Linked Lists

How do you reverse a singly linked list? How do you reverse a doubly linked list? Write a C program to do the same.

This is THE most frequently asked interview question. The most!.
Singly linked lists
Here are a few C programs to reverse a singly linked list.

Method1 (Iterative) -

```
#include <stdio.h>

// Variables
typedef struct node
{
   int value;
   struct node *next;
}mynode;


// Globals (not required, though).
mynode *head, *tail, *temp;


// Functions
void add(int value);
void iterative_reverse();
void print_list();


// The main() function
int main()
{
    head=(mynode *)0;

    // Construct the linked list.
    add(1);
    add(2);
    add(3);

    //Print it
    print_list();

    // Reverse it.
    iterative_reverse();

    //Print it again
    print_list();

    return(0);
}
```

```
// The reverse function
void iterative_reverse()
{
     mynode *p, *q, *r;

     if(head == (mynode *)0)
     {
        return;
     }
     p       = head;
     q       = p->next;
     p->next = (mynode *)0;


     while (q != (mynode *)0)
     {
        r       = q->next;
        q->next = p;
        p       = q;
        q       = r;
     }

     head = p;
}


// Function to add new nodes to the linked list
void add(int value)
{
   temp = (mynode *) malloc(sizeof(struct node));
   temp->next=(mynode *)0;
   temp->value=value;

   if(head==(mynode *)0)
   {
      head=temp;
      tail=temp;
   }
   else
   {
     tail->next=temp;
     tail=temp;
   }
}


// Function to print the linked list.
void print_list()
{
  printf("\n\n");
  for(temp=head; temp!=(mynode *)0; temp=temp->next)
  {
     printf("[%d]->",(temp->value));
  }
  printf("[NULL]\n\n");
}
```

Method2 (Recursive, without using any temporary variable)

```
#include <stdio.h>

// Variables
typedef struct node
{
   int value;
   struct node *next;
}mynode;

// Globals.
mynode *head, *tail, *temp;


// Functions
void add(int value);
mynode* reverse_recurse(mynode *root);
void print_list();


// The main() function
int main()
{
    head=(mynode *)0;

    // Construct the linked list.
    add(1);
    add(2);
    add(3);

    //Print it
    print_list();

    // Reverse it.
    if(head != (mynode *)0)
    {
      temp = reverse_recurse(head);
      temp->next = (mynode *)0;
    }

    //Print it again
    print_list();

    return(0);
}


// Reverse the linked list recursively
//
// This function uses the power of the stack to make this
// *magical* assignment
//
// node->next->next=node;
//
// :)
```

```
mynode* reverse_recurse(mynode *root)
{
  if(root->next!=(mynode *)0)
  {
      reverse_recurse(root->next);
      root->next->next=root;
      return(root);
  }
  else
  {
      head=root;
  }
}



// Function to add new nodes to the linked list.
void add(int value)
{
     temp = (mynode *) malloc(sizeof(struct node));
     temp->next=(mynode *)0;
     temp->value=value;

     if(head==(mynode *)0)
     {
        head=temp;
        tail=temp;
     }
     else
     {
       tail->next=temp;
       tail=temp;
     }
}


// Function to print the linked list.
void print_list()
{
    printf("\n\n");
    for(temp=head; temp!=(mynode *)0; temp=temp->next)
    {
       printf("[%d]->",(temp->value));
    }
    printf("[NULL]\n\n");
}
```

Method3 (Recursive, but without ANY global variables. Slightly messy!)

```
#include <stdio.h>

// Variables
typedef struct node
{
   int value;
   struct node *next;
}mynode;


// Functions
void add(mynode **head, mynode **tail, int value);
mynode* reverse_recurse(mynode *current, mynode *next);
void print_list(mynode *);


int main()
{
    mynode *head, *tail;
    head=(mynode *)0;

    // Construct the linked list.
    add(&head, &tail, 1);
    add(&head, &tail, 2);
    add(&head, &tail, 3);

    //Print it
    print_list(head);

    // Reverse it.
    head = reverse_recurse(head, (mynode *)0);

    //Print it again
    print_list(head);

    getch();
    return(0);
}
```

```
// Reverse the linked list recursively
mynode* reverse_recurse(mynode *current, mynode *next)
{
  mynode *ret;

  if(current==(mynode *)0)
  {
    return((mynode *)0);
  }

  ret = (mynode *)0;
  if (current->next != (mynode *)0)
  {
     ret = reverse_recurse(current->next, current);
  }
  else
  {
     ret = current;
  }

  current->next = next;
  return ret;
}


// Function to add new nodes to the linked list.
// Takes pointers to pointers to maintain the
// *actual* head and tail pointers (which are local to main()).

void add(mynode **head, mynode **tail, int value)
{
     mynode *temp1, *temp2;

     temp1 = (mynode *) malloc(sizeof(struct node));
     temp1->next=(mynode *)0;
     temp1->value=value;

     if(*head==(mynode *)0)
     {
        *head=temp1;
        *tail=temp1;
     }
     else
     {
        for(temp2 = *head; temp2->next!= (mynode *)0; temp2=temp2-
>next);
        temp2->next = temp1;
        *tail=temp1;
     }
}
```

```
// Function to print the linked list.
void print_list(mynode *head)
{
    mynode *temp;
    printf("\n\n");
    for(temp=head; temp!=(mynode *)0; temp=temp->next)
    {
       printf("[%d]->",(temp->value));
    }
    printf("[NULL]\n\n");
}
```

Doubly linked lists

This is really easy, just keep swapping the prev and next pointers and at the end swap the head and the tail:)

```
#include<stdio.h>
#include<ctype.h>

typedef struct node
{
  int value;
  struct node *next;
  struct node *prev;
}mynode ;

mynode *head, *tail;
void add_node(int value);
void print_list();
void reverse();

int main()
{

 head=NULL;
 tail=NULL;

 add_node(1);
 add_node(2);
 add_node(3);
 add_node(4);
 add_node(5);

 print_list();

 reverse();

 print_list();

 return(1);

}
```

```
void add_node(int value)
{
  mynode *temp, *cur;
  temp = (mynode *)malloc(sizeof(mynode));
  temp->next=NULL;
  temp->prev=NULL;

  if(head == NULL)
  {
    printf("\nAdding a head pointer\n");
    head=temp;
    tail=temp;
    temp->value=value;

  }
  else
  {
   for(cur=head;cur->next!=NULL;cur=cur->next);
   cur->next=temp;
   temp->prev=cur;
   temp->value=value;
   tail=temp;

  }

}
void print_list()
{
  mynode *temp;

  printf("\n--------------------------------\n");
  for(temp=head;temp!=NULL;temp=temp->next)
  {
    printf("\n[%d]\n",temp->value);
  }

}

void reverse()
{
  mynode *cur, *temp, *save_next;
  if(head==tail)return;
  if(head==NULL || tail==NULL)return;
  for(cur=head;cur!=NULL;)
  {
    printf("\ncur->value : [%d]\n",cur->value);
    temp=cur->next;
    save_next=cur->next;
    cur->next=cur->prev;
    cur->prev=temp;
    cur=save_next;
  }

  temp=head;
  head=tail;
  tail=temp;
}
```

Given only a pointer to a node to be deleted in a singly linked list, how do you delete it?

This is a very good interview question

The solution to this is to copy the data from the next node into this node and delete the next node!. Ofcourse this wont work if the node to be deleted is the last node. Mark it as dummy in that case. If you have a Circular linked list, then this might be all the more interesting. Try writing your own C program to solve this problem. Having a doubly linked list is always better.

How do you sort a linked list? Write a C program to sort a linked list.

This is a very popular interview question, which most people go wrong. The ideal solution to this problem is to keep the linked list sorted as you build it. Another question on this website teaches you how to insert elements into a linked list in the sorted order. This really saves a lot of time which would have been required to sort it.

However, you need to Get That Job....

Method1 (Usual method)

The general idea is to decide upon a sorting algorithm (say bubble sort). Then, one needs to come up with different scenarios to swap two nodes in the linked list when they are not in the required order. The different scenarios would be something like

```
1. When the nodes being compared are not adjacent and one of them is th
e first node.
2. When the nodes being compared are not adjacent and none of them is t
he first node
3. When the nodes being compared are adjacent and one of them is the fi
rst node.
4. When the nodes being compared are adjacent and none of them is the f
irst node.
```

One example bubble sort for a linked list goes like this (working C code!)....

```
#include<stdio.h>
#include<ctype.h>

typedef struct node
{
  int value;
  struct node *next;
  struct node *prev;
}mynode ;

void add_node(struct node **head, int *count, int value);
void print_list(char *listName, struct node *head);
mynode *bubbleSort(mynode *head, int count);


int main()
{
 mynode *head;
 int count = 0;

 head = (struct node *)NULL;

 add_node(&head, &count, 100);
 add_node(&head, &count, 3);
 add_node(&head, &count, 90);
 add_node(&head, &count, 7);
 add_node(&head, &count, 9);

 print_list("myList(BEFORE)", head);
 head = bubbleSort(head, count);
 print_list("myList(AFTER) ", head);


 getch();
 return(0);

}


mynode *bubbleSort(mynode *head, int count)
{
  int i, j;
  mynode *p0, *p1, *p2, *p3;

  for(i = 1; i < count; i++)
  {
    p0 = (struct node *)NULL;
    p1 = head;
    p2 = head->next;
    p3 = p2->next;
 for(j = 1; j <= (count - i); j++)
    {
       if(p1->value > p2->value)
       {
           // Adjust the pointers...
           p1->next = p3;
           p2->next = p1;
           if(p0)p0->next=p2;


           // Set the head pointer if it was changed...
           if(head == p1)head=p2;


           // Progress the pointers
           p0 = p2;
           p2 = p1->next;
           p3 = p3->next!=(struct node *)NULL?p3-
>next: (struct node *)NULL;

       }
       else
       {
          // Nothing to swap, just progress the pointers...
          p0 = p1;
          p1 = p2;
          p2 = p3;
          p3 = p3->next!=(struct node *)NULL?p3-
>next: (struct node *)NULL;
       }
    }
  }
  return(head);
}

void add_node(struct node **head, int *count, int value)
{
  mynode *temp, *cur;
  temp = (mynode *)malloc(sizeof(mynode));
  temp->next=NULL;
  temp->prev=NULL;

  if(*head == NULL)
  {
     *head=temp;
     temp->value=value;
  }
  else
  {
   for(cur=*head;cur->next!=NULL;cur=cur->next);
   cur->next=temp;
   temp->prev=cur;
   temp->value=value;
  }

  *count = *count + 1;
}
```

```
void print_list(char *listName, struct node *head)
{
  mynode *temp;

  printf("\n[%s] -> ", listName);
  for(temp=head;temp!=NULL;temp=temp->next)
  {
    printf("[%d]->",temp->value);
  }

  printf("NULL\n");

}
```

As you can see, the code becomes quite messy because of the pointer logic. Thats why I have not elaborated too much on the code, nor on variations such as sorting a doubly linked list. You have to do it yourself once to understand it.

Method2 (Divide and Conquer using Merge Sort)
Here is some cool working C code...

```
#include<stdio.h>
#include<ctype.h>

typedef struct node
{
  int value;
  struct node *next;
  struct node *prev;
}mynode ;

void add_node(struct node **head, int value);
void print_list(char *listName, struct node *head);
void mergeSort(struct node** headRef);
struct node *merge2SortedLLs(struct node *head1, struct node *head2);
void splitLLInto2(struct node* source, struct node** frontRef, struct n
ode** backRef);

// The main function..
int main()
{
   mynode *head;
   head = (struct node *)NULL;
   add_node(&head, 1);
   add_node(&head, 10);
   add_node(&head, 5);
   add_node(&head, 70);
   add_node(&head, 9);
   add_node(&head, -99);
   add_node(&head, 0);
   print_list("myList", head);
   mergeSort(&head);
   print_list("myList", head);
   getch();
   return(0);
}
```

```
// This is a recursive mergeSort function...
void mergeSort(struct node** headRef)
{
  struct node* head = *headRef;
  struct node* a;
  struct node* b;

  // Base case -- length 0 or 1
  if ((head == NULL) || (head->next == NULL))
  {
    return;
  }

  // Split head into 'a' and 'b' sublists
  splitLLInto2(head, &a, &b);

  // Recursively sort the sublists
  mergeSort(&a);
  mergeSort(&b);

  // Merge the two sorted lists together
  *headRef = merge2SortedLLs(a, b);
}



// This is an iterative function that joins two already sorted
// Linked lists...
struct node *merge2SortedLLs(struct node *head1, struct node *head2)
{
   struct node *a, *b, *c, *newHead, *temp;

   a = head1;
   b = head2;
   c       = (struct node *)NULL;
   newHead = (struct node*)NULL;


   if(a==NULL)return(b);
   else if(b==NULL)return(a);

   while(a!=NULL && b!=NULL)
   {
     if(a->value < b->value)
     {
       //printf("\na->value < b->value\n");
       if(c==NULL)
       {
         c  = a;
       }
       else
       {
         c->next = a;
         c = c->next;
       }
       a  = a->next;
     }
```

```
      else if(a->value > b->value)
      {
        //printf("\na->value > b->value\n");
        if(c==NULL)
        {
          c  = b;
        }
        else
        {
          c->next = b;
          c = c->next;
        }
        b  = b->next;
      }
      else
      {
        // Both are equal.
        // Arbitraritly chose to add one of them and make
        // sure you skip both!

        if(c == NULL)
        {
          c  = a;
        }
        else
        {
          c->next = a;
          c = c->next;
        }

        a  = a->next;
        b  = b->next;
      }

      // Make sure the new head is set...
      if(newHead == NULL)
       newHead = c;

    }

    if(a==NULL && b==NULL)
      return(newHead);

    if(a==NULL)
      c->next = b;
    else if(b==NULL)
      c->next = a;

    return(newHead);

}

// Uses the fast/slow pointer strategy
//
// This efficient code splits a linked list into two using
// the same technique as the one used to find the
// middle of a linked list!
```

```
void splitLLInto2(struct node* source, struct node** frontRef, struct n
ode** backRef)
{
  struct node* fast;
  struct node* slow;

  if (source==NULL || source->next==NULL)
  {
    // length < 2 cases
    *frontRef = source;
    *backRef = NULL;
  }
  else
  {
    slow = source;
    fast = source->next;
    // Advance 'fast' two nodes, and advance 'slow' one node
    while (fast != NULL)
    {
       fast = fast->next;
       if (fast != NULL)
       {
          slow = slow->next;
          fast = fast->next;
       }
    }
    // 'slow' is before the midpoint in the list, so split it in two
    // at that point.
    *frontRef = source;
    *backRef = slow->next;
    slow->next = NULL;
  }
}


void add_node(struct node **head, int value)
{
  mynode *temp, *cur;
  temp = (mynode *)malloc(sizeof(mynode));
  temp->next=NULL;
  temp->prev=NULL;

  if(*head == NULL)
  {
     *head=temp;
     temp->value=value;
  }
  else
  {
   for(cur=*head;cur->next!=NULL;cur=cur->next);
   cur->next=temp;
   temp->prev=cur;
   temp->value=value;
  }
}
```

```
void print_list(char *listName, struct node *head)
{
  mynode *temp;

  printf("\n[%s] -> ", listName);
  for(temp=head;temp!=NULL;temp=temp->next)
  {
    printf("[%d]->",temp->value);
  }

  printf("NULL\n");

}
```

The code to merge two already sorted sub-linked lists into a sorted linked list could be either iterative or recursive. You already saw the iterative version above. Here is a recursive version of the same...

