S contains at least one of u, v.

   Note that it can't contain both of these vertices. Also, by the claim in the beginning, we have a maximum independent set that contains v.

   (b) S contains none of u, v.

   By a completely similar analysis, we can show that $S \cup \{v\}$ is also an independent set, which is larger than S, and it contradicts the fact that S is a maximum independent set of G.

   Hence, we have shown our claim, so it is always optimal to choose a degree 1 vertex. Since u can't be in such an independent set, the remaining part of the set must come from $G \setminus \{u, v\}$, and must be independent. Note that the maximum independent set of this subgraph is an independent set by definition, and taking the union of this with v yields an independent set of G. It is also the maximum independent set of G, since if it wasn't, then removing the vertex v from the maximum independent set of G which has v would yield a larger independent set of $G \setminus \{u, v\}$. By the induction hypothesis, the recursive call in the function corresponding to this case gives a maximum independent set of this case, and we are done in this case.

3. $deg(v) = 2$.

   Let u_1, u_2 be the two neighbours of v. In this case, we make two subcases:

   (a) There is an edge between u_1, u_2. In this case, we can choose exactly one of v, u_1, u_2 in the maximum independent set (if we choose none, the maximum property is violated by adding in v, and if we choose two or more, the set is no longer independent). By the claim, it is optimal to choose v, and the remaining set must come from $G \setminus \{v, u_1, u_2\}$. Using a completely analogous argument as in the degree 1 case, we can show that the union of the chosen vertex (which is v) and the answer of the recursive call is a maximum independent set of the graph.

(b) There is no edge between u_1, u_2. In this case, we can choose either one or two vertices from $\{v, u_1, u_2\}$ in the maximum independent set of G. We combine the nodes v, u_1, u_2 into a node v' to get a new graph G'. Note that the recursive call computes a maximum independent set of this new graph since it has two less vertices than G, call that set S. Our claim is the following:

Claim 2. *If S contains v', then it is optimal to choose u_1, u_2 in G, and if S doesn't contain v, then it is optimal to choose v in G.*

Proof. We first show how to construct an independent set of G from S which has exactly one more vertex than S. We make two cases:

i. S contains v'.

Note that since S contains v', no vertex adjacent to v' in G' is in S, and thus no neighbour of u_1, u_2 is in S. Hence we can add u_1, u_2 after removing v' to get an independent set of G with exactly one more vertex than S has.

ii. S doesn't contain v'.

In this case, we can add v to S, and this is an independent set because no neighbour of v is in S. Since we have added one vertex to S, the size of this set is exactly one more than that of S.

Now we show that the size of any independent set S' of G is at most $1 + |S|$. We make two cases again:

i. At least two of v, u_1, u_2 are in S'.

In this case, since v is adjacent to u_1, u_2, so v can't be in S'. Hence u_1, u_2 are both in S. Consider the corresponding independent set in G' (formed by choosing the vertices in S' except u_1, u_2 but possibly also containing v' – the set without v' is independent because of being a subset of S', but as we show in the next sentence, we can choose v' as well and still have this set as independent in G'). Note that since u_1, u_2 have both been chosen, v' has been chosen in the corresponding independent set in G' (since the neighbours of v' are precisely the neighbours of u_1, u_2 except v, and none of them have been chosen in S' because u_1, u_2 have been chosen), so we have an independent set of size $|S'| - 1$ in G'. Now that we know that the maximum size of an independent set in G' is $|S|$, we have $|S'| - 1 \leq |S|$, or $|S'| \leq 1 + |S|$, as needed.

ii. Exactly one of v, u_1, u_2 is in S'.

Again, considering the corresponding independent set in G', either v' is chosen or it is not, and in either case, we have that the size of that set is either $|S'|$ or $|S'| - 1$. Since the size of the corresponding independent set is at most $|S|$ (as S is a maximum independent set of G'), $|S'|$ is at most $\max(|S|, |S| + 1)$, so we have $|S'| \leq |S| + 1$ in this case as well.

From this analysis, we see that any independent set of G must have at most $|S|+1$ vertices. Since we have exhibited an independent set of G with exactly these many vertices, it must be a maximum independent set of G in each case. □

Note that this finishes the proof of the claim, and thus by the claim (and the construction mentioned in the proof), the independent set returned by the algorithm is indeed a maximum independent set of G.

4. $deg(v) \geq 3$.

In this case, either we choose v or not. If we choose v, the answer is the union of v with the

maximum independent set of the graph obtained after we remove v and all its three neighbours, and if we don't, then the answer is the union of v with the maximum independent set of the graph obtained after we remove v from G, as done in class. Since both of these graphs have number of vertices strictly less than that of G, using the inductive hypothesis, we are done in this case as well.

