1. Find the nth node from the last node of a given singly linked list

Solution 1:

```
node * returnNthFromLast(node * head, int n)
{
        node * ptrToHead1 = head;
        node * ptrToHead2 = head;

        While(ptrToHead1 != NULL) count ++;
        int i = 0;
        while(i < count – n)
        {
                ptrToHead2 = ptrToHead2->next;
                i++;
        }
        return ptrToHead2;
}
```

Take two pointers say p1 and p2. Point p1 and p2 to the head of the given liked list and advance the p1 pointer to the nth node from the head of the linked list. Now the pointer p1 is at the nth node from the head of the given linked list. Now start moving the two pointers parallelly one node at a time. When the pointer p1 becomes NULL (ie, end of the given linked list) the pointer p2 will be at the nth node from the end of the linked list.

Solution 2:

```
node * returnNthFromLast(node * head, int n)
{
        int i = 0;
        node * p1 = head;
        node * p2 = head;

        while(i < n){ p1 = p1->next; }
        while(p1 != NULL){ p1 = p1->next; p2 = p2->next; }

        return p2;
}
```

Complexity:

The complexity of the algorithm is O(n) for first loop and O(n) for second loop. So, the total complexity is O(n) + O(n) = O(n).

<Suresh Chowta> suresh.chowta@gmail.com

2. Find the Middle node of a given singly linked list:

Solution:

Take two pointers p1 and p2 and point them to the head of the given linked list. Now in a loop advance the pointers p1 and p2 as

- p1 = p1->next;
- p2 = p2->next->next;

When the pointer p2 reaches the end of the linked list (becomes NULL) then the pointer p1 will be at the middle node of the given linked list.

Algorithm:

```
node * returnMiddleNode(node * head, int n)
{
        node * p1 = head;
        node * p2 = head;

        while(p1 != NULL)
        {
                p1 = p1->next->next;
                p2 = p2->next;
        }
        return p2;
}
```

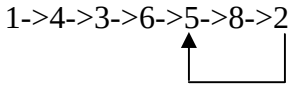
Complexity:

The complexity of the algorithm is O(n).

<Suresh Chowta> suresh.chowta@gmail.com

3. Find a loop in a given singly linked list:

Eg., of loop:



Solution 1:

The easiest way to find a loop in a linked list (not efficient) is take two pointers say p1 and p2 to the head of the linked list and compare each node of the linked list with each and every other node of the given linked list. If at any point of time, if they are equal then there is a loop in the linked list, else there is no loop in the linked list. The algorithm for this method is given below.

```
int isThereALoop(node * head)
{
        node * p1 = head;
        node * p2 = head;

        while(p1 != NULL)
        {
                p2 = head;
                while(p2 != NULL)
                {
                        if(p1 == p2) return 1; //return 1 if there is a loop
                        p2 = p2->next;
                }
                p1 = p1->next;
        }

        return 0; //return 0 if there is no loop
}
```

Complexity:

The complexity of the algorithm is O(n) for the outer while and O(n) for the inner while, leading to a total complexity of $O(n^2)$.

Solution 2:

Take two pointers to the head of the linked list say p1 and p2. Move one pointer slowly and one pointer fastly. Now if they meet at some point of time then there is a loop in the linked list. If any of the pointers become NULL, then there is no loop in the given Linked list.

```
int isThereALoop(node * head)
{
        node * p1 = head;
        node * p2 = head;

        while(p1 != NULL && p2 != NULL)
        {
                p1 = p1->next->next;
                p2 = p2->next;

                if(p1 == p2)
                {
                        return 1; //returns 1 if there is a loop in the given linked list
                }
        }

        return 0;  //returns 0 if there is no loop in the given linked list
}
```

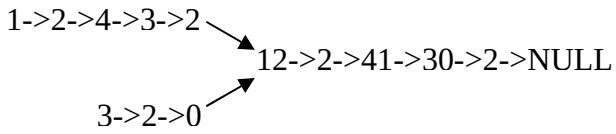
Complexity:

The complexity of the algorithm is O(n).