Recursive solution to merge two already sorted linked lists into a single linked list

```
struct node* sortedMergeRecursive(struct node* a, struct node* b)
{
  struct node* result = NULL;

  if (a==NULL) return(b);
  else if (b==NULL) return(a);

  // Pick either a or b, and recur
  if (a->data <= b->data)
  {
     result = a;
     result->next = sortedMergeRecursive(a->next, b);
  }
  else
  {
     result = b;
     result->next = sortedMergeRecursive(a, b->next);
  }

  return(result);
}
```

Also, see how the splitLLInto2() function uses the same technique used to find the middle of a linked list to split a linked list into two without having to keep a count of the number of nodes in the linkes list!

Here is another solution (not that great, though) to split a linked list into two. It used the count of the number of nodes to decide where to split

```
void splitLLInto2(struct node* source,struct node** frontRef, struct no
de** backRef)
{
  int len = Length(source); //Get the length of the original LL..
  int i;
  struct node* current = source;

  if (len < 2)
  {
    *frontRef = source;
    *backRef = NULL;
  }
  else
  {
     int hopCount = (len-1)/2;

     for (i = 0; i<hopCount; i++)
     {
        current = current->next;
     }

     // Now cut at current
    *frontRef = source;
    *backRef = current->next;
    current->next = NULL;
  }
}
```

Using recursive stack space proportional to the length of a list is not recommended. However, the recursion in this case is ok ? it uses stack space which is proportional to the log of the length of the list. For a 1000 node list, the recursion will only go about 10 deep. For a 2000 node list, it will go 11 deep. If you think about it, you can see that doubling the size of the list only increases the depth by 1.

How to declare a structure of a linked list?

The right way of declaring a structure for a linked list in a C program is

```
struct node {
  int value;
  struct node *next;
};
typedef struct node *mynode;
```

Note that the following are not correct

```
typedef struct {
  int value;
  mynode next;
} *mynode;
```

The typedef is not defined at the point where the "next" field is declared.

```
struct node {
  int value;
  struct node next;
};
typedef struct node mynode;
```

You can only have pointer to structures, not the structure itself as its recursive!

Write a C program to implement a Generic Linked List.

Here is a C program which implements a generic linked list. This is also one of the very popular interview questions thrown around. The crux of the solution is to use the void C pointer to make it generic. Also notice how we use function pointers to pass the address of different functions to print the different generic data.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct list {
    void *data;
    struct list *next;
} List;

struct check {
    int i;
    char c;
    double d;
} chk[] = { { 1, 'a', 1.1 },
            { 2, 'b', 2.2 },
            { 3, 'c', 3.3 } };

void insert(List **, void *, unsigned int);
void print(List *, void (*)(void *));
void printstr(void *);
void printint(void *);
void printchar(void *);
void printcomp(void *);

List *list1, *list2, *list3, *list4;

int main(void)
{
    char c[]    = { 'a', 'b', 'c', 'd' };
    int i[]     = { 1, 2, 3, 4 };
    char *str[] = { "hello1", "hello2", "hello3", "hello4" };

    list1 = list2 = list3 = list4 = NULL;

    insert(&list1, &c[0], sizeof(char));
    insert(&list1, &c[1], sizeof(char));
    insert(&list1, &c[2], sizeof(char));
    insert(&list1, &c[3], sizeof(char));

    insert(&list2, &i[0], sizeof(int));
    insert(&list2, &i[1], sizeof(int));
    insert(&list2, &i[2], sizeof(int));
    insert(&list2, &i[3], sizeof(int));

    insert(&list3, str[0], strlen(str[0])+1);
    insert(&list3, str[1], strlen(str[0])+1);
    insert(&list3, str[2], strlen(str[0])+1);
    insert(&list3, str[3], strlen(str[0])+1);

    insert(&list4, &chk[0], sizeof chk[0]);
    insert(&list4, &chk[1], sizeof chk[1]);
    insert(&list4, &chk[2], sizeof chk[2]);

    printf("Printing characters:");
    print(list1, printchar);
    printf(" : done\n\n");

    printf("Printing integers:");
    print(list2, printint);
    printf(" : done\n\n");

    printf("Printing strings:");
    print(list3, printstr);
    printf(" : done\n\n");

    printf("Printing composite:");
    print(list4, printcomp);
    printf(" : done\n");

    return 0;
}

void insert(List **p, void *data, unsigned int n)
{
    List *temp;
    int i;

    /* Error check is ignored */
    temp = malloc(sizeof(List));
    temp->data = malloc(n);
    for (i = 0; i < n; i++)
        *(char *)(temp->data + i) = *(char *)(data + i);
    temp->next = *p;
    *p = temp;
}

void print(List *p, void (*f)(void *))
{
    while (p)
    {
        (*f)(p->data);
        p = p->next;
    }
}
```

```
void printstr(void *str)
{
    printf(" \"%s\"", (char *)str);
}

void printint(void *n)
{
    printf(" %d", *(int *)n);
}

void printchar(void *c)
{
    printf(" %c", *(char *)c);
}

void printcomp(void *comp)
{
    struct check temp = *(struct check *)comp;
    printf(" '%d:%c:%f", temp.i, temp.c, temp.d);
}
```

How do you reverse a linked list without using any C pointers?

One way is to reverse the data in the nodes without changing the pointers themselves. One can also create a new linked list which is the reverse of the original linked list. A simple C program can do that for you. Please note that you would still use the "next" pointer fields to traverse through the linked list (So in effect, you are using the pointers, but you are not changing them when reversing the linked list).

How would you detect a loop in a linked list? Write a C program to detect a loop in a linked list.

This is also one of the classic interview questions

There are multiple answers to this problem. Here are a few C programs to attack this problem.

Brute force method

Have a double loop, where you check the node pointed to by the outer loop, with every node of the inner loop.

```
typedef struct node
{
  void *data;
  struct node *next;
}mynode;
```

```
  mynode * find_loop(NODE * head)
{
  mynode *current = head;

  while(current->next != NULL)
  {
    mynode *temp = head;
    while(temp->next != NULL && temp != current)
    {
      if(current->next == temp)
      {
        printf("\nFound a loop.");
        return current;
      }
      temp = temp->next;
    }
    current = current->next;
  }
  return NULL;
}
```

Visited flag

Have a visited flag in each node of the linked list. Flag it as visited when you reach the node. When you reach a node and the flag is already flagged as visited, then you know there is a loop in the linked list.

Fastest method

Have 2 pointers to start of the linked list. Increment one pointer by 1 node and the other by 2 nodes. If there's a loop, the 2nd pointer will meet the 1st pointer somewhere. If it does, then you know there's one.

Here is some code

```
p=head;
q=head->next;

while(p!=NULL && q!=NULL)
{
  if(p==q)
  {
    //Loop detected!
    exit(0);
  }
  p=p->next;
  q=(q->next)?(q->next->next):q->next;
}

// No loop.
```

How do you find the middle of a linked list? Write a C program to return the middle of a linked list?

Another popular interview question

Here are a few C program snippets to give you an idea of the possible solutions.

Method1 (Uses one slow pointer and one fast pointer)

```
#include<stdio.h>
#include<ctype.h>

typedef struct node
{
  int value;
  struct node *next;
  struct node *prev;
}mynode ;



void add_node(struct node **head, int value);
void print_list(char *listName, struct node *head);
void getTheMiddle(mynode *head);

// The main function..
int main()
{
   mynode *head;
   head = (struct node *)NULL;

   add_node(&head, 1);
   add_node(&head, 10);
   add_node(&head, 5);
   add_node(&head, 70);
   add_node(&head, 9);
   add_node(&head, -99);
   add_node(&head, 0);
   add_node(&head, 555);
   add_node(&head, 55);

   print_list("myList", head);
   getTheMiddle(head);

   getch();
   return(0);
}


// This function uses one slow and one fast
// pointer to get to the middle of the LL.
//
// The slow pointer is advanced only by one node
// and the fast pointer is advanced by two nodes!
```

```
void getTheMiddle(mynode *head)
{
  mynode *p = head;
  mynode *q = head;

  if(q!=NULL)
  {
       while((q->next)!=NULL && (q->next->next)!=NULL)
       {
          p=(p!=(mynode *)NULL?p->next:(mynode *)NULL);
          q=(q!=(mynode *)NULL?q->next:(mynode *)NULL);
          q=(q!=(mynode *)NULL?q->next:(mynode *)NULL);
       }
       printf("The middle element is [%d]",p->value);
  }
}



// Function to add a node
void add_node(struct node **head, int value)
{
  mynode *temp, *cur;
  temp = (mynode *)malloc(sizeof(mynode));
  temp->next=NULL;
  temp->prev=NULL;

  if(*head == NULL)
  {
     *head=temp;
     temp->value=value;
  }
  else
  {
   for(cur=*head;cur->next!=NULL;cur=cur->next);
   cur->next=temp;
   temp->prev=cur;
   temp->value=value;
  }
}



// Function to print the linked list...
void print_list(char *listName, struct node *head)
{
  mynode *temp;

  printf("\n[%s] -> ", listName);
  for(temp=head;temp!=NULL;temp=temp->next)
  {
    printf("[%d]->",temp->value);
  }

  printf("NULL\n");

}
```

Here p moves one step, where as q moves two steps, when q reaches end, p will be at the middle of the linked list.

Method2(Uses a counter)

```
#include<stdio.h>
#include<ctype.h>

typedef struct node
{
  int value;
  struct node *next;
  struct node *prev;
}mynode ;

void add_node(struct node **head, int value);
void print_list(char *listName, struct node *head);
mynode *getTheMiddle(mynode *head);

// The main function..
int main()
{
   mynode *head, *middle;
   head = (struct node *)NULL;
   add_node(&head, 1);
   add_node(&head, 10);
   add_node(&head, 5);
   add_node(&head, 70);
   add_node(&head, 9);
   add_node(&head, -99);
   add_node(&head, 0);
   add_node(&head, 555);
   add_node(&head, 55);

   print_list("myList", head);
   middle = getTheMiddle(head);
   printf("\nMiddle node -> [%d]\n\n", middle->value);

   getch();
   return(0);
}

// Function to get to the middle of the LL
mynode *getTheMiddle(mynode *head)
{
  mynode *middle = (mynode *)NULL;
  int i;
  for(i=1; head!=(mynode *)NULL; head=head->next,i++)
  {
     if(i==1)
        middle=head;
     else if ((i%2)==1)
        middle=middle->next;
   }
      return middle;
}
```

```
// Function to add a new node to the LL
void add_node(struct node **head, int value)
{
  mynode *temp, *cur;
  temp = (mynode *)malloc(sizeof(mynode));
  temp->next=NULL;
  temp->prev=NULL;

  if(*head == NULL)
  {
     *head=temp;
     temp->value=value;
  }
  else
  {
   for(cur=*head;cur->next!=NULL;cur=cur->next);
   cur->next=temp;
   temp->prev=cur;
   temp->value=value;
  }
}


// Function to print the LL
void print_list(char *listName, struct node *head)
{
  mynode *temp;

  printf("\n[%s] -> ", listName);
  for(temp=head;temp!=NULL;temp=temp->next)
  {
    printf("[%d]->",temp->value);
  }

  printf("NULL\n");

}
```

In a similar way, we can find the 1/3 th node of linked list by changing (i%2==1) to (i%3==1) and in the same way we can find nth node of list by changing (i%2==1) to (i%n==1) but make sure ur (n<=i).

If you are using C language to implement the heterogeneous linked list, what pointer type will you use?

The heterogeneous linked list contains different data types in its nodes and we need a link, pointer to connect them. It is not possible to use ordinary pointers for this. So we go for void pointer. Void pointer is capable of storing pointer to any type as it is a generic pointer type.
Check out the C program to implement a Generic linked list in the same FAQ.

How to compare two linked lists? Write a C program to compare two linked lists.

Here is a simple C program to accomplish the same.

```
int compare_linked_lists(struct node *q, struct node *r)
{
    static int flag;

    if((q==NULL ) && (r==NULL))
    {
         flag=1;
    }
    else
    {
        if(q==NULL || r==NULL)
        {
            flag=0;
        }
        if(q->data!=r->data)
        {
            flag=0;
        }
        else
        {
           compare_linked_lists(q->link,r->link);
        }
    }
    return(flag);
}
```

Another way is to do it on similar lines as strcmp() compares two strings, character by character (here each node is like a character).

How to create a copy of a linked list? Write a C program to create a copy of a linked list.

Check out this C program which creates an exact copy of a linked list.

```
copy_linked_lists(struct node *q, struct node **s)
{
    if(q!=NULL)
    {
        *s=malloc(sizeof(struct node));
        (*s)->data=q->data;
        (*s)->link=NULL;
        copy_linked_list(q->link, &((*s)->link));
    }
}
```

Write a C program to free the nodes of a linked list.

Before looking at the answer, try writing a simple C program (with a for loop) to do this. Quite a few people get this wrong.

This is the wrong way to do it

```
struct list *listptr, *nextptr;
for(listptr = head; listptr != NULL; listptr = listptr->next)
{
  free(listptr);
}
```

If you are thinking why the above piece of code is wrong, note that once you free the listptr node, you cannot do something like listptr = listptr->next!. Since listptr is already freed, using it to get listptr->next is illegal and can cause unpredictable results!

This is the right way to do it

```
struct list *listptr, *nextptr;
for(listptr = head; listptr != NULL; listptr = nextptr)
{
  nextptr = listptr->next;
  free(listptr);
}
head = NULL;
```

After doing this, make sure you also set the head pointer to NULL!

Can we do a Binary search on a linked list?

Great C datastructure question!

The answer is ofcourse, you can write a C program to do this. But, the question is, do you really think it will be as efficient as a C program which does a binary search on an array?

Think hard, real hard.

Do you know what exactly makes the binary search on an array so fast and efficient? Its the ability to access any element in the array in constant time. This is what makes it so fast. You can get to the middle of the array just by saying array[middle]!. Now, can you do the same with a linked list? The answer is No. You will have to write your own, possibly inefficient algorithm to get the value of the middle node of a linked list. In a linked list, you loosse the ability to get the value of any node in a constant time.

One solution to the inefficiency of getting the middle of the linked list during a binary search is to have the first node contain one additional pointer that points to the node in the middle. Decide at the first node if you need to check the first or the second half of the linked list. Continue doing that with each half-list.

Write a C program to return the nth node from the end of a linked list.

Here is a solution which is often called as the solution that uses frames.

Suppose one needs to get to the 6th node from the end in this LL. First, just keep on incrementing the first pointer (ptr1) till the number of increments cross n (which is 6 in this case)

```
STEP 1    :   1(ptr1,ptr2) -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -
> 10

STEP 2    :   1(ptr2) -> 2 -> 3 -> 4 -> 5 -> 6(ptr1) -> 7 -> 8 -> 9 -
> 10
```

Now, start the second pointer (ptr2) and keep on incrementing it till the first pointer (ptr1) reaches the end of the LL.

```
STEP 3    :   1 -> 2 -> 3 -> 4(ptr2) -> 5 -> 6 -> 7 -> 8 -> 9 -
> 10 (ptr1)
```

So here you have!, the 6th node from the end pointed to by ptr2!

Here is some C code..