By the above case analysis, we can see that the algorithm returns a maximum independent set in all cases, and thus the induction is complete.

Time complexity analysis Let n be the number of nodes in the graph, and let n' be as defined. Let the number of constant-time operations needed in this algorithm be denoted by $f(n, n')$. Then we claim that $f(n, n') \leq c(n+1)^2 2^{n'/4}$ for any non-empty graph where c is a suitably chosen large enough constant, which will be specified later on.

We first begin with two claims:

Claim 3. $n' = \sum_{v \in V(G)} \max(0, deg(v) - 2)$.

Proof. Note that for each vertex with degree i contributes $i-2$ to the right hand side if $i \geq 3$ and 0 otherwise. Hence by grouping vertices by degree, we have the result. More precisely, we have:

$$\begin{aligned}
\sum_{v \in V(G)} \max(0, deg(v) - 2) &= \sum_{j \in \mathbb{N}} \sum_{v \in V(G), deg(v) = j} \max(0, j-2) \\
&= \sum_{j \in \mathbb{N}_{\geq 3}} \sum_{v \in V(G), deg(v) = j} (j-2) \\
&= \sum_{i \in \mathbb{N}} \sum_{v \in V(G), deg(v) = i+2} i \\
&= \sum_{i \in \mathbb{N}} i \times |\{v \in V(G) \mid deg(v) = i+2\}| \\
&= n'
\end{aligned}$$

as required. □

Claim 4. *Whenever we create a graph G' from G in the algorithm, $n'(G') \leq n'(G)$.*

Proof. Note that in the cases with minimum degree 0, 1, or at least 3, we always remove vertices from the graph, which either reduces the degree of any remaining vertex or keeps it the same, and also removes the non-negative contribution of the removed vertices from the expression of n'. Hence the inequality holds in that case. The same holds in the subcase of the case of minimum degree 2 where we have an edge between u_1, u_2.

The more important case is when the minimum degree is 2 and we replace a group of nodes with a new node.

In the case when the degree is 2, the highest possible degree of the combined node is $deg(u_1) + deg(u_2) - 2$ (this is in the case when $N(u_1), N(u_2)$ intersect in only one vertex v). The contribution of v, u_1, u_2 to $n'(G)$ is $0, \max(deg(u_1) - 2, 0), \max(deg(u_2) - 2, 0)$ respectively. Also, for any vertex adjacent to both u_1, u_2, its degree in the new graph is one less than its old degree, and this can't increase the difference between $n'(G'), n'(G)$. Hence we have

$$n'(G') - n'(G) \leq \max(deg(u_1) + deg(u_2) - 4, 0) - \max(deg(u_1) - 2, 0) - \max(deg(u_2) - 2, 0)$$

Note that the real function $f(a) = \max(a, 0)$ is the point-wise maximum of the identity function and the 0 function, both of which are convex, and hence it is also convex. So it satisfies $f(a) + f(b) \geq 2f\left(\frac{a+b}{2}\right)$ (special case of Jensen's inequality), and thus applying this inequality to $(a, b) = (deg(u_1) - 2, deg(u_2) - 2)$, we have that

$$\begin{aligned} n'(G') - n'(G) &\leq \max(deg(u_1) + deg(u_2) - 4, 0) - \max(deg(u_1) - 2, 0) - \max(deg(u_2) - 2, 0) \\ &= 2\max\left(\frac{deg(u_1) + deg(u_2) - 4}{2}, 0\right) - \max(deg(u_1) - 2, 0) - \max(deg(u_2) - 2, 0) \\ &\leq 0 \end{aligned}$$

This completes the proof. □

Now we move to the main proof of the time complexity.

Note that all the constants c_i mentioned in the below proof shall be such that the mentioned upper bound holds for all n, and not just the particular n considered in the inductive step.

Proof. The proof goes by strong induction on the lexicographical ordering of pairs (n, n') of non-negative integers where $0 \leq n' \leq n$.

For the base case, if the number of operations to check if the graph is empty is c_1, then if we have $c \geq c_1$, the base case holds.

The number of operations to find out the vertex with the least degree is at most linear in n, by traversing constant-sized adjacency lists for each vertex. Suppose it is upper bounded by $c_2 n$ for all n.

If the degree is 0, then removing the vertex from the graph takes at most linear operations (wherever operations are mentioned, we mean constant-time operations unless specified otherwise). Suppose the time for this and taking the union and returning the set is upper bounded by $c_3 n$ for all n. So we have

$$\begin{aligned} f(n, n') &\leq f(n-1, n') + c_3 n + c_2 n \\ &\leq cn^2 2^{n'/4} + c_3 n + c_2 n \\ &\leq c(n+1)^2 2^{n'/4} \end{aligned}$$

where the last inequality is true if we have $c > c_3 + c_2$.