<Suresh Chowta> suresh.chowta@gmail.com

4. Find the Intersection of two given singly linked lists:

Eg., Intersection of two linked lists



Solution 1:

```
node * returnIntersectingNode(node * head1, node * head2)
{
        node * p1 = head1;
        node * p2 = head2;

        while(p1 != NULL)
        {
                p2 = head2;
                while(p2 != NULL)
                {
                        if(p1 == p2) return p1;
                        p2 = p2->next;
                }
                p1 = p1->next;
        }
        return 0;
}
```

Complexity:

The complexity of the algorithm is $O(n^2)$.

Solution 2:

If the algorithm is given the headers of the above two linked lists the algorithm should return the address of node whose value is 12. Find the length of the first linked list say L1. Find the length of the second list say L2. Find the difference of the two lengths. Take two pointers to the two headers and advance the pointer of the longest list, equal to the difference of the two lists. So, the first pointer points to node with value 4 and the second pointer points to the node with value 3. Now advance the two pointers parallelly until they become equal ie., both the pointers contain the address of 12.

```
node * returnIntersectingNode(node * head1, node * head2)
{
	node * p1 = head1;
	node * p2 = head2;

	int L1 = 0, L2 = 0, i = 0;

	if(p1 == NULL || p2 == NULL) return NULL;

	while(p1 != NULL)    //Find the Length of first List
	{
		L1 = L1 + 1;   p1 = p1->next;
	}

	while(p2 != NULL)    //Find the Length of second List
	{
		L2 = L2 + 1;   p2 = p2->next;
	}

	p1 = head1;
	p2 = head2;

	if(L2 > L1)              //Adjust the Pointer of the longest List depending on
	{                        //the difference of the lengths of the two lists
		while(i <= abs(L2 - L1)) { p1 = p1->next; }
	}
	else
	{
		while(i <= abs(L2 - L1)) { p2 = p2->next; }
	}

	while(p1 != p2)          //Advance the two pointers Until they meet
	{
		p1 = p1->next;
		p2 = p2->next;
	}
	if(p1 == NULL) return 0;
	else return p1;
}
```

Complexity:

The complexity of the algorithm is O(L1 + L2). Where L1 and L2 are the lengths of the two linked lists.

<Suresh Chowta> suresh.chowta@gmail.com

5. Reversing a given singly linked list:

Algorithm:

```
Node * Reverse (Node * head)
{
  Node * pr = NULL;
  Node * tmp;
  while (head != NULL)
  {
    tmp = head->next;
    head->next = pr;
    pr = head;
    head = tmp;
  }
  return pr;
}
```

Complexity:

The complexity of the algorithm is O(n) where n is the length of the given linked list.

<Suresh Chowta> suresh.chowta@gmail.com

6. Delete a node from a singly linked list when the pointer to the node to be deleted is given:

It is easy to delete a node if given a pointer to the previous node of the node to be deleted and if given a pointer to the node that is to be deleted then there is no perfect way of doing that, but it can be done as below

Copy the value of the next node to the current node, which is to be deleted and delete the next node but this, may cause logical errors in the program.

Algorithm:

```
void deleteNode(node * n)
{
        if(n == NULL) return;

        node * temp = n->next;
        n->value = temp->value;
        n->next = n->next->next;
        free(temp);
}
```

<Suresh Chowta> suresh.chowta@gmail.com

7. Print the values 1 to n without using any loop:

Solution:

Use recursion to print the values from 1 to n.

Algorithm:

```
Void print1ton(int n)
{
        if(n>1)
        {
                print(n-1);
        }
        printf("%d\t", n);
}
```

Complexity:

The complexity of the algorithm is O(n).