```
struct node
{
  int data;
  struct node *next;
}mynode;


mynode * nthNode(mynode *head, int n /*pass 0 for last node*/)
{
  mynode *ptr1,*ptr2;
  int count;

  if(!head)
  {
    return(NULL);
  }

  ptr1  = head;
  ptr2  = head;
  count = 0;

  while(count < n)
  {
     count++;
     if((ptr1=ptr1->next)==NULL)
     {
        //Length of the linked list less than n. Error.
        return(NULL);
     }
  }

  while((ptr1=ptr1->next)!=NULL)
  {
    ptr2=ptr2->next;
  }

  return(ptr2);
}
```

How would you find out if one of the pointers in a linked list is corrupted or not?

This is a really good interview question. The reason is that linked lists are used in a wide variety of scenarios and being able to detect and correct pointer corruptions might be a very valuable tool. For example, data blocks associated with files in a file system are usually stored as linked lists. Each data block points to the next data block. A single corrupt pointer can cause the entire file to be lost!

- Discover and fix bugs when they corrupt the linked list and not when effect becomes visible in some other part of the program. Perform frequent consistency checks (to see if the linked list is indeed holding the data that you inserted into it).
- It is good programming practice to set the pointer value to NULL immediately after freeing the memory pointed at by the pointer. This will help in debugging, because it will tell you that the object was freed somewhere beforehand. Keep track of how many objects are pointing to a object using reference counts if required.
- Use a good debugger to see how the datastructures are getting corrupted and trace down the problem. Debuggers like ddd on linux and memory profilers like Purify, Electric fence are good starting points. These tools should help you track down heap corruption issues easily.
- Avoid global variables when traversing and manipulating linked lists. Imagine what would happen if a function which is only supposed to traverse a linked list using a global head pointer accidently sets the head pointer to NULL!.
- Its a good idea to check the addNode() and the deleteNode() routines and test them for all types of scenarios. This should include tests for inserting/deleting nodes at the front/middle/end of the linked list, working with an empty linked list,

running out of memory when using malloc() when allocating memory for new nodes, writing through NULL pointers, writing more data into the node fields then they can hold (resulting in corrupting the (probably adjacent) "prev" and "next" pointer fields), make sure bug fixes and enhancements to the linked list code are reviewed and well tested (a lot of bugs come from quick and dirty bug fixing), log and handle all possible errors (this will help you a lot while debugging), add multiple levels of logging so that you can dig through the logs. The list is endless...

- Each node can have an extra field associated with it. This field indicates the number of nodes after this node in the linked list. This extra field needs to be kept up-to-date when we inserte or delete nodes in the linked list (It might become slightly complicated when insertion or deletion happens not at end, but anywhere in the linked list). Then, if for any node, p->field > 0 and p->next == NULL, it surely points to a pointer corruption.
- You could also keep the count of the total number of nodes in a linked list and use it to check if the list is indeed having those many nodes or not.

The problem in detecting such pointer corruptions in C is that its only the programmer who knows that the pointer is corrupted. The program has no way of knowing that something is wrong. So the best way to fix these errors is check your logic and test your code to the maximum possible extent. I am not aware of ways in C to recover the lost nodes of a corrupted linked list. C does not track pointers so there is no good way to know if an arbitrary pointer has been corrupted or not. The platform may have a library service that checks if a pointer points to valid memory (for instance on Win32 there is a IsBadReadPtr, IsBadWritePtr API.) If you detect a cycle in the link list, it's definitely bad. If it's a doubly linked list you can verify, pNode->Next->Prev == pNode.

I have a hunch that interviewers who ask this question are probably hinting at something called Smart Pointers in C++. Smart pointers are particularly useful in the face of exceptions as they ensure proper destruction of dynamically allocated objects. They can also be used to keep track of dynamically allocated objects shared by multiple owners. This topic is out of scope here, but you can find lots of material on the Internet for Smart Pointers.

Write a C program to insert nodes into a linked list in a sorted fashion

The solution is to iterate down the list looking for the correct place to insert the new node. That could be the end of the list, or a point just before a node which is larger than the new node.

Note that we assume the memory for the new node has already been allocated and a pointer to that memory is being passed to this function.

```
// Special case code for the head end
void linkedListInsertSorted(struct node** headReference, struct node* n
ewNode)
{
  // Special case for the head end
  if (*headReference == NULL || (*headReference)->data >= newNode-
>data)
  {
     newNode->next = *headReference;
     *headReference = newNode;
  }
  else
  {
     // Locate the node before which the insertion is to happen!
     struct node* current = *headReference;
     while (current->next!=NULL && current->next->data < newNode-
>data)
     {
        current = current->next;
     }
     newNode->next = current->next;
     current->next = newNode;
   }
}
```

Write a C program to remove duplicates from a sorted linked list.

As the linked list is sorted, we can start from the beginning of the list and compare adjacent nodes. When adjacent nodes are the same, remove the second one. There's a tricky case where the node after the next node needs to be noted before the deletion.

```
// Remove duplicates from a sorted list
void RemoveDuplicates(struct node* head)
{
  struct node* current = head;
  if (current == NULL) return; // do nothing if the list is empty

  // Compare current node with next node
  while(current->next!=NULL)
  {
      if (current->data == current->next->data)
      {
         struct node* nextNext = current->next->next;
         free(current->next);
         current->next = nextNext;
      }
      else
      {
         current = current->next; // only advance if no deletion
      }
   }
}
```

How to read a singly linked list backwards?

Use Recursion.

How can I search for data in a linked list?

The only way to search a linked list is with a linear search, because the only way a linked list's members can be accessed is sequentially. Sometimes it is quicker to take the data from a linked list and store it in a different data structure so that searches can be more efficient.

Write your own....

Write your own C program to implement the atoi() function.

The prototype of the atoi() function is ...

```
int atoi(const char *string);
```

Here is a C program which explains a different way of coding the atoi() function in the C language.

```
#include<stdio.h>

int myatoi(const char *string);

int main(int argc, char* argv[])
{
  printf("\n%d\n", myatoi("1998"));
  getch();
  return(0);
}

int myatoi(const char *string)
{
    int i;
    i=0;
    while(*string)
    {
        i=(i<<3) + (i<<1) + (*string - '0');
        string++;

        // Dont increment i!

    }
    return(i);
}
```

Try working it out with a small string like "1998", you will find out it does work!.

Ofcourse, there is also the trivial method ....

```
"1998" == 8 + (10 * 9) + (100 * 9) + (1 * 1000) = 1998
```

This can be done either by going from right to left or left to right in the string

One solution is given below

```
int myatoi(const char* string)
{
  int value = 0;

  if (string)
  {
    while (*string && (*string <= '9' && *string >= '0'))
    {
      value = (value * 10) + (*string - '0');
      string++;
    }
  }
  return value;
}
```

Note that these functions have no error handling incorporated in them (what happens if someone passes non-numeric data (say "1A998"), or negative numeric strings (say "-1998")). I leave it up to you to add these cases. The essense is to understand the core logic first.

Implement the memmove() function. What is the difference between the memmove() and memcpy() function?

One more most frequently asked interview question!.

memmove() offers guaranteed behavior if the source and destination arguments overlap. memcpy() makes no such guarantee, and may therefore be more efficient to implement. It's always safer to use memmove().

Note that the prototype of memmove() is ...

```
void *memmove(void *dest, const void *src, size_t count);
```

Here is an implementation..

```
#include <stdio.h>
#include <string.h>


void *mymemmove(void *dest, const void *src, size_t count);


int main(int argc, char* argv[])
{
  char *p1, *p2;
  char *p3, *p4;
  int  size;
```

```
printf("\n--------------------------------\n");

/* ----------------------------------------
 *
 * CASE 1 : From (SRC) < To (DEST)
 *
 *      +--+---------------------+--+
 *      |  |                     |  |
 *      +--+---------------------+--+
 *      ^  ^
 *      |  |
 *    From To
 *
 * --------------------------------------- */

p1 = (char *) malloc(12);
memset(p1,12,'\0');
size=10;

strcpy(p1,"ABCDEFGHI");

p2 = p1 + 2;

printf("\n--------------------------------\n");
printf("\nFrom (before) = [%s]",p1);
printf("\nTo (before)   = [%s]",p2);

mymemmove(p2,p1,size);

printf("\n\nFrom (after) = [%s]",p1);
printf("\nTo (after)   = [%s]",p2);

printf("\n--------------------------------\n");



/* ----------------------------------------
 *
 * CASE 2 : From (SRC) > To (DEST)
 *
 *      +--+---------------------+--+
 *      |  |                     |  |
 *      +--+---------------------+--+
 *      ^  ^
 *      |  |
 *     To From
 *
 * --------------------------------------- */


p3 = (char *) malloc(12);
memset(p3,12,'\0');
p4 = p3 + 2;

strcpy(p4, "ABCDEFGHI");

printf("\nFrom (before) = [%s]",p4);
```

```
  printf("\nTo (before)   = [%s]",p3);

  mymemmove(p3, p4, size);

  printf("\n\nFrom (after) = [%s]",p4);
  printf("\nTo (after)   = [%s]",p3);

  printf("\n--------------------------------\n");


  /* ----------------------------------------
   *
   * CASE 3 : No overlap
   *
   * --------------------------------------- */

  p1 = (char *) malloc(30);
  memset(p1,30,'\0');
  size=10;

  strcpy(p1,"ABCDEFGHI");

  p2 = p1 + 15;

  printf("\n--------------------------------\n");
  printf("\nFrom (before) = [%s]",p1);
  printf("\nTo (before)   = [%s]",p2);

  mymemmove(p2,p1,size);

  printf("\n\nFrom (after) = [%s]",p1);
  printf("\nTo (after)   = [%s]",p2);

  printf("\n--------------------------------\n");

  printf("\n\n");

  return 0;
}




void *mymemmove(void *to, const void *from, size_t size)
{
    unsigned char *p1;
    const unsigned char *p2;

    p1 = (unsigned char *) to;
    p2 = (const unsigned char *) from;

    p2 = p2 + size;

    // Check if there is an overlap or not.
    while (p2 != from && --p2 != to);
    if (p2 != from)
    {
        // Overlap detected!

        p2  = (const unsigned char *) from;
        p2  = p2 + size;
        p1  = p1 + size;

        while (size-- != 0)
        {
            *--p1 = *--p2;
        }
    }
    else
    {
        // No overlap OR they overlap as CASE 2 above.
        // memcopy() would have done this directly.

        while (size-- != 0)
        {
            *p1++ = *p2++;
        }
    }
    return(to);
}
```

And here is the output

```
--------------------------------

From (before) = [ABCDEFGHI]
To   (before) = [CDEFGHI]


From (after) = [ABABCDEFGHI]
To   (after) = [ABCDEFGHI]

--------------------------------

From (before) = [ABCDEFGHI]
To   (before) = [α╙ABCDEFGHI]


From (after) = [CDEFGHI]
To   (after) = [ABCDEFGHI]

--------------------------------

From (before) = [ABCDEFGHI]
To   (before) = [FEδ‼&:F]


From (after) = [ABCDEFGHI]
To (after)   = [ABCDEFGHI]

--------------------------------
```

So then, whats the difference between the implementation of memmove() and memcpy(). Its just that memcpy() will not care if the memories overlap and will either copy from left to right or right to left without checking which method to used depending on the type of the overlap. Also note that the C code proves that the results are the same irrespective of the Endian-ness of the machine.

Write C code to implement the strstr() (search for a substring) function.

This is also one of the most frequently asked interview questions. Its asked almost 99% of the times. Here are a few C programs to implement your own strstr() function.

There are a number of ways to find a string inside another string. Its important to be aware of these algorithms than to memorize them. Some of the fastest algorithms are quite tough to understand!.

Method1

The first method is the classic Brute force method. The Brute Force algorithm checks, at all positions in the text between 0 and (n-m), if an occurrence of the pattern starts at that position or not. Then, after each successfull or unsuccessful attempt, it shifts the pattern exactly one position to the right. The time complexity of this searching phase is O(mn). The expected number of text character comparisons is 2n.

Here 'n' is the size of the string in which the substring of size 'm' is being searched for.

Here is some code (which works!)

```
#include<stdio.h>

void BruteForce(char *x /* pattern */,
                int m   /* length of the pattern */,
                char *y /* actual string being searched */,
                int n   /* length of this string */)
{
   int i, j;
   printf("\nstring    : [%s]"
          "\nlength    : [%d]"
          "\npattern   : [%s]"
          "\nlength    : [%d]\n\n", y,n,x,m);


   /* Searching */
   for (j = 0; j <= (n - m); ++j)
   {
      for (i = 0; i < m && x[i] == y[i + j]; ++i);
        if (i >= m) {printf("\nMatch found at\n\n->[%d]\n-
>[%s]\n",j,y+j);}
   }
}
```

```
int main()
{
  char *string  = "hereroheroero";
  char *pattern = "hero";

  BF(pattern,strlen(pattern),string,strlen(string));
  printf("\n\n");
  return(0);
}
```

This is how the comparison happens visually

```
hereroheroero
   !
hero


hereroheroero
 !
 hero


hereroheroero
  !
  hero


hereroheroero
   !
   hero


hereroheroero
    !
    hero


hereroheroero
     !
     hero


hereroheroero
      |||| ----> Match!
      hero


hereroheroero
       !
       hero
```

```
hereroheroero
        !
        hero
```

```
hereroheroero
         !
         hero
```

Method2

The second method is called the Rabin-Karp method.

Instead of checking at each position of the text if the pattern occurs or not, it is better to check first if the contents of the current string "window" looks like the pattern or not. In order to check the resemblance between these two patterns, a hashing function is used. Hashing a string involves computing a numerical value from the value of its characters using a hash function.

The Rabin-Karp method uses the rule that if two strings are equal, their hash values must also be equal. Note that the converse of this statement is not always true, but a good hash function tries to reduce the number of such hash collisions. Rabin-Karp computes hash value of the pattern, and then goes through the string computing hash values of all of its substrings and checking if the pattern's hash value is equal to the substring hash value, and advancing by 1 character every time. If the two hash values are the same, then the algorithm verifies if the two string really are equal, rather than this being a fluke of the hashing scheme. It uses regular string comparison for this final check. Rabin-Karp is an algorithm of choice for multiple pattern search. If we want to find any of a large number, say k, fixed length patterns in a text, a variant Rabin-Karp that uses a hash table to check whether the hash of a given string belongs to a set of hash values of patterns we are looking for. Other algorithms can search for a single pattern in time order O(n), hence they will search for k patterns in time order O(n*k). The variant Rabin-Karp will still work in time order O(n) in the best and average case because a hash table allows to check whether or not substring hash equals any of the pattern hashes in time order of O(1).

Here is some code (not working though!)

```
#include<stdio.h>

hashing_function()
{
  // A hashing function to compute the hash values of the strings.
  ....
}
```

```
void KarpRabinR(char *x, int m, char *y, int n)
{
   int hx, hy, i, j;

   printf("\nstring    : [%s]"
          "\nlength    : [%d]"
          "\npattern   : [%s]"
          "\nlength    : [%d]\n\n", y,n,x,m);


   /* Preprocessing  phase */
   Do preprocessing here..