If the degree is 1, then removing both vertices and returning the set takes linear time, say upper bounded by $c_4 n$ for all n. Then we have

$$\begin{aligned} f(n, n') &\leq \max_{0 \leq i \leq n'} f(n-2, n'-i) + c_4 n + c_2 n \\ &\leq c(n-1)^2 2^{n'/4} + c_4 n + c_2 n \\ &\leq c(n+1)^2 2^{n'/4} \end{aligned}$$

where the last inequality is true if we have $c > c_4 + c_2$.

If the degree is 2, the checking of the two subcases takes constant number of operations, upper bounded by a constant, say c_5 for all n.

In the first subcase, the removal of the vertices to get a new graph, taking the union and returning takes at most $c_6 n$ operations for all n, for some constant c_6. Then we have

$$\begin{aligned} f(n,n') &\leq \max_{0\leq i\leq n'} f(n-3,n'-i) + c_6 n + c_5 + c_2 n \\ &\leq c(n-2)^2 2^{n'/4} + c_6 n + c_5 + c_2 n \\ &\leq c(n-2)^2 2^{n'/4} + c_6 n + c_5 n + c_2 n \\ &\leq c(n+1)^2 2^{n'/4} \end{aligned}$$

where the last inequality is true if $c > c_5 + c_6 + c_2$.

In the second subcase, the creation of the new graph takes a linear number of operations, and so does taking the union, searching in the returned set, modifying the returned set and returning it. Suppose it is upper bounded by $c_7 n$ for all n.

$$\begin{aligned} f(n,n') &\leq \max_{0\leq i\leq n'} f(n-2,n'-i) + c_7 n + c_5 + c_2 n \\ &\leq c(n-1)^2 2^{n'/4} + c_7 n + c_5 + c_2 n \\ &\leq c(n-1)^2 2^{n'/5} + c_7 n + c_5 n + c_2 n \\ &\leq c(n+1)^2 2^{n'/4} \end{aligned}$$

where the last inequality is true if $c > c_7 + c_5 + c_2$.

If none of the above cases is true, then the minimum degree d in the graph must be at least 3, and as a consequence, all vertices in the graph must have degree at least d. Comparing sizes of sets and returning the larger one of them takes linear number of operations, and so does creating two graphs with one and two vertices removed. Suppose all of this is upper bounded by $c_8 n$ for all n.

We compute the difference in n' in this case. When we remove a vertex with degree d, n' decreases by at least $2d-2$, since the contribution of this vertex is $d-2$, and the reduction in the degree of each other vertex is exactly 1. When we remove this vertex and the set of all neighbours of this vertex, the reduction in n' is at least $(d+1)(d-2)$. Let the exact reduction be d'.

We then have:

$$\begin{aligned} f(n,n') &\leq f(n-1,n'-2d+2) + f(n-d-1,n'-d') + c_8 n + c_2 n \\ &\leq cn^2 2^{(n'-2d+2)/4} + c(n-d)^2 2^{(n'-d')/4} + c_8 n + c_2 n \\ &\leq cn^2 2^{(n'-2d+2)/4} + c(n-d)^2 2^{(n'-(d+1)(d-2))/4} + c_8 n + c_2 n \\ &\leq cn^2 2^{(n'-4)/4} + c(n-3)^2 2^{(n'-4)/4} + c_8 n + c_2 n \\ &= \frac{1}{2}\cdot cn^2 2^{n'/4} + \frac{1}{2}\cdot c(n-3)^2 2^{n'/4} + c_8 n + c_2 n \\ &\leq \frac{1}{2}\cdot cn^2 2^{n'/4} + \frac{1}{2}\cdot cn^2 2^{n'/4-1} + c_8 n + c_2 n \\ &= cn^2 2^{n'/4} + c_8 n + c_2 n \\ &\leq c(n+1)^2 2^{n'/4} \end{aligned}$$

where the last inequality is true if $c > c_8 + c_2$. We have made use of the facts that $d \geq 3$, $d' \geq 4d-8$, and that $(d+1)(d-2)$ is increasing on $[2,\infty)$.

Note that if we choose $c = 1 + c_2 + \max(c_1 - c_2, c_3, c_4, c_5 + c_6, c_5 + c_7, c_8)$, all of these claims will be valid, and thus our proof by induction is complete. □

Since our claim is completed, we have that for $n > 0$, our algorithm takes $O((n+1)^2 2^{n'/4})$ time. Since for $n > 0$, we have $n+1 \leq 2n$, so the algorithm is in $O(n^2 2^{n'/4})$, and since $n' \leq n$ for any graph with bounded degree 3, we have that the runtime of the algorithm is in $O(n^2 2^{n/4})$, as required.