8. Find the missing number:

An array of size n is given, which is filled with the numbers from 1 to n randomly and uniquely in the array. One of the entries in the array is made zero. The algorithm should return the number, which is replaced by zero.

```
int returnMissingNumber(int * array, int n)
{
        int sum = n*(n+1)/2;
        int tsum = 0, i;

        for(i = 0;i < n; i++) sum = sum + array[i];

        return sum – tsum;
}
```

(OR)

```
int returnMissingNumber(int * array, int n)
{       //This algorithm avoids the overflow problem during summation.
        int xor = 0;

        for(i = 1;i <= n; i++) xor = xor ^ i;
        for(i = 0;i < n; i++) xor = xor ^ array[i];
        return xor;
}
```

Complexity:

The complexity of the algorithm is O(n).

<Suresh Chowta> suresh.chowta@gmail.com

9. Find the Number which is present only once in the array:

An array of size 2n+1 is given. Initially the array is filled with numbers randomly. Except one number "a" every number is repeated. ie., except one number "a", every other number is present twice in the array. The algorithm should find the number "a".

Eg., 1, 2, 6, 3, 5, 5, 2, 1, 6, 3, 9

In the above example value of "a" is 9, as every other number is repeated except "9".

Solution 1:

```
int returnSingleNumber(int * array, int n)
{
        int i, j;
        for(i = 0;i < n; i++)
        {
                for(j = 0;j < n; j++)
                {
                        if(array[i] == array[j]) break;
                }
                if(j == n) return array[i];
        }
        return NULL;
}
```

Complexity:

Since the algorithm is having two for loops (for loop within a for loop) the complexity of the algorithm is $O(n^2)$.

Solution 2:

```
int returnSingleNumber(int * array, int n)
{
        int xor = 0;

        for(i = 0;i < n; i++) xor = xor ^ array[i];

        return xor;
}
```

Complexity:

The complexity of the algorithm is O(n).

<Suresh Chowta> suresh.chowta@gmail.com

10. Find the two integers, whose sum is "C" in the given array:

Given an array of integers of size "n", find any two integers from the array whose sum is equal to given constant "C". The general method for finding this is, test each and every combination of integers present in the array. So, the complexity of this method would be $O(n^2)$. Three methods are given below to solve this problem whose complexities are $O(n^2)$, O(nlog(n)) and O(n).

Algorithm:

Method 1: $O(n^2)$ Complexity

```
Void printCombinations(int * array, const int c, const int n)
{
        for(int i=0;i<n;i++)
                for(int j=0;j<n;j++)
                {
                        if(i == j) continue;
                        if(array[i] + array[j] == c)
                        {
                                printf("%d, %d", array[i], array[j]);
                        }
                }
}
```

Method 2: O(nlog(n)) Complexity

```
Void printCombinations(int * array, const int c, const int n)
{
        sort(array, n); // Use any sorting algorithm that needs O(nlog(n))

        int i = 0, j = n – 1;

        while (i < j)
        {
                if(array[i] + array[j] == c)
                {
                        printf("%d, %d", array[i], array[j]);
                }
                if(array[i] + array[j] < c) i++;
                else j--;
        }
}
```

Method 3: O(n) Complexity (with O(n) Space Complexity for hash table)

Void printCombinations(int * array, const int c, const int n)
{

1. Hash all the n numbers.
2. Scan the array from left to right
3. Say the number being considered is “a” then subtract the number “a” from “c” to get the other number “b”.
4. Hash the number “b” onto the same hash to check for the existence.
5. If the number “b”exists then print “a” and “b”.
6. If no more elements remain to scan in “array” then exit, else goto step 2.

// Note: A perfect hash function is assumed in this case.
}

This solution is not possible practically as there are no possible perfect hash functions existing.