   /* Searching */
   j = 0;
   while (j <= n-m)
   {
      if (hx == hy && memcmp(x, y + j, m) == 0)
      {
         // Hashes match and so do the actual strings!
         printf("\nMatch found at : [%d]\n",j);
      }

      hy = hashing_function(y[j], y[j + m], hy);
      ++j;
   }
}


int main()
{

  char *string="hereroheroero";
  char *pattern="hero";

  KarpRabin(pattern,strlen(pattern),string,strlen(string));

  printf("\n\n");
  return(0);

}
```

This is how the comparison happens visually

```
hereroheroero
   !
hero


hereroheroero
 !
 hero
```

```
hereroheroero
  !
  hero


hereroheroero
   !
   hero


hereroheroero
    !
    hero


hereroheroero
     !
     hero


hereroheroero
      |||| ----> Hash values match, so do the strings!
      hero


hereroheroero
       !
       hero


hereroheroero
        !
        hero


hereroheroero
         !
         hero
```

Method3

The Knuth-Morris-Pratt or the Morris-Pratt algorithms are extensions of the basic Brute Force algorithm. They use precomputed data to skip forward not by 1 character, but by as many as possible for the search to succeed.

Here is some code

```
void preComputeData(char *x, int m, int Next[])
{
   int i, j;
   i = 0;
   j = Next[0] = -1;
   while (i < m)
   {
      while (j > -1 && x[i] != x[j])
         j = Next[j];
      Next[++i] = ++j;

   }
}


void MorrisPrat(char *x, int m, char *y, int n)
{
   int i, j, Next[1000];

   /* Preprocessing */
   preComputeData(x, m, Next);

   /* Searching */
   i = j = 0;
   while (j < n)
   {
      while (i > -1 && x[i] != y[j])
         i = Next[i];
      i++;
      j++;
      if (i >= m)
      {
         printf("\nMatch found at : [%d]\n",j - i);
         i = Next[i];
      }
   }
}


int main()
{
  char *string="hereroheroero";
  char *pattern="hero";

  MorrisPrat(pattern,strlen(pattern),string,strlen(string));

  printf("\n\n");
  return(0);
}
```

This is how the comparison happens visually

```
hereroheroero
   !
hero


hereroheroero
   !
   hero
```

```
hereroheroero
    !
    hero
```

```
hereroheroero
     !
     hero
```

```
hereroheroero
      |||| ----> Match found!
      hero
```

```
hereroheroero
          !
          hero
```

Method4

The Boyer Moore algorithm is the fastest string searching algorithm. Most editors use this algorithm.

It compares the pattern with the actual string from right to left. Most other algorithms compare from left to right. If the character that is compared with the rightmost pattern symbol does not occur in the pattern at all, then the pattern can be shifted by m positions behind this text symbol.

The following example illustrates this situation.

Example:

```
0 1 2 3 4 5 6 7 8 9 ...
a b b a d a b a c b a
        | |
b a b a c |
  <------ |
          |
          b a b a c
```

The comparison of "d" with "c" at position 4 does not match. "d" does not occur in the pattern. Therefore, the pattern cannot match at any of the positions 0,1,2,3,4, since all corresponding windows contain a "d". The pattern can be shifted to position 5. The best case for the Boyer-Moore algorithm happens if, at each search attempt the first compared character does not occur in the pattern. Then the algorithm requires only O(n/m) comparisons .

Bad character heuristics

This method is called bad character heuristics. It can also be applied if the bad character (the character that causes a mismatch), occurs somewhere else in the pattern. Then the pattern can be shifted so that it is aligned to this text symbol. The next example illustrates this situation.

Example:

```
0 1 2 3 4 5 6 7 8 9 ...
a b b a b a b a c b a
        |
b a b a c
    <----
    |
    b a b a c
```

Comparison between "b" and "c" causes a mismatch. The character "b" occurs in the pattern at positions 0 and 2. The pattern can be shifted so that the rightmost "b" in the pattern is aligned to "b".

Good suffix heuristics

Sometimes the bad character heuristics fails. In the following situation the comparison between "a" and "b" causes a mismatch. An alignment of the rightmost occurence of the pattern symbol a with the text symbol a would produce a negative shift. Instead, a shift by 1 would be possible. However, in this case it is better to derive the maximum possible shift distance from the structure of the pattern. This method is called good suffix heuristics.

Example:

```
0 1 2 3 4 5 6 7 8 9 ...
a b a a b a b a c b a
    | | |
c a b a b
    <----
    | | |
    c a b a b
```

The suffix "ab" has matched. The pattern can be shifted until the next occurence of ab in the pattern is aligned to the text symbols ab, i.e. to position 2.

In the following situation the suffix "ab" has matched. There is no other occurence of "ab" in the pattern.Therefore, the pattern can be shifted behind "ab", i.e. to position 5.

Example:

```
0 1 2 3 4 5 6 7 8 9 ...
a b c a b a b a c b a
    | | |
c b a a b
          c b a a b
```

In the following situation the suffix "bab" has matched. There is no other occurence of "bab" in the pattern. But in this case the pattern cannot be shifted to position 5 as before, but only to position 3, since a prefix of the pattern "ab" matches the end of "bab". We refer to this situation as case 2 of the good suffix heuristics.

Example:

```
0 1 2 3 4 5 6 7 8 9 ...
a a b a b a b a c b a
  | | | |
a b b a b
      a b b a b
```

The pattern is shifted by the longest of the two distances that are given by the bad character and the good suffix heuristics.

The Boyer-Moore algorithm uses two different heuristics for determining the maximum possible shift distance in case of a mismatch: the "bad character" and the "good suffix" heuristics. Both heuristics can lead to a shift distance of m. For the bad character heuristics this is the case, if the first comparison causes a mismatch and the corresponding text symbol does not occur in the pattern at all. For the good suffix heuristics this is the case, if only the first comparison was a match, but that symbol does not occur elsewhere in the pattern.

A lot of these algorithms have been explained here with good visualizations. Remember, again that its sufficient to know the basic Brute force algorithm and be aware of the other methods. No one expects you to know every possible algorithm on earth.

Write your own printf() function in C?

This is again one of the most frequently asked interview questions. Here is a C program which implements a basic version of printf(). This is a really, really simplified version of printf(). Note carefully how floating point and other compilcated support has been left out. Also, note how we use low level puts() and putchar(). Dont make a fool of yourself by using printf() within the implementation of printf()!

```
#include<stdio.h>
#include<stdarg.h>
```

```
main()
{
    void myprintf(char *,...);
    char * convert(unsigned int, int);
    int i=65;
    char str[]="This is my string";
    myprintf("\nMessage = %s%d%x",str,i,i);
}

void myprintf(char * frmt,...)
{

    char *p;
    int i;
    unsigned u;
    char *s;
    va_list argp;


    va_start(argp, fmt);

    p=fmt;
    for(p=fmt; *p!='\0';p++)
    {
        if(*p=='%')
        {
            putchar(*p);continue;
        }

        p++;

        switch(*p)
        {
            case 'c' : i=va_arg(argp,int);putchar(i);break;
            case 'd' : i=va_arg(argp,int);
                                if(i<0){i=-i;putchar('-
');}puts(convert(i,10));break;
            case 'o': i=va_arg(argp,unsigned int); puts(convert(i,8));b
reak;
            case 's': s=va_arg(argp,char *); puts(s); break;
            case 'u': u=va_arg(argp,argp, unsigned int); puts(convert(u
,10));break;
            case 'x': u=va_arg(argp,argp, unsigned int); puts(convert(u
,16));break;
            case '%': putchar('%');break;
        }
    }

    va_end(argp);
}
```

```
char *convert(unsigned int, int)
{
    static char buf[33];
    char *ptr;

    ptr=&buf[sizeof(buff)-1];
    *ptr='\0';
    do
    {
        *--ptr="0123456789abcdef"[num%base];
        num/=base;
    }while(num!=0);
    return(ptr);
}
```

Implement the strcpy() function.

Here are some C programs which implement the strcpy() function. This is one of the most frequently asked C interview questions.

Method1

```
char *mystrcpy(char *dst, const char *src)
{
  char *ptr;
  ptr = dst;
  while(*dst++=*src++);
  return(ptr);
}
```

The strcpy function copies src, including the terminating null character, to the location specified by dst. No overflow checking is performed when strings are copied or appended. The behavior of strcpy is undefined if the source and destination strings overlap. It returns the destination string. No return value is reserved to indicate an error.

Note that the prototype of strcpy as per the C standards is

```
char *strcpy(char *dst, const char *src);
```

Notice the const for the source, which signifies that the function must not change the source string in anyway!.

Method2

```
char *my_strcpy(char dest[], const char source[])
{
  int i = 0;
  while (source[i] != '\0')
  {
    dest[i] = source[i];
    i++;
  }
  dest[i] = '\0';
  return(dest);
}
```

Implement the strcmp(str1, str2) function.

There are many ways one can implement the strcmp() function. Note that strcmp(str1,str2) returns a -ve number if str1 is alphabetically above str2, 0 if both are equal and +ve if str2 is alphabetically above str1.

Here are some C programs which implement the strcmp() function. This is also one of the most frequently asked interview questions. The prototype of strcmp() is

```
int strcmp( const char *string1, const char *string2 );
```

Here is some C code..

```
#include <stdio.h>

int mystrcmp(const char *s1, const char *s2);

int main()
{
  printf("\nstrcmp() = [%d]\n", mystrcmp("A","A"));
  printf("\nstrcmp() = [%d]\n", mystrcmp("A","B"));
  printf("\nstrcmp() = [%d]\n", mystrcmp("B","A"));
  return(0);
}

int mystrcmp(const char *s1, const char *s2)
{
    while (*s1==*s2)
    {
        if(*s1=='\0')
           return(0);
        s1++;
        s2++;
    }
    return(*s1-*s2);
}
```

And here is the output...

```
strcmp() = [0]
strcmp() = [-1]
strcmp() = [1]
```

Implement the substr() function in C.

Here is a C program which implements the substr() function in C.

```
int main()
{
  char str1[] = "India";
  char str2[25];

  substr(str2, str1, 1, 3);
  printf("\nstr2 : [%s]", str2);
  return(0);
}

substr(char *dest, char *src, int position, int length)
{
    dest[0]='\0';
    strncat(dest, (src + position), length);
}
```

Here is another C program to do the same...

```
#include <stdio.h>
#include <conio.h>

void mySubstr(char *dest, char *src, int position, int length);

int main()
{
 char subStr[100];
 char str[]="My Name Is Sweet";

 mySubstr(subStr, str, 1, 5);
 printf("\nstr    = [%s]"
        "\nsubStr = [%s]\n\n",
        str, subStr);
 getch();
 return(0);
}

void mySubstr(char *dest, char *src, int position, int length)
{
  while(length > 0)
  {
    *dest = *(src+position);
    dest++;
    src++;
    length--;
  }
}
```

Write your own copy() function.

Here is some C code that simulates a file copy action.

```
#include <stdio.h>              /* standard I/O routines. */
#define MAX_LINE_LEN 1000 /* maximum line length supported. */


void main(int argc, char* argv[])
{
    char* file_path_from;
    char* file_path_to;
    FILE* f_from;
    FILE* f_to;
    char buf[MAX_LINE_LEN+1];

    file_path_from = "<something>";
    file_path_to   = "<something_else>";

    f_from = fopen(file_path_from, "r");
    if (!f_from) {exit(1);}

    f_to = fopen(file_path_to, "w+");
    if (!f_to) {exit(1);}

    /* Copy source to target, line by line. */
    while (fgets(buf, MAX_LINE_LEN+1, f_from))
    {
        if (fputs(buf, f_to) == EOF){exit(1);}
    }

    if (!feof(f_from)){exit(1);}

    if (fclose(f_from) == EOF) {exit(1);}
    if (fclose(f_to) == EOF)   {exit(1);}

    return(0);
}
```

Write C programs to implement the toupper() and the isupper() functions.

toUpper()

```
int toUpper(int ch)
{
    if(ch>='a' && c<='z')
        return('A' + ch - 'a');
    else
        return(ch);
}
```

isUpper()

```
int isUpper(int ch)
{
    if(ch>='A' && ch <='Z')
       return(1); //Yes, its upper!
    else
       return(0); // No, its lower!
}
```

Its important to know that the upper and lower case alphabets have corresponding integer values.

```
A-Z - 65-90
a-z - 97-122
```

Another way to do this conversion is to maintain a correspondance between the upper and lower case alphabets. The program below does that. This frees us from the fact that these alphabets have a corresponding integer values. I dont know what one should do for non-english alphabets. Do other languages have upper and lower case letters in the first place :) !

```
#include <string.h>

#define UPPER   "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
#define LOWER   "abcdefghijklmnopqrstuvwxyz"

int toUpper(int c)
{
    const char *upper;
    const char *const lower = LOWER;

    // Get the position of the lower case alphabet in the LOWER string
using the strchr() function ..
    upper = ( ((CHAR_MAX >= c)&&(c > '\0')) ? strchr(lower, c) : NULL);


    // Now return the corresponding alphabet at that position in the UP
PER string ..
    return((upper != NULL)?UPPER[upper - lower] : c);
}
```

Note that these routines dont have much error handling incorporated in them. Its really easy to add error handling to these routines or just leave it out (as I like it). This site consciously leaves out error handling for most of the programs to prevent unwanted clutter and present the core logic first.

Write a C program to implement your own strdup() function.

Here is a C program to implement the strdup() function.

```
char *mystrdup(char *s)
{
    char *result = (char*)malloc(strlen(s) + 1);
    if (result == (char*)0){return (char*)0;}
    strcpy(result, s);
    return result;
}
```

Write a C program to implement the strlen() function.

The prototype of the strlen() function is...

```
size_t strlen(const char *string);
```

Here is some C code which implements the strlen() function....

```
int my_strlen(char *string)
{
  int length;
  for (length = 0; *string != '\0', string++)
  {
    length++;
  }
  return(length);
}
```

Also, see another example

```
int my_strlen(char *s)
{
  char *p=s;

  while(*p!='\0')
    p++;

  return(p-s);
}
```

Write your own strcat() function.

Here is a C function which implements the strcat() function...

```c
/* Function to concatenate string t to end of s; return s */
char *myStrcat(char *s, const char *t)
{
    char *p = s;

    if (s == NULL || t == NULL)
        return s;   /* we need not have to do anything */

    while (*s)
        s++;

    while (*s++ = *t++)
        ;

    return p;
}
```

Programs

Write a C program to swap two variables without using a temporary variable.

This questions is asked almost always in every interview.

The best way to swap two variables is to use a temporary variable.

```
int a,b,t;

t = a;
a = b;
b = t;
```

There is no way better than this as you will find out soon. There are a few slick expressions that do swap variables without using temporary storage. But they come with their own set of problems.

Method1 (The XOR trick)

```
a ^= b ^= a ^= b;
```

Although the code above works fine for most of the cases, it tries to modify variable 'a' two times between sequence points, so the behavior is undefined. What this means is it wont work in all the cases. This will also not work for floating-point values. Also, think of a scenario where you have written your code like this

```
swap(int *a, int *b)
{
  *a ^= *b ^= *a ^= *b;
}
```

Now, if suppose, by mistake, your code passes the pointer to the same variable to this function. Guess what happens? Since Xor'ing an element with itself sets the variable to zero, this routine will end up setting the variable to zero (ideally it should have swapped the variable with itself). This scenario is quite possible in sorting algorithms which sometimes try to swap a variable with itself (maybe due to some small, but not so fatal coding error). One solution to this problem is to check if the numbers to be swapped are already equal to each other.

