**Jose Carlos Moreno Soria**

**Problem 1**

#include <iostream>

using namespace std;

int main(){

int nums[] = {3,3};

int target = 6;

int x=-1;

int y=x;

int j=0;

for(int i=0; j < sizeof(nums); i++){

if(i>=sizeof(nums)){

j++;

i=j;

}

if(i!=j&&nums[i]+nums[j]==target){

y=i;

x=j;

break;

}

}

cout<<"["<<x<<", "<<y<<"]"<<endl;

return 0;

}

The time complexity is f(n)=O(nlogn) because the largest variable is how many loops are executed, which is upper bounded by checking each element in the array with every other element to see if it sums to the target, this requires in the worst case scenario a loop of the array length n, then another loop of n-1, then n-2, until n-n which ends the search. The worst case scenario would be both numbers for the solution being at the very end of the array or there being no solution to the target.

**Problem 2**

#include <iostream>

using namespace std;

int main()

{

int nums[] = {5,7,7,8,8,10};

int target = 8;

int x=-1;

int y=x;

for(int i=0; i < sizeof(nums); i++){

if(x==-1&&nums[i]==target){

x=i;

}

else if(nums[i]==target){

y=i;

}

}

cout<<"["<<x<<", "<<y<<"]"<<endl;

return 0;

}

The time complexity is f(n)=O(n) because the largest variable is how many loops are executed, which is upper bounded by the size of the array entered. The worst case scenario would be having the last instance of the target be the last element of the array or the search not finding the target.

**Problem 3**

**WRONG ANSWER, COME BACK TO THIS**

#include <iostream>

using namespace std;

int main()

{

int nums1[] = {1,2};

int nums2[] = {3,4,5};

int l1=sizeof(nums1)/sizeof(int);

int l2=sizeof(nums2)/sizeof(int);

int len = l1+l2;

bool isOdd=true;

if(len%2==0){isOdd=false;}

double median=-1;

int len2 = 1+len/2;

int merged[len2];

int x=0;

int y=x;

for(int i=0; i<len2; i++){

if(x==l1){

merged[i]=nums2[y];

y++;

continue;

}

if(y==l2){

merged[i]=nums1[x];

x++;

continue;

}

if(nums1[x]<nums2[y]){

merged[i]=nums1[x];

x++;

}

else{

merged[i]=nums2[y];

y++;

}

}

median=merged[len2-1];

if(isOdd==false){median = (median+merged[len2-2])/2;}

cout<<median<<endl;

return 0;

}

The time complexity is f(n)=O(n/2) because to find the median, it is not necessary to go through every element of both lists, instead both lists only need to be sorted until they reach the midpoint of the length of both lists combined.

**Problem 4**

#include <iostream>

using namespace std;

struct node{

node\* next=nullptr;

int value;

}

node\* cutList(int n, node\* list){

node\* head = list;

node\* cut = head;

int length=0;

while(cut->next!=nullptr){

cut=cut->next;

length++;

}

length-=n;

while(length>0){

cut=cut->next;

length--;

}

cut->next=nullptr;

return head;

}

int main()

{

int n = 3;

node\* clist = cutList(n, list);

return 0;

}

The time complexity is f(n)=O(2n) because the function must first go through the list in full once to determine the length and then a second time until the point where it will cut the list. In the worst case scenario, only the last node in the list will be removed and so it will execute the loop 2n-1 times.

**Problem 5**

#include <iostream>

using namespace std;

struct node{

node\* next=nullptr;

int value;

};

node\* mergeLists(node\* lists[]){

node\* sortedHead;

node\* sortedTail = sortedHead;

bool notdone=true;

int min=INT\_MAX;

node\* minNode;

while(notdone){

int a=-1;

for(int i=0; i<sizeof(lists); i++){

if(lists[i]!=nullptr&&lists[i]->value<min){

a=i;

minNode = lists[i];

min=minNode->value;

}

}

if(a==-1){notdone=false; break;}

sortedTail=minNode;

sortedTail=sortedTail->next;

lists[a]=lists[a]->next;

}

return sortedHead;

}

int main()

{

node\* lists[] = {};

node\* sortedList = mergLists(lists);

return 0;

}

The time complexity is f(n)=O(m\*n) because the dominant operation is the while loop and for loop, the while loop being determined by the number of elements, the length of the final sorted list, and the for loop being determined by the length of the array of linked lists.