<Suresh Chowta> suresh.chowta@gmail.com

11. Algorithm to print the contents of linked list in reverse order:

Algorithm:

```
void print_reverse(node *head)
{
        if(head->next == NULL)
        {
                print(head->info);
                return;
        }

        print_reverse(head->next);
        print(head->info);

}
```

<Suresh Chowta> suresh.chowta@gmail.com

12. Reversing the position of two sub-arrays in a given array:

eg.,

given A = <1, 2, 3, 4, 5>
where B = <1, 2, 3> and C = <4, 5>

Now reversing the postion of B and C will result in A = <4, 5, 1, 2, 3>

Solution:

```
void reverse(int * A, Bi, Bj, Ci, Cj)
{
    1. Reverse the elements from Bi to Bj
    2. Reverse the elemtnts from Ci to Cj
    3. Reverse the array from Bi to Cj
}
```

13. Print the Last "N" lines of the File (File should be read only once):

Solution 1:

A bad solution for this problem is read the whole file line by line into an array of strings and print the last "N" strings from that array of strings, which would require a space equal to the size of the file. This solution is not even possible to implement if the file size is too large.

Solution 2:

Another efficient solution is using a circular queue of size "N" and keep on inserting the lines read from the file. If the queue is filled delete an element from the queue and insert the new line into the queue. When the file is exhausted we will have the last "N" lines in the queue. **This is how the tail command in UNIX is implemented.**

The time complexity for both the above solutions is same. But in the first case the space needed is equal to the size of the file and in the second case the space needed is just "N" (the number of lines to be printed).

14. Reverse the words of a given sentence:

eg.,

If the given sentence is “This is a question on algorithms”
The algorithm should return “algorithms on question a is This”

Solution:

The main idea of the algorithm given below is to just reverse the given string till it finds the NULL and then reverse word by word of this reversed sentence.

For example for the above given sentence… The first step results in “smhtirogla no noitseuq a si sihT”. The second step reverses each and every word from this sentence till the space and results in “algorithms on question a is This”

Assume a function reverse(char * string, char delimiter) which reverses the given string till the delimiter. Then the algorithms to solve the problem is given below

```
void reverseSpecial(char * string)
{
        char * ptr = string;
        char * tptr = string;

        reverse(ptr, '\0'); // Reverse the whole string till it finds NULL

        while(*ptr != NULL)
        {
                if(*ptr == ' ') // Check for space
                {
                        reverse(tptr, ' ');  // Reverse the word
                        tptr = ptr+1;
                }
                ptr = ptr + 1;
        }
}
```

15. Find the elements that are present in the both the lists (Intersection):

Assume the two lists are the two arrays of different sizes. The problem is to find the intersection of the two given arrays.

Solution 1:

The simple solution for this problem would be comparing every element from the first array with each and every element of the second array. If they match then output the element. This solution needs O(m) for the outerloop and O(n) for the inner loop leading to a total complexity of O(mn).

If the arrays are A and B of size m and n then the solution will look like

```
for(i = 1 to n)
{
        for(j = 1 to m)
        {
                if(A[i] == B[j]) print A[i];
        }
}
```

Solution 2:

The better solution compared to the above solution is Sort the two arrays, which will required mlog(m) and nlog(n) time complexity. Now start comparing from the starting points of the two arrays and find the intersecting elements. So, the total complexity of the algorithm is O(mlog(m) + nlog(n))

```
Sort(A, m) //O(mlog(m))
Sort(B, n)  //O(nlog(n))

i = 0, j = 0;

while ( i<m && j<n) // O(m+n)
{
        if(A[i] == B[j]) print A[i];
        else if(A[i] < B[j]) i++;
        else j++;
}
```

The total complexity of the algorithm is O(mlog(m) + nlog(n) + m + n) which is equal to O(mlog(m) + nlog(n)).