```
swap(int *a, int *b)
{
  if(*a!=*b)
  {
    *a ^= *b ^= *a ^= *b;
  }
}
```

Method2

This method is also quite popular

```
 a=a+b;
 b=a-b;
 a=a-b;
```

But, note that here also, if a and b are big and their addition is bigger than the size of an int, even this might end up giving you wrong results.

Method3

One can also swap two variables using a macro. However, it would be required to pass the type of the variable to the macro. Also, there is an interesting problem using macros. Suppose you have a swap macro which looks something like this

```
#define swap(type,a,b) type temp;temp=a;a=b;b=temp;
```

Now, think what happens if you pass in something like this

```
swap(int,temp,a) //You have a variable called "temp" (which is quite po
ssible).
```

This is how it gets replaced by the macro

```
int temp;
temp=temp;
temp=b;
b=temp;
```

Which means it sets the value of "b" to both the variables!. It never swapped them! Scary, isn't it?

So the moral of the story is, dont try to be smart when writing code to swap variables. Use a temporary variable. Its not only fool proof, but also easier to understand and maintain.

What is the 8 queens problem? Write a C program to solve it.

The 8 queens problem is a classic problem using the chess board. This problem is to place 8 queens on the chess board so that they do not attack each other horizontally, vertically or diagonally. It turns out that there are 12 essentially distinct solutions to this problem.

Suppose we have an array t[8] which keeps track of which column is occupied in which row of the chess board. That is, if t[0]==5, then it means that the queen has been placed in the fifth column of the first row. We need to couple the backtracking algorithm with a procedure that checks whether the tuple is completable or not, i.e. to check that the next placed queen 'i' is not menaced by any of the already placed 'j' (j < i):

```
Two queens are in the same column         if t[i]=t[j]
Two queens are in the same major diagonal if (t[i]-t[j])=(i-j)
two queens are in the same minor diagonal if (t[j]-t[i])=(i-j)
```

Here is some working C code to solve this problem using backtracking

```
#include<stdio.h>
static int t[10]={-1};
void queens(int i);
int empty(int i);

void print_solution();

int main()
{
  queens(1);
  print_solution();
  return(0);
}

void queens(int i)
{
  for(t[i]=1;t[i]<=8;t[i]++)
  {
    if(empty(i))
    {
       if(i==8)
       {
          print_solution();
          /* If this exit is commented, it will show ALL possible combi
nations */
          exit(0);
       }
       else
       {
          // Recurse!
          queens(i+1);
       }

    }// if

  }// for
}


int empty(int i)
{
  int j;
  j=1;

  while(t[i]!=t[j] && abs(t[i]-t[j])!=(i-j) &&j<8)j++;

  return((i==j)?1:0);
}
```

```
void print_solution()
{
  int i;
  for(i=1;i<=8;i++)printf("\nt[%d] = [%d]",i,t[i]);
}
```

And here is one of the possible solutions

```
t[1] = [1] // This means the first square of the first row.
t[2] = [5] // This means the fifth square of the second row.
t[3] = [8] ..
t[4] = [6] ..
t[5] = [3] ..
t[6] = [7] ..
t[7] = [2] ..
t[8] = [4] // This means the fourth square of the last row.
```

Write a C program to print a square matrix helically.

Here is a C program to print a matrix helically. Printing a matrix helically means printing it in this spiral fashion

```
 >-----------+
             |
 +---->--+   |
 |       |   |
 |       |   |
 |   <---+   |
 |           |
 +-----------+
```

This is a simple program to print a matrix helically.

```
#include<stdio.h>

/* HELICAL MATRIX */

int main()
{
       int arr[][4] = { {1,2,3,4},
                        {5,6,7,8},
                        {9,10,11,12},
                        {13, 14, 15, 16}
                      };

       int i, j, k,middle,size;
       printf("\n\n");
       size = 4;
       for(i=size-1, j=0; i>0; i--, j++)
       {
              for(k=j; k<i; k++) printf("%d ", arr[j][k]);
              for(k=j; k<i; k++) printf("%d ", arr[k][i]);
              for(k=i; k>j; k--) printf("%d ", arr[i][k]);
              for(k=i; k>j; k--) printf("%d ", arr[k][j]);
       }

       middle = (size-1)/2;
       if (size % 2 == 1) printf("%d", arr[middle][middle]);
       printf("\n\n");
       return 1;
}
```

Write a C program to reverse a string.

There are a number of ways one can reverse strings. Here are a few of them. These should be enough to impress the interviewer! The methods span from recursive to non-recursive (iterative).

Also note that there is a similar question about reversing the words in a sentence, but still keeping the words in place. That is

```
I am a good boy
```

would become

```
boy good a am I
```

This is dealt with in another question. Here I only concentrate on reversing strings. That is

```
I am a good boy
```

would become

```
yob doog a ma I
```

Here are some sample C programs to do the same

Method1 (Recursive)

```
#include <stdio.h>

static char str[]="STRING TO REVERSE";

int main(int argc, char *argv)
{
   printf("\nOriginal string : [%s]", str);
   // Call the recursion function
   reverse(0);

   printf("\nReversed string : [%s]", str);
   return(0);
}

int reverse(int pos)
{
   // Here I am calculating strlen(str) everytime.
   // This can be avoided by doing this computation
   // earlier and storing it somewhere for later use.

   if(pos<(strlen(str)/2))
   {
      char ch;

      // Swap str[pos] and str[strlen(str)-pos-1]
      ch = str[pos];
      str[pos]=str[strlen(str)-pos-1];
      str[strlen(str)-pos-1]=ch;

      // Now recurse!
      reverse(pos+1);
   }
}
```

Method2

```
#include <stdio.h>
#include <malloc.h>
#include <string.h>

void ReverseStr ( char *buff, int start, int end )
{
   char tmp ;

   if ( start >= end )
   {
      printf ( "\n%s\n", buff );
      return;
   }

   tmp = *(buff + start);
   *(buff + start) = *(buff + end);
   *(buff + end) = tmp ;

   ReverseStr (buff, ++start, --end );
}

int main()
{
   char buffer[]="This is Test";
   ReverseStr(buffer,0,strlen(buffer)-1);
   return 0;
}
```

Method3

```
public static String reverse(String s)
{
    int N = s.length();
    if (N <= 1) return s;
    String left = s.substring(0, N/2);
    String right = s.substring(N/2, N);
    return reverse(right) + reverse(left);
}
```

Method4

```
for(int i = 0, j = reversed.Length - 1; i < j; i++, j--)
{
    char temp = reversed[i];
    reversed[i] = reversed[j];
    reversed[j] = temp;
}
return new String(reversed);
```

Method5

```
public static String reverse(String s)
{
   int N = s.length();
   String reverse = "";
   for (int i = 0; i < N; i++)
        reverse = s.charAt(i) + reverse;
   return reverse;
}
```

Method6

```
public static String reverse(String s)
{
   int N = s.length();
   char[] a = new char[N];
   for (int i = 0; i < N; i++)
       a[i] = s.charAt(N-i-1);
   String reverse = new String(a);
   return reverse;
}
```

Write a C program to reverse the words in a sentence in place.

That is, given a sentence like this

```
I am a good boy
```

The in place reverse would be

```
boy good a am I
```

Method1

First reverse the whole string and then individually reverse the words

```
I am a good boy
<------------->

yob doog  a   ma   I
<-> <--> <->  <-> <->

boy good a am I
```

Here is some C code to do the same ....

```
/*
  Algorithm..

  1. Reverse whole sentence first.
  2. Reverse each word individually.

  All the reversing happens in-place.
*/

#include <stdio.h>

void rev(char *l, char *r);


int main(int argc, char *argv[])
{
   char buf[] = "the world will go on forever";
   char *end, *x, *y;

   // Reverse the whole sentence first..
   for(end=buf; *end; end++);
   rev(buf,end-1);


   // Now swap each word within sentence...
   x = buf-1;
   y = buf;

   while(x++ < end)
   {
      if(*x == '\0' || *x == ' ')
      {
        rev(y,x-1);
        y = x+1;
      }
   }
   // Now print the final string....
   printf("%s\n",buf);

   return(0);
}



// Function to reverse a string in place...
void rev(char *l,char *r)
{
   char t;
   while(l<r)
   {
      t    = *l;
      *l++ = *r;
      *r-- = t;
   }
}
```

Method2

Another way to do it is, allocate as much memory as the input for the final output. Start from the right of the string and copy the words one by one to the output.

```
Input   : I am a good boy
                     <--
                <-------
              <---------
           <------------
         <--------------


Output : boy
       : boy good
       : boy good a
       : boy good a am
       : boy good a am I
```

The only problem to this solution is the extra space required for the output and one has to write this code really well as we are traversing the string in the reverse direction and there is no null at the start of the string to know we have reached the start of the string!. One can use the strtok() function to breakup the string into multiple words and rearrange them in the reverse order later.

Method3

Create a linked list like

```
+---+    +----------+    +----+    +----------+    +---+    +----------
+
| I | -> | <spaces> | -> | am | -> | <spaces> | -> | a | -
> | <spaces> | --+
+---+    +----------+    +----+    +----------+    +---+    +----------
+   |

    |

    |
 +---------------------------------------------------------------------
----+
 |
 |     +------+    +----------+    +-----+    +------+
 +---> | good | -> | <spaces> | -> | boy | -> | NULL |
       +------+    +----------+    +-----+    +------+
```

Now its a simple question of reversing the linked list!. There are plenty of algorithms to reverse a linked list easily. This also keeps track of the number of spaces between the words. Note that the linked list algorithm, though inefficient, handles multiple spaces between the words really well.

Write a C program generate permutations.

Iterative C program

```
#include <stdio.h>
#define SIZE 3
int main(char *argv[],int argc)
{
  char list[3]={'a','b','c'};
  int i,j,k;

  for(i=0;i<SIZE;i++)
    for(j=0;j<SIZE;j++)
      for(k=0;k<SIZE;k++)
        if(i!=j && j!=k && i!=k)
          printf("%c%c%c\n",list[i],list[j],list[k]);

  return(0);
}
```

Recursive C program

```
#include <stdio.h>
#define N  5


int main(char *argv[],int argc)
{
  char list[5]={'a','b','c','d','e'};
  permute(list,0,N);
  return(0);
}


void permute(char list[],int k, int m)
{
  int i;
  char temp;

  if(k==m)
  {
    /* PRINT A FROM k to m! */
    for(i=0;i<N;i++){printf("%c",list[i]);}
    printf("\n");
  }
  else
  {
     for(i=k;i<m;i++)
     {
        /* swap(a[i],a[m-1]); */
        temp=list[i];
        list[i]=list[m-1];
        list[m-1]=temp;

        permute(list,k,m-1);

        /* swap(a[m-1],a[i]); */

        temp=list[m-1];
        list[m-1]=list[i];
        list[i]=temp;
       }
  }
}
```

Write a C program for calculating the factorial of a number

Here is a recursive C program

```
fact(int n)
{
    int fact;
    if(n==1)
        return(1);
    else
        fact = n * fact(n-1);
    return(fact);
}
```

Please note that there is no error handling added to this function (to check if n is negative or 0. Or if n is too large for the system to handle). This is true for most of the answers in this website. Too much error handling and standard compliance results in a lot of clutter making it difficult to concentrate on the crux of the solution. You must ofcourse add as much error handling and comply to the standards of your compiler when you actually write the code to implement these algorithms.

Write a C program to calculate pow(x,n)?

There are again different methods to do this in C

Brute force C program

```
int pow(int x, int y)
{
  if(y == 1) return x ;
  return x * pow(x, y-1) ;
}
```

Divide and Conquer C program

```
#include <stdio.h>
int main(int argc, char*argv[])
{
  printf("\n[%d]\n",pow(5,4));
}

int pow(int x, int n)
{
  if(n==0)return(1);
  else if(n%2==0)
  {
    return(pow(x,n/2)*pow(x,(n/2)));
  }
  else
  {
    return(x*pow(x,n/2)*pow(x,(n/2)));
  }
}
```

Also, the code above can be optimized still by calculating pow(z, (n/2)) only one time (instead of twice) and using its value in the two return() expressions above.

Write a C program which does wildcard pattern matching algorithm

Here is an example C program...

```
#include<stdio.h>
#define TRUE 1
#define FALSE 0

int wildcard(char *string, char *pattern);

int main()
{
  char *string = "hereheroherr";
  char *pattern = "*hero*";

  if(wildcard(string, pattern)==TRUE)
  {
    printf("\nMatch Found!\n");
  }
  else
  {
    printf("\nMatch not found!\n");
  }
  return(0);
}

int wildcard(char *string, char *pattern)
{
  while(*string)
  {
     switch(*pattern)
     {
        case '*': do {++pattern;}while(*pattern == '*');
                  if(!*pattern) return(TRUE);
                  while(*string){if(wildcard(pattern,string++)==TRUE)re
turn(TRUE);}
                  return(FALSE);
        default : if(*string!=*pattern)return(FALSE); break;
     }
     ++pattern;
     ++string;
    }
```

```
   while (*pattern == '*') ++pattern;
   return !*pattern;
}
```

How do you calculate the maximum subarray of a list of numbers?

This is a very popular question

You are given a large array X of integers (both positive and negative) and you need to find the maximum sum found in any contiguous subarray of X.

```
Example X = [11, -12, 15, -3, 8, -9, 1, 8, 10, -2]
Answer is 30.
```

There are various methods to solve this problem, some are listed below

Brute force

```
maxSum = 0
for L = 1 to N
{
  for R = L to N
  {
     sum = 0
     for i = L to R
     {
        sum = sum + X[i]
     }
     maxSum = max(maxSum, sum)
   }
}

O(N^3)
```

Quadratic

Note that sum of [L..R] can be calculated from sum of [L..R-1] very easily.

```
maxSum = 0
for L = 1 to N
{
  sum = 0
  for R = L to N
  {
    sum = sum + X[R]
    maxSum = max(maxSum, sum)
  }
}
```

Using divide-and-conquer

```
O(N log(N))

maxSum(L, R)
{
  if L > R then
    return 0

  if L = R then
    return max(0, X[L])

  M = (L + R)/2

  sum = 0; maxToLeft = 0
  for i = M downto L do
  {
     sum = sum + X[i]
     maxToLeft = max(maxToLeft, sum)
  }

  sum = 0; maxToRight = 0
  for i = M to R do
  {
     sum = sum + X[i]
     maxToRight = max(maxToRight, sum)
  }


  maxCrossing = maxLeft + maxRight
  maxInA = maxSum(L,M)
  maxInB = maxSum(M+1,R)
  return max(maxCrossing, maxInA, maxInB)
}
```

Here is working C code for all the above cases

```
#include<stdio.h>
#define N 10
int maxSubSum(int left, int right);
int list[N] = {11, -12, 15, -3, 8, -9, 1, 8, 10, -2};

int main()
{
  int i,j,k;
  int maxSum, sum;

  /*---------------------------------------
  * CUBIC - O(n*n*n)
  *---------------------------------------*/
  maxSum = 0;
  for(i=0; i<N; i++)
  {
    for(j=i; j<N; j++)
    {
      sum = 0;
      for(k=i ; k<j; k++)
      {
        sum = sum + list[k];
      }
      maxSum = (maxSum>sum)?maxSum:sum;
    }
   }

   printf("\nmaxSum = [%d]\n", maxSum);


   /*------------------------------------
    * Quadratic - O(n*n)
    * ------------------------------------ */

   maxSum = 0;
   for(i=0; i<N; i++)
   {
      sum=0;
      for(j=i; j<N ;j++)
      {
         sum = sum + list[j];
         maxSum = (maxSum>sum)?maxSum:sum;
      }
   }

   printf("\nmaxSum = [%d]\n", maxSum);


   /*---------------------------------------
    * Divide and Conquer - O(nlog(n))
    * -------------------------------------- */

   printf("\nmaxSum : [%d]\n", maxSubSum(0,9));

   return(0);
}


int maxSubSum(int left, int right)
{
  int mid, sum, maxToLeft, maxToRight, maxCrossing, maxInA, maxInB;
  int i;

  if(left>right){return 0;}
  if(left==right){return((0>list[left])?0:list[left]);}
  mid = (left + right)/2;

  sum=0;
  maxToLeft=0;
  for(i=mid; i>=left; i--)
  {
     sum = sum + list[i];
     maxToLeft = (maxToLeft>sum)?maxToLeft:sum;
  }


  sum=0;
  maxToRight=0;
  for(i=mid+1; i<=right; i++)
  {
      sum = sum + list[i];
      maxToRight = (maxToRight>sum)?maxToRight:sum;
  }

  maxCrossing = maxToLeft + maxToRight;
  maxInA = maxSubSum(left,mid);
  maxInB = maxSubSum(mid+1,right);
  return(((maxCrossing>maxInA)?maxCrossing:maxInA)>maxInB?((maxCrossing
>maxInA)?maxCrossing:maxInA):maxInB);
}
```

Note that, if the array has all negative numbers, then this code will return 0. This is wrong because it should return the maximum sum, which is the least negative integer in the array. This happens because we are setting maxSum to 0 initially. A small change in this code can be used to handle such cases.

How to generate fibonacci numbers? How to find out if a given number is a fibonacci number or not? Write C programs to do both.

Lets first refresh ourselves with the Fibonacci sequence

```
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, .....
```

Fibonacci numbers obey the following rule

```
F(n) = F(n-1) + F(n-2)
```

Here is an iterative way to generate fibonacci numbers and also return the nth number.

```
int fib(int n)
{
   int f[n+1];
   f[1] = f[2] = 1;

   printf("\nf[1] = %d", f[1]);
   printf("\nf[2] = %d", f[2]);

   for (int i = 3; i <= n; i++)
   {
       f[i] = f[i-1] + f[i-2];
```

```
        printf("\nf[%d] = [%d]",i,f[i]);
    }
    return f[n];
}
```

Here is a recursive way to generate fibonacci numbers.

```
int fib(int n)
{
  if (n <= 2) return 1
  else return fib(n-1) + fib(n-2)
}
```

Here is an iterative way to just compute and return the nth number (without storing the previous numbers).

```
int fib(int n)
{
  int a = 1, b = 1;
  for (int i = 3; i <= n; i++)
  {
     int c = a + b;
     a = b;
     b = c;
  }
  return a;
}
```

There are a few slick ways to generate fibonacci numbers, a few of them are listed below

Method1

If you know some basic math, its easy to see that

```
         n
 [ 1 1 ]       =    [ F(n+1) F(n)   ]
 [ 1 0 ]            [ F(n)   F(n-1) ]
```

or

```
(f(n) f(n+1)) [ 0 1 ] = (f(n+1) f(n+2))
              [ 1 1 ]
```

or

```
                  n
(f(0) f(1)) [ 0 1 ]  = (f(n) f(n+1))
            [ 1 1 ]
```

The n-th power of the 2 by 2 matrix can be computed efficiently in O(log n) time. This implies an O(log n) algorithm for computing the n-th Fibonacci number.

Here is the pseudocode for this

```
int Matrix[2][2] = {{1,0}{0,1}}

int fib(int n)
{
    matrixpower(n-1);
    return Matrix[0][0];
}

void matrixpower(int n)
{
   if (n > 1)
   {
      matrixpower(n/2);
      Matrix = Matrix * Matrix;
   }
   if (n is odd)
   {
      Matrix = Matrix * {{1,1}{1,0}}
   }
}
```

And here is a program in C which calculates fib(n)

```
#include<stdio.h>

int M[2][2]={{1,0},{0,1}};
int A[2][2]={{1,1},{1,0}};
int C[2][2]={{0,0},{0,0}};  // Temporary matrix used for multiplication
.

void matMul(int n);
void mulM(int m);

int main()
{
   int n;
   n=6;
   matMul(n-1);

   // The nth fibonacci will be stored in M[0][0]
   printf("\n%dth Fibonaci number : [%d]\n\n", n, M[0][0]);
   return(0);

}


// Recursive function with divide and conquer strategy

void matMul(int n)
{
   if(n>1)
   {
     matMul(n/2);
     mulM(0);     // M * M
   }
   if(n%2!=0)
   {
    mulM(1);     //  M * {{1,1}{1,0}}
   }
}


// Function which does some basic matrix multiplication.

void mulM(int m)
{
   int i,j,k;

   if(m==0)
   {
     // C = M * M

     for(i=0;i<2;i++)
      for(j=0;j<2;j++)
      {
        C[i][j]=0;
        for(k=0;k<2;k++)
           C[i][j]+=M[i][k]*M[k][j];
      }
   }
   else
   {
     // C = M *  {{1,1}{1,0}}

     for(i=0;i<2;i++)
      for(j=0;j<2;j++)
      {
        C[i][j]=0;
        for(k=0;k<2;k++)
         C[i][j]+=A[i][k]*M[k][j];
      }
   }

   // Copy back the temporary matrix in the original matrix M
```

```
    for(i=0;i<2;i++)
      for(j=0;j<2;j++)
      {
        M[i][j]=C[i][j];
      }
}
```

Method2

```
f(n) = (1/sqrt(5)) * (((1+sqrt(5))/2) ^ n - ((1-sqrt(5))/2) ^ n)
```

So now, how does one find out if a number is a fibonacci or not?.

The cumbersome way is to generate fibonacci numbers till this number and see if this number is one of them. But there is another slick way to check if a number is a fibonacci number or not.

```
N is a Fibonacci number if and only if (5*N*N + 4) or (5*N*N -
 4) is a perfect square!
```

Dont believe me?

```
3 is a Fibonacci number since (5*3*3 + 4) is 49 which is 7*7
5 is a Fibonacci number since (5*5*5 - 4) is 121 which is 11*11
4 is not a Fibonacci number since neither (5*4*4 + 4) = 84 nor (5*4*4 -
 4) = 76 are perfect squares.
```

To check if a number is a perfect square or not, one can take the square root, round it to the nearest integer and then square the result. If this is the same as the original whole number then the original was a perfect square.

Solve the Rat In A Maze problem using backtracking.

This is one of the classical problems of computer science. There is a rat trapped in a maze. There are multiple paths in the maze from the starting point to the ending point. There is some cheese at the exit. The rat starts from the entrance of the maze and wants to get to the cheese.

This problem can be attacked as follows.

- Have a m*m matrix which represents the maze.
- For the sake of simplifying the implementation, have a boundary around your matrix and fill it up with all ones. This is so that you know when the rat is trying to go out of the boundary of the maze. In the real world, the rat would know not to go out of the maze, but hey! So, initially the matrix (I mean, the maze) would be

something like (the ones represent the "exra" boundary we have added). The ones inside specify the obstacles.

```
111111111111111111111
100000000000000000001
100000010000000000001
100000010000000000001
100000000100001000001
100001000010000000001
100000000100000000001
100000000000000000001
111111111111111111111
```

- The rat can move in four directions at any point in time (well, right, left, up, down). Please note that the rat can't move diagonally. Imagine a real maze and not a matrix. In matrix language
    - Moving right means adding {0,1} to the current coordinates.
    - Moving left means adding {0,-1} to the current coordinates.
    - Moving up means adding {-1,0} to the current coordinates.
    - Moving right means adding {1,0} to the current coordinates.
- The rat can start off at the first row and the first column as the entrance point.
- From there, it tries to move to a cell which is currently free. A cell is free if it has a zero in it.
- It tries all the 4 options one-by-one, till it finds an empty cell. If it finds one, it moves to that cell and marks it with a 1 (saying it has visited it once). Then it continues to move ahead from that cell to other cells.
- If at a particular cell, it runs out of all the 4 options (that is it cant move either right, left, up or down), then it needs to backtrack. It backtracks till a point where it can move ahead and be closer to the exit.
- If it reaches the exit point, it gets the cheese, ofcourse.
- The complexity is O(m*m).

Here is some pseudocode to chew upon

```
findpath()
{
  Position offset[4];
  Offset[0].row=0; offset[0].col=1;//right;
  Offset[1].row=1; offset[1].col=0;//down;
  Offset[2].row=0; offset[2].col=-1;//left;
  Offset[3].row=-1; offset[3].col=0;//up;

  // Initialize wall of obstacles around the maze
  for(int i=0; i<m+1;i++)
    maze[0][i] = maze[m+1][i]=1; maze[i][0] = maze[i][m+1]=1;
  Position here;
  Here.row=1;
  Here.col=1;

  maze[1][1]=1;
  int option = 0;
  int lastoption = 3;

  while(here.row!=m || here.col!=m)
  {
     //Find a neighbor to move
     int r,c;

       while (option<=LastOption)
       {
          r=here.row + offset[position].row;
          c=here.col + offset[option].col;
          if(maze[r][c]==0)break;
          option++;
       }

       //Was a neighbor found?
       if(option<=LastOption)
       {
         path->Add(here);
         here.row=r;here.col=c;
         maze[r][c]=1;
         option=0;
       }
       else
       {
          if(path->Empty())return(False);
          Position  next;
          Path->Delete(next);
          If(new.row==here.row)
             Option=2+next.col - here.col;
          Else { option = 3 + next.row - here.col;}
             Here=next;
       }
       return(TRUE);
  }
}
```

What Little-Endian and Big-Endian? How can I determine whether a machine's byte order is big-endian or little endian? How can we convert from one to another?

First of all, Do you know what Little-Endian and Big-Endian mean?

Little Endian means that the lower order byte of the number is stored in memory at the lowest address, and the higher order byte is stored at the highest address. That is, the little end comes first.

For example, a 4 byte, 32-bit integer

```
    Byte3 Byte2 Byte1 Byte0
```

will be arranged in memory as follows:

```
    Base_Address+0   Byte0
    Base_Address+1   Byte1
    Base_Address+2   Byte2
    Base_Address+3   Byte3
```

Intel processors use "Little Endian" byte order.

"Big Endian" means that the higher order byte of the number is stored in memory at the lowest address, and the lower order byte at the highest address. The big end comes first.

```
  Base_Address+0   Byte3
  Base_Address+1   Byte2
  Base_Address+2   Byte1
  Base_Address+3   Byte0
```

Motorola, Solaris processors use "Big Endian" byte order.

In "Little Endian" form, code which picks up a 1, 2, 4, or longer byte number proceed in the same way for all formats. They first pick up the lowest order byte at offset 0 and proceed from there. Also, because of the 1:1 relationship between address offset and byte number (offset 0 is byte 0), multiple precision mathematic routines are easy to code. In "Big Endian" form, since the high-order byte comes first, the code can test whether the number is positive or negative by looking at the byte at offset zero. Its not required to know how long the number is, nor does the code have to skip over any bytes to find the byte containing the sign information. The numbers are also stored in the order in which they are printed out, so binary to decimal routines are particularly efficient.

Here is some code to determine what is the type of your machine

```
int num = 1;
if(*(char *)&num == 1)
{
  printf("\nLittle-Endian\n");
}
else
```

```
{
  printf("Big-Endian\n");
}
```

And here is some code to convert from one Endian to another.

```
int myreversefunc(int num)
{
  int byte0, byte1, byte2, byte3;

  byte0 = (num & x000000FF) >>  0 ;
  byte1 = (num & x0000FF00) >>  8 ;
  byte2 = (num & x00FF0000) >> 16 ;
  byte3 = (num & xFF000000) >> 24 ;

  return((byte0 << 24) | (byte1 << 16) | (byte2 << 8) | (byte3 << 0));
}
```

Write C code to solve the Tower of Hanoi problem.

Here is an example C program to solve the Tower Of Hanoi problem...

```
main()
{
    towers_of_hanio(n,'L','R','C');
}

towers_of_hanio(int n, char from, char to, char temp)
{
    if(n>0)
    {
        tower_of_hanio(n-1, from, temp, to);
        printf("\nMove disk %d from %c to %c\n", n, from, to);
        tower_of_hanio(n-1, temp, to, from);
    }
}
```

Write C code to return a string from a function

This is one of the most popular interview questions

This C program wont work!

```
char *myfunction(int n)
{
   char buffer[20];
   sprintf(buffer, "%d", n);
   return retbuf;
}
```

This wont work either!

```
char *myfunc1()
{
  char temp[] = "string";
  return temp;
}


char *myfunc2()
{
   char temp[] = {'s', 't', 'r', 'i', 'n', 'g', '\0'};
   return temp;
}


int main()
{
    puts(myfunc1());
    puts(myfunc2());
}
```

The returned pointer should be to a static buffer (like static char buffer[20];), or to a buffer passed in by the caller function, or to memory obtained using malloc(), but not to a local array.

This will work

```
char *myfunc()
{
   char *temp = "string";
   return temp;
}

int main()
{
   puts(someFun());
}
```

So will this

```
calling_function()
```

```
{
  char *string;
  return_string(&string);
  printf(?\n[%s]\n?, string);
}

boolean return_string(char **mode_string /*Pointer to a pointer! */)
{
   *string = (char *) malloc(100 * sizeof(char));
   DISCARD strcpy((char *)*string, (char *)?Something?);
}
```

Write a C program which produces its own source code as its output

This is one of the most famous interview questions

One of the famous C programs is...

```
char*s="char*s=%c%s%c;main(){printf(s,34,s,34);}";main(){printf(s,34,s,
34);}
```

So how does it work?

It's not difficult to understand this program. In the following statement,

```
printf(f,34,f,34,10);
```

the parameter "f" not only acts as the format string, but also as a value for the %s specifier. The ASCII value of double quotes is 34, and that of new-line is 10. With these fact ready, the solution is just a matter of tracing the program.

Write a C progam to convert from decimal to any base (binary, hex, oct etc...)

Here is some really cool C code

```
#include <stdio.h>

int main()
{
  decimal_to_anybase(10, 2);
  decimal_to_anybase(255, 16);
  getch();
}

decimal_to_anybase(int n, int base)
{
  int i, m, digits[1000], flag;
  i=0;

  printf("\n\n[%d] converted to base [%d] : ", n, base);

  while(n)
  {
     m=n%base;
     digits[i]="0123456789abcdefghijklmnopqrstuvwxyz"[m];
     n=n/base;
     i++;
   }

   //Eliminate any leading zeroes
   for(i--;i>=0;i--)
   {
     if(!flag && digits[i]!='0')flag=1;
     if(flag)printf("%c",digits[i]);
   }
}
```

Write C code to check if an integer is a power of 2 or not in a single line?

Even this is one of the most frequently asked interview questions. I really dont know whats so great in it. Nevertheless, here is a C program

Method1

```
if(!(num & (num - 1)) && num)
{
  // Power of 2!
}
```

Method2

```
if(((~i+1)&i)==i)
{
 //Power of 2!
}
```

Write a C program to find the GCD of two numbers.

Here is a C program ....