<Suresh Chowta> suresh.chowta@gmail.com

Solution 3:

Another Solution which needs a complexity of O(m+n) which is relatively fast compared to both the above solutions and needs an extra space of O(m+n) is

1. Hash the first array.
2. Hash the second array elements to the same hash table one at a time
    a) If there is a collision then output the element and continue to step 2 if there are elements in the second array
    b) Else continue to step 2 if there are elements existing in the second array.
    c) Else exit;

This solution is not possible practically as there are no possible perfect hash functions existing.

<Suresh Chowta> suresh.chowta@gmail.com

16. Inorder, preorder and postorder traversals of a given binary tree:

In-order:

```
Void printInorder(node * tree)
{
        if(tree == NULL) return;
        printInorder(tree->left);
        printf("%d", tree->value);
        printInorder(tree->right);
}
```

Pre-order:

```
Void printPreorder(node * tree)
{
        if(tree == NULL) return;
        printf("%d", tree->value);
        printPreorder(tree->left);
        printPreorder(tree->right);
}
```

Post-order:

```
Void printPostorder(node * tree)
{
        if(tree == NULL) return;
        printPostorder(tree->left);
        printPostorder(tree->right);
        printf("%d", tree->value);
}
```

17. Count the Number of Leaves present in the given binary tree:

All the nodes in a given binary tree can be categorized into three types

1. Nodes with two children
2. Nodes with left child and right child equal to NULL
3. Nodes with right child and left child equal to NULL
4. Nodes with both left and right childs as NULL

Here we have to count the Number of nodes, which have both the left and right child as NULL (Nodes with no children).

Solution:

```
Int returnNoOfLeaves (node * tree)
{
        if(tree == NULL) return 0;
        if(tree->left == NULL and tree->right == NULL) return 1;

        return returnNoOfLeaves(tree->left) + returnNoOfLeaves(tree->right);
}
```

18. Check wether the given combination of braces is valid or not:

eg.,

A given combination of braces is valid only if for each and every open brace there is a closed brace associated with it and the whole combination of braces is semantically correct.

Eg., The combination (())(((()() is not valid because there is no closing combination for the third and fourth opening braces.

Eg., On the other hand, the combination (())(((()())) is valid because, it is semantically correct.

Solution 1:

1. Take a stack "S"
2. Scan the given string
3. If the scanned character is "(" push onto the stack. Continue to next character.
4. If the scanned character is ")" check with the top of the stack. If the top of the stack is "(" then pop out the stack and continue to the next charcter.
5. Else push the scanned character onto the stack.
6. Continue to the next character and goto step 2 if the character is not NULL
7. If the stack is empty then report "valid" else report "invalid"

Solution 2:

Instead of taking the stack to keep track of the opening braces, we can use a counter and can reduce the space required by the solution.

1. Scan the given String
2. If the character scanned is "(" increment the counter
3. If the character scanned is ")" and counter is "0" then report "invalid" else decrement the counter by 1.
4. Continue to the next character and goto step 2 if the character is not NULL

19. Check wether a given integer is a power of two or not:

Solution 1:

The easiest solution to check wether the given integer is a power of 2 or not is to check the binary representation of the given integer and count the number of 1's present in the binary representation of the integer. If the number of 1's present in the binary representation of the given integer is equal to 1 then we can say that the given integer is a power of 2.

```
Int isPowerOfTwo(int N)
{
        int count = 0;
        int i;

        for(i=0; i<32; i++)
        {
                if( pow(2, i) & N) count++;
        }

        if(count == 1) return 1; // Return 1 if true

        return 0; // Return 0 if false
}
```

The above solution has a time complexity of O(n). But a better solution with time complexity equal to O(1) exists and the algorithm is given below.

Solution 2:

The basic idea in this method is to check wether there are any extra 1's present in the binary representation. To do this just obtain the two's compliment of the given integer and perform logical AND with the actual integer. If the result is equal to the actual integer then we can say that the number provided is a power of two.

```
Int isPowerOfTwo(int N)
{
        if( N & (~N+1) == N) return 1;
        return 0;
}
```