```
#include <stdio.h>

int gcd(int a, int b);
int gcd_recurse(int a, int b);


int main()
{
  printf("\nGCD(%2d,%2d) = [%d]", 6,4,  gcd(6,4));
  printf("\nGCD(%2d,%2d) = [%d]", 4,6,  gcd(4,6));
  printf("\nGCD(%2d,%2d) = [%d]", 3,17, gcd(3,17));
  printf("\nGCD(%2d,%2d) = [%d]", 17,3, gcd(17,3));
  printf("\nGCD(%2d,%2d) = [%d]", 1,6,  gcd(1,6));
  printf("\nGCD(%2d,%2d) = [%d]", 10,1, gcd(10,1));
  printf("\nGCD(%2d,%2d) = [%d]", 10,6, gcd(10,6));

  printf("\nGCD(%2d,%2d) = [%d]", 6,4,  gcd_recurse(6,4));
  printf("\nGCD(%2d,%2d) = [%d]", 4,6,  gcd_recurse(4,6));
  printf("\nGCD(%2d,%2d) = [%d]", 3,17, gcd_recurse(3,17));
  printf("\nGCD(%2d,%2d) = [%d]", 17,3, gcd_recurse(17,3));
  printf("\nGCD(%2d,%2d) = [%d]", 1,6,  gcd_recurse(1,6));
  printf("\nGCD(%2d,%2d) = [%d]", 10,1, gcd_recurse(10,1));
  printf("\nGCD(%2d,%2d) = [%d]", 10,6, gcd_recurse(10,6));


  getch();
  getch();
  return(0);
}

// Iterative algorithm
int gcd(int a, int b)
{
   int temp;

   while(b)
   {
      temp = a % b;
      a = b;
      b = temp;
   }

   return(a);
}


// Recursive algorithm
int gcd_recurse(int a, int b)
{
   int temp;
   temp = a % b;

   if (temp == 0)
   {
      return(b);
   }
   else
   {
      return(gcd_recurse(b, temp));
   }
}
```

And here is the output ...

```
Iterative
----------------
GCD( 6, 4) = [2]
GCD( 4, 6) = [2]
GCD( 3,17) = [1]
GCD(17, 3) = [1]
GCD( 1, 6) = [1]
GCD(10, 1) = [1]
GCD(10, 6) = [2]

Recursive
----------------
GCD( 6, 4) = [2]
GCD( 4, 6) = [2]
GCD( 3,17) = [1]
GCD(17, 3) = [1]
GCD( 1, 6) = [1]
GCD(10, 1) = [1]
GCD(10, 6) = [2]
```

Note that you should add error handling to check if someone has passed negative numbers and zero.

Finding a duplicated integer problem .

Given an array of n integers from 1 to n with one integer repeated..

Here is the simplest of C programs (kind of dumb)

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>

int i,j=0,k,a1[10];

main()
```

```
{
     printf("Enter the array of numbers between 1 and 100(you can repea
t the numbers):");
     for(i=0;i<=9;i++)
     {
         scanf("%d",&a1[i]);
     }

     while(j<10)
     {
        for(k=0;k<10;k++)
        {
            if(a1[j]==a1[k] && j!=k)
            {
                printf("Duplicate found!");
                printf("The duplicate is %d\n",a1[j]);
                getch();
            }
        }
        j=j+1;
    }

    getch();
    return(0);
}
```

Write code to remove duplicates in a sorted array.

There are a number of ways to remove duplicates from a sorted array. Here are a few C programs...

Method1

In this simple C program, we change the original array and also send the new size of the array back to the caller.

```
#include <stdio.h>

int removeDuplicates(int a[], int array_size);

// The main function
int main()
{

  // Different test cases..
  int my_array1[]    = {1, 1, 2, 3, 5, 6, 6, 7, 10, 25, 100, 123, 123};
  int my_array1_size = 13;

  int my_array2[]    = {1, 2, 3, 5, 6};
  int my_array2_size = 5;
  int my_array3[]    = {1, 1, 1, 1, 1};
  int my_array3_size = 5;

  int my_array4[]    = {123, 123};
  int my_array4_size = 2;

  int my_array5[]    = {1, 123, 123};
  int my_array5_size = 3;

  int my_array6[]    = {123, 123, 166};
  int my_array6_size = 3;

  int my_array7[]    = {1, 2, 8, 8 , 24, 60, 60, 60, 60, 75, 100, 100,
123};
  int my_array7_size = 13;


  my_array1_size = removeDuplicates(my_array1, my_array1_size);
  my_array2_size = removeDuplicates(my_array2, my_array2_size);
  my_array3_size = removeDuplicates(my_array3, my_array3_size);
  my_array4_size = removeDuplicates(my_array4, my_array4_size);
  my_array5_size = removeDuplicates(my_array5, my_array5_size);
  my_array6_size = removeDuplicates(my_array6, my_array6_size);
  my_array7_size = removeDuplicates(my_array7, my_array7_size);

  return(0);
}


// Function to remove the duplicates
int removeDuplicates(int a[], int array_size)
{
  int i, j;

  j = 0;


  // Print old array...
  printf("\n\nOLD : ");
  for(i = 0; i < array_size; i++)
  {
     printf("[%d] ", a[i]);
  }


  // Remove the duplicates ...
  for (i = 1; i < array_size; i++)
  {
    if (a[i] != a[j])
    {
      j++;
      a[j] = a[i]; // Move it to the front
    }
  }

  // The new array size..
  array_size = (j + 1);
  // Print new array...
  printf("\n\nNEW : ");
  for(i = 0; i< array_size; i++)
  {
     printf("[%d] ", a[i]);
  }
  printf("\n\n");



  // Return the new size...
  return(j + 1);
}
```

And here is the output...

```
OLD : [1] [1] [2] [3] [5] [6] [6] [7] [10] [25] [100] [123] [123]
NEW : [1] [2] [3] [5] [6] [7] [10] [25] [100] [123]


OLD : [1] [2] [3] [5] [6]
NEW : [1] [2] [3] [5] [6]


OLD : [1] [1] [1] [1] [1]
NEW : [1]


OLD : [123] [123]
NEW : [123]


OLD : [1] [123] [123]
NEW : [1] [123]


OLD : [123] [123] [166]
NEW : [123] [166]


OLD : [1] [2] [8] [8] [24] [60] [60] [60] [60] [75] [100] [100] [123]
NEW : [1] [2] [8] [24] [60] [75] [100] [123]
```

Method2

If we dont want to change the input array and just want to print the array without any duplicates, the solution is very simple. Check out the removeDuplicatesNoModify() function in the program below. It keeps a track of the most recently seen number and does not print any duplicates of it when traversing the sorted array.

```c
#include <stdio.h>

void removeDuplicatesNoModify(int my_array[], int my_array1_size);
void print_array(int array[], int array_size, int current_pos, int dup_
start, int dup_end);


// The main function
int main()
{
  // Different inputs...
  int my_array1[]    = {1, 1, 2, 3, 5, 6, 6, 7, 10, 25, 100, 123, 123};
  int my_array1_size = 13;

  int my_array2[]    = {1, 2, 3, 5, 6};
  int my_array2_size = 5;

  int my_array3[]    = {1, 1, 1, 1, 1};
  int my_array3_size = 5;

  int my_array4[]    = {123, 123};
  int my_array4_size = 2;

  int my_array5[]    = {1, 123, 123};
  int my_array5_size = 3;

  int my_array6[]    = {123, 123, 166};
  int my_array6_size = 3;

  int my_array7[]    = {1, 2, 8, 8 , 24, 60, 60, 60, 60, 75, 100, 100,
123};
  int my_array7_size = 13;


  removeDuplicatesNoModify(my_array1, my_array1_size);
  removeDuplicatesNoModify(my_array2, my_array2_size);
  removeDuplicatesNoModify(my_array3, my_array3_size);
  removeDuplicatesNoModify(my_array4, my_array4_size);
  removeDuplicatesNoModify(my_array5, my_array5_size);
  removeDuplicatesNoModify(my_array6, my_array6_size);
  removeDuplicatesNoModify(my_array7, my_array7_size);

  return(0);
}



//  This function just prints the array without duplicates.
//  It does not modify the original array!

void removeDuplicatesNoModify(int array[], int array_size)
{
   int i, last_seen_unique;

   if(array_size <= 1){return;}

   last_seen_unique = array[0];
   printf("\n\nOld : ", array_size);

   for(i = 0; i < array_size; i++)
   {
     printf("[%2d] ", array[i]);
   }

   printf("\nNew : ", array_size);

   printf("[%2d] ", array[0]);
   for(i=1; i < array_size; i++)
   {
       if(array[i]!=last_seen_unique)
       {
          printf("[%2d] ", array[i]);
          last_seen_unique = array[i];
       }
   }

   printf("\n");
}
```

And here is the output..

```
Old : [ 1] [ 1] [ 2] [ 3] [ 5] [ 6] [ 6] [ 7] [10] [25] [100] [123] [12
3]
New : [ 1] [ 2] [ 3] [ 5] [ 6] [ 7] [10] [25] [100] [123]


Old : [ 1] [ 2] [ 3] [ 5] [ 6]
New : [ 1] [ 2] [ 3] [ 5] [ 6]


Old : [ 1] [ 1] [ 1] [ 1] [ 1]
New : [ 1]


Old : [123] [123]
New : [123]


Old : [ 1] [123] [123]
New : [ 1] [123]


Old : [123] [123] [166]
New : [123] [166]


Old : [ 1] [ 2] [ 8] [ 8] [24] [60] [60] [60] [60] [75] [100] [100] [12
3]
New : [ 1] [ 2] [ 8] [24] [60] [75] [100] [123]
```

Method3

Here is a slightly compilcated, but more visual version of the removeDuplicates() function. It shrinks the original array as and when it find duplicates. It is also optimized to identify continuous strings of duplicates and eliminate them at one shot.

```
#include <stdio.h>

void removeDuplicates(int array[], int *array_size) ;
void print_array(int array[], int array_size, int current_pos, int dup_
start, int dup_end);


// The main function
int main()
{
  // Different inputs...
  int my_array1[]    = {1, 1, 2, 3, 5, 6, 6, 7, 10, 25, 100, 123, 123};
  int my_array1_size = 13;

  int my_array2[]    = {1, 2, 3, 5, 6};
  int my_array2_size = 5;

  int my_array3[]    = {1, 1, 1, 1, 1};
  int my_array3_size = 5;

  int my_array4[]    = {123, 123};
  int my_array4_size = 2;

  int my_array5[]    = {1, 123, 123};
  int my_array5_size = 3;

  int my_array6[]    = {123, 123, 166};
  int my_array6_size = 3;

  int my_array7[]    = {1, 2, 8, 8 , 24, 60, 60, 60, 60, 75, 100, 100,
123};
  int my_array7_size = 13;

  removeDuplicates(my_array1, &my_array1_size);
  removeDuplicates(my_array2, &my_array2_size);
  removeDuplicates(my_array3, &my_array3_size);
  removeDuplicates(my_array4, &my_array4_size);
  removeDuplicates(my_array5, &my_array5_size);
  removeDuplicates(my_array6, &my_array6_size);
  removeDuplicates(my_array7, &my_array7_size);

  return(0);
}

// Changes the original array and resets the size of the array if dupli
cates
// have been removed.

void removeDuplicates(int array[], int *array_size)
```

```
{
   int i, j, k, l;
   int current_pos;
   int dup_start;
   int dup_end;

   printf("\nInitial array (size : [%d])\n\n", *array_size);
   for(i = 0; i < *array_size; i++)
   {
     printf("[%2d] ", array[i]);
   }
   printf("\n\n\n------------------------------------------------\n");


   if(*array_size == 1){return;}


   // Remove the dups...
   for (i = 0; (i < *array_size); i++)
   {
      //Start with the next element in the array and check if its a dup
licate...
      for(j = i+1; (j < *array_size); j++)
      {
         if(array[i]!=array[j])
         {
            // No duplicate, just continue...
            break;
         }
         else
         {
            // The next element is a duplicate.
            // See if there are more duplicates, as we want to optimize
 here.
            //
            // That is, if we have something like
            //
            // Array : [1, 1, 1, 2]
            //
            // then, we want to copy 2 directly in the second position
and reduce the
            // array to
            //
            // Array : [1, 2].
            //
            // in a single iteration.

            current_pos = i;
            dup_start = j;

            j++;

            while((array[i]==array[j]) && (j < *array_size))
            {
              j++;
            }
            dup_end = j-1;

            print_array(array, *array_size, current_pos, dup_start, dup
_end);

            // Now remove elements of the array from "dup_start" to "du
p_end"
            // and shrink the size of the array.

            for(k = (dup_end + 1), l = dup_start ; k < *array_size;)
            {
               array[l++]=array[k++];
            }

            // Reduce the array size by the number of elements removed.
            *array_size = *array_size - (dup_end - dup_start + 1);

         }
      }
   }

   printf("\n\n------------------------------------------------");
   printf("\n\nFinal array (size : [%d])\n\n", *array_size);
   for(i = 0; i < *array_size; i++)
   {
     printf("[%2d] ", array[i]);
   }
   printf("\n\n");

}




// This function prints the array with some special pointers to the num
bers that
// are duplicated.
//
// Dont bother too much about this function, it just helps in understan
ding
// how and where the duplicates are being removed from.

void print_array(int array[], int array_size, int current_pos, int dup_
start, int dup_end)
{
  int i;

  printf("\n\n");

  for(i = 0; i < array_size; i++)
  {
    printf("[%2d] ", array[i]);
  }

  printf("\n");

  for(i = 0; i < array_size; i++)
```

```
{
  if((i == current_pos) ||
     (i == dup_start && i == dup_end) ||
     ((i == dup_start || i == dup_end) && (dup_start != dup_end)))
  {
     printf("  ^  ");
  }
  else
  {
     printf("     ");
  }
}

printf("\n");

for(i = 0; i < array_size; i++)
{
  if((i == current_pos) ||
     (i == dup_start && i == dup_end) ||
     ((i == dup_start || i == dup_end) && (dup_start != dup_end)))
  {
     printf("  |  ");
  }
  else
  {
     printf("     ");
  }
}

printf("\n");

for(i = 0; i < array_size; i++)
{
  if(i == current_pos)
  {
     printf("  C  ");
  }
  else if(i == dup_start && i == dup_end)
  {
     printf(" S/E ");
  }
  else if((i == dup_start || i == dup_end) && (dup_start != dup_end))
  {
     if(i == dup_start)
     {
        printf("  S--");
     }
     else
     {
        printf("--E  ");
     }
  }
  else if(i>dup_start && i<dup_end)
  {
     printf("-----");
  }
  else
```

```
    {
       printf("     ");
    }
  }

}
```

And here is the output (for one of the inputs)...

```
C - Current position.
S - Start of duplicates.
E - End of duplicates.

Old : [ 1] [ 2] [ 8] [ 8] [24] [60] [60] [60] [60] [75] [100] [100] [12
3]

-----------------------------------------------------------------------
--
[ 1] [ 2] [ 8] [ 8] [24] [60] [60] [60] [60] [75] [100] [100] [123]
            ^    ^
            |    |
            C   S/E


[ 1] [ 2] [ 8] [24] [60] [60] [60] [60] [75] [100] [100] [123]
                      ^    ^         ^
                      |    |         |
                      C    S---------E


[ 1] [ 2] [ 8] [24] [60] [75] [100] [100] [123]
                                ^    ^
                                |    |
                                C   S/E

-----------------------------------------------------------------------
--
New : [ 1] [ 2] [ 8] [24] [60] [75] [100] [123]
```

If there are other elegant methods of removing duplicate numbers from an array, please let me know!.

Find the maximum of three integers using the ternary operator.

Here is how you do it

```
max = ((a>b)?((a>c)?a:c):((b>c)?b:c));
```

Here is another way

```
max = ((a>b)?a:b)>c?((a>b)?a:b):c;
```

Here is some code to find the max of 4 numbers...

Method1

```
#include <stdio.h>
#include <stdlib.h>

#define max2(x,y)  ((x)>(y)?(x):(y))
#define max4(a,b,c,d)  max2(max2((a),(b)),max2((c),(d)))

int main ( void )
{
    printf ( "Max: %d\n", max4(10,20,30,40));
    printf ( "Max: %d\n", max4(10,0,3,4));
    return EXIT_SUCCESS;
}
```

Method2

```
#include <stdio.h>
#include <stdlib.h>

int retMax(int i1, int i2, int i3, int i4)
{
   return(((i1>i2)?i1:i2) > ((i3>i4)?i3:i4)? ((i1>i2)?i1:i2):((i3>i4)?i
3:i4));
}

int main()
{
   int val = 0 ;

   val = retMax(10, 200, 10, 530);
   val = retMax(9, 2, 5, 7);

   return 0;
}
```

How do you initialize a pointer inside a function?

This is one of the very popular interview questions, so take a good look at it!.

```
myfunction(int *ptr)
{
  int myvar = 100;
  ptr = &myvar;
}

main()
{
 int *myptr;
 myfunction(myptr);

 //Use pointer myptr.

}
```

Do you think this works? It does not!.

Arguments in C are passed by value. The called function changed the passed copy of the pointer, and not the actual pointer.

There are two ways around this problem

Method1

Pass in the address of the pointer to the function (the function needs to accept a pointer-to-a-pointer).

```
calling_function()
{
  char *string;
  return_string(/* Pass the address of the pointer */&string);
  printf(?\n[%s]\n?, string);
}

boolean return_string(char **mode_string /*Pointer to a pointer! */)
{
  *string = (char *) malloc(100 * sizeof(char)); // Allocate memory to
the pointer passed, not its copy.
  DISCARD strcpy((char *)*string, (char *)?Something?);
}
```

Method2

Make the function return the pointer.

```
char *myfunc()
{
  char *temp = "string";
  return temp;
}

int main()
{
  puts(myfunc());
}
```

Write C code to dynamically allocate one, two and three dimensional arrays (using malloc())

Its pretty simple to do this in the C language if you know how to use C pointers. Here are some example C code snipptes....

One dimensional array

```
int *myarray = malloc(no_of_elements * sizeof(int));

//Access elements as myarray[i]
```

Two dimensional array

Method1

```
int **myarray = (int **)malloc(no_of_rows * sizeof(int *));
for(i = 0; i < no_of_rows; i++)
{
 myarray[i] = malloc(no_of_columns * sizeof(int));
}

// Access elements as myarray[i][j]
```

Method2 (keep the array's contents contiguous)

```
int **myarray = (int **)malloc(no_of_rows * sizeof(int *));
myarray[0] = malloc(no_of_rows * no_of_columns * sizeof(int));

for(i = 1; i < no_of_rows; i++)
  myarray[i] = myarray[0] + (i * no_of_columns);

// Access elements as myarray[i][j]
```

Method3

```
int *myarray = malloc(no_of_rows * no_of_columns * sizeof(int));

// Access elements using myarray[i * no_of_columns + j].
```

Three dimensional array

```
#define MAXX 3
#define MAXY 4
```

```
#define MAXZ 5

main()
{
    int ***p,i,j;
    p=(int ***) malloc(MAXX * sizeof(int ***));

    for(i=0;i<MAXX;i++)
    {
        p[i]=(int **)malloc(MAXY * sizeof(int *));
        for(j=0;j<MAXY;j++)
            p[i][j]=(int *)malloc(MAXZ * sizeof(int));
    }

    for(k=0;k<MAXZ;k++)
        for(i=0;i<MAXX;i++)
            for(j=0;j<MAXY;j++)
                p[i][j][k]=<something>;

}
```

How would you find the size of structure without using sizeof()?

Try using pointers

```
struct MyStruct
{
  int i;
  int j;
};

int main()
{
   struct MyStruct *p=0;
   int size = ((char*)(p+1))-((char*)p);
   printf("\nSIZE : [%d]\nSIZE : [%d]\n", size);
   return 0;
}
```

Write a C program to multiply two matrices.

Are you sure you know this? A lot of people think they already know this, but guess what? So take a good look at this C program. Its asked in most of the interviews as a warm up question.

```
// Matrix A (m*n)
// Matrix B (n*k)
// Matrix C (m*k)

for(i=0; i<m; i++)
```

```
{
   for(j=0;j<k;j++)
   {
      c[i][j]=0;
      for(l=0;l<n;l++)
           c[i][j] += a[i][l] * b[l][j];
   }
}
```

Write a C program to check for palindromes.

An example of a palidrome is "avon sees nova"

There a number of ways in which we can find out if a string is a palidrome or not. Here are a few sample C programs...

Method1

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>

void isPalindrome(char *string);

int main()
{
  isPalindrome("avon sees nova");
  isPalindrome("a");
  isPalindrome("avon sies nova");
  isPalindrome("aa");
  isPalindrome("abc");
  isPalindrome("aba");
  isPalindrome("3a2");
  exit(0);
}

void isPalindrome(char *string)
{
  char *start, *end;

  if(string)
  {
     start = string;
     end   = string + strlen(string) - 1;

     while((*start == *end) && (start!=end))
     {
       if(start<end)start++;
       if(end>start)end--;
     }

     if(*start!=*end)
     {
        printf("\n[%s] - This is not a palidrome!\n", string);
     }
     else
     {
        printf("\n[%s] - This is a palidrome!\n", string);
     }
  }
  printf("\n\n");
}
```

Method2

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>

int isPalindrome(char string[]);

int main()
{
  isPalindrome("avon sees nova");
  isPalindrome("a");
  isPalindrome("avon sies nova");
  isPalindrome("aa");
  isPalindrome("abc");
  isPalindrome("aba");
  isPalindrome("3a2");
  return(0);
}

int isPalindrome(char string[])
{
  int count, countback, end, N;

  N   = strlen(string);
  end = N-1;

  for((count=0, countback = end); count <= (end/2); ++count,--
countback)
  {
    if(string[count]!=string[countback])
    {
      return(1);
    }
  }

  printf("\n[%s] is a palidrome!\n", string);
  return(0);
}
```

Write a C program to convert a decimal number into a binary number.

99% of the people who attend interviews can't answer this question, believe me!. Here is a recursive C program which does this....

```
#include<stdio.h>
void generatebits(int num);

void generatebits(int num)
{
  int temp;
  if(num)
  {
    temp = num % 2;
    generatebits(num >>= 1);
    printf("%d",temp);
  }
}

int main()
{
  int num;
  printf("\nEnter a number\n");
  scanf("%d", &num);
  printf("\n\n");
  generatebits(num);
  getch();
  return(0);
}
```

The reason we have shown a recursive algorithm is that, because of the magic of recursion, we dont have to reverse the bits generated to produce the final output. One can always write an iterative algorithm to accomplish the same, but it would require you to first store the bits as they are generated and then reverse them before producing the final output.

Write C code to implement the Binary Search algorithm.

Here is a C function

```
int binarySearch(int arr[],int size, int item)
{
   int left, right, middle;
   left  = 0;
   right = size-1;
   while(left<=right)
   {
      middle = ((left + right)/2);

      if(item == arr[middle])
      {
        return(middle);
      }

      if(item > arr[middle])
      {
        left  = middle+1;
      }
      else
      {
        right = middle-1;
      }
   }

   return(-1);
}
```

Note that the Binary Search algorithm has a prerequisite that the array passed to it must be already sorted in ascending order. This will not work on an unsorted array. The complexity of this algorithm is O(log(n)).

Wite code to evaluate a polynomial.

```
typedef struct node
{
  float cf;
  float px;
  float py;
  struct node *next;
}mynode;


float evaluate(mynode *head)
{
   float x,y,sum;
   sum = 0;
   mynode *poly;

   for(poly = head->next; poly != head; poly = poly->next)
   {
     sum = sum + poly->cf * pow(x, poly->px) * pow(y, poly->py);
   }

   return(sum);
}
```

Write code to add two polynomials.

Here is some pseudocode

```
mynode *polynomial_add(mynode *h1, mynode *h2, mynode *h3)
{
   mynode *p1, *p2;
   int x1, x2, y1, y2, cf1, cf2, cf;

   p1 = h1->next;

   while(p1!=h1)
   {
      x1  = p1->px;
      y1  = p1->py;
      cf1 = p1->cf;

      // Search for this term in the second polynomial

      p2 = h2->next;

      while(p2 != h2)
      {
         x2  = p2->px;
         y2  = p2->py;
         cf2 = p2->cf;

         if(x1 == x2 && y1 == y2)break;

         p2 = p2->next;

      }


      if(p2 != h2)
      {
          // We found something in the second polynomial.

          cf = cf1 + cf2;
          p2->flag = 1;

          if(cf!=0){h3=addNode(cf,x1,y1,h3);}
      }
      else
      {
         h3=addNode(cf,x1,y1,h3);
      }

      p1 = p1->next;

   }//while


   // Add the remaining elements of the second polynomail to the result

   while(p2 != h2)
   {
      if(p2->flag==0)
      {
          h3=addNode(p2->cf, p2->px, p2->py, h3);
      }
      p2=p2->next;
   }

   return(h3);
}
```

Write a program to add two long positive numbers (each represented by linked lists).

Check out this simple implementation

```
mynode *long_add(mynode *h1, mynode *h2, mynode *h3)
{
  mynode *c, *c1, *c2;
  int sum, carry, digit;

  carry = 0;
  c1 = h1->next;
  c2 = h2->next;

  while(c1 != h1 && c2 != h2)
  {
     sum   = c1->value + c2->value + carry;
     digit = sum % 10;
     carry = sum / 10;

     h3 = insertNode(digit, h3);

     c1 = c1->next;
     c2 = c2->next;
  }

  if(c1 != h1)
  {
     c = c1;
     h = h1;
  }
  else
  {
     c = c2;
     h = h2;
  }

  while(c != h)
  {
    sum   = c->value + carry;
    digit = sum % 10;
    carry = sum / 10;
    h3 = insertNode(digit, h3);
    c = c->next;
  }

  if(carry==1)
  {
     h3 = insertNode(carry, h3);
  }

  return(h3);
}
```

How do you compare floating point numbers?

This is Wrong!.

```
double a, b;

if(a == b)
{
  ...
}
```

The above code might not work always. Thats because of the way floating point numbers are stored.

A good way of comparing two floating point numbers is to have a accuracy threshold which is relative to the magnitude of the two floating point numbers being compared.

```
#include <math.h>
if(fabs(a - b) <= accurary_threshold * fabs(a))
```

There is a lot of material on the net to know how floating point numbers can be compared. Go for it if you really want to understand.

Another way which might work is something like this. I have not tested it!

```
int compareFloats(float f1, float f2)
{
  char *b1, *b2;
  int i;

  b1 = (char *)&f1;
  b2 = (char *)&f2;

  /* Assuming sizeof(float) is 4 bytes) */

  for (i = 0; i<4; i++, b1++, b2++)
  {
    if (*b1 != *b2)
    {
      return(NOT_EQUAL); /* You must have defined this before */
    }
  }

  return(EQUAL);
}
```

What's a good way to implement complex numbers in C?

Use structures and some routines to manipulate them.

How can I display a percentage-done indication on the screen?

The character '\r' is a carriage return without the usual line feed, this helps to overwrite the current line. The character '\b' acts as a backspace, and will move the cursor one position to the left.

Write a program to check if a given year is a leap year or not?

Use this if() condition to check if a year is a leap year or not

```
if(year % 4 == 0 && (year % 100 != 0 || year % 400 == 0))
```

Is there something we can do in C but not in C++?

I have a really funny answer

Declare variable names that are keywords in C++ but not C.

```
#include <stdio.h>
int main(void)
{
  int old, new=3;
  return 0;
}
```

This will compile in C, but not in C++!

How to swap the two nibbles in a byte ?

Try this

```
#include <stdio.h>

unsigned char swap_nibbles(unsigned char c)
{
  unsigned char temp1, temp2;
  temp1 = c & 0x0F;
  temp2 = c & 0xF0;
  temp1=temp1 << 4;
  temp2=temp2 >> 4;

  return(temp2|temp1); //adding the bits
}

int main(void)
{
  char ch=0x34;
  printf("\nThe exchanged value is %x",swap_nibbles(ch));
  return 0;
}
```

How to scan a string till we hit a new line using scanf()?

Use `scanf("%[^\n]", address);`

Write pseudocode to compare versions (like 115.10.1 vs 115.11.5).

This question is also quite popular, because it has real practical uses, specially during patching when version comparison is required

The pseudocode is something like

```
while(both version strings are not NULL)
{
  // Extract the next version segment from Version string1.
  // Extract the next version segment from Version string2.

  // Compare these segments, if they are equal, continue
  // to check the next version segments.

  // If they are not equal, return appropriate code depending
  // on which segment is greater.
}
```

And here is some code in PL-SQL

```
---------------------------------------------------------------------
```

compare_versions()

Function to compare releases. Very useful when comparing file versions!

This function compare two versions in pl-
sql language. This function can compare
Versions like 115.10.1 vs. 115.10.2 (and say 115.10.2 is greater), 115.
10.1 vs. 115.10 (and say
115.10.1 is greater), 115.10 vs. 115.10 (and say both are equal)

```
function compare_releases(release_1 in varchar2, release_2 in varchar2)

return boolean is

  release_1_str varchar2(132);
  release_2_str varchar2(132);
  release_1_ver number;
  release_2_ver number;
  ret_status boolean := TRUE;

begin

  release_1_str := release_1 || '.';
  release_2_str := release_2 || '.';

  while release_1_str is not null or release_2_str is not null loop

    -- Parse out a current version segment from release_1

    if (release_1_str is null) then
       release_1_ver := 0;
    else
      release_1_ver := nvl(to_number(substr(release_1_str,1, instr(rele
ase_1_str,'.')-1)),-1);
      release_1_str := substr(release_1_str,instr(release_1_str,'.')+1)
;
    end if;

    -- Next parse out a version segment from release_2
    if (release_2_str is null) then
       release_2_ver := 0;
    else
       release_2_ver := nvl(to_number(substr(release_2_str,1, instr(rel
ease_2_str,'.')-1)),-1);
       release_2_str := substr(release_2_str,instr(release_2_str,'.')+1
);
    end if;


     if (release_1_ver > release_2_ver) then
        ret_status := FALSE;
        exit;
     elsif (release_1_ver < release_2_ver) then
        exit;
     end if;
     -- Otherwise continue to loop.
   end loop;

   return(ret_status);

end compare_releases;
```

How do you get the line numbers in C?

Use the following Macros

```
__FILE__ Source file name (string constant) format "patx.c"
__LINE__ Current source line number (integer)
__DATE__ Date compiled (string constant)format "Dec 14 1985"
__TIME__ Time compiled (string constant) format "15:24:26"
__TIMESTAMP__ Compile date/time (string constant)format "Tue Nov 19 11:
39:12 1997"
```

Usage example

```
static char stamp[] =    "***\nmodule " __FILE__    "\ncompiled " __TIM
ESTAMP__     "\n***";

...

int main()
{
   ...

   if ( (fp = fopen(fl,"r")) == NULL )
   {
      printf( "open failed, line %d\n%s\n",__LINE__, stamp );
      exit( 4 );
   }

   ...
}
```

And the output is something like

```
*** open failed, line 67
******
module myfile.c
compiled Mon Jan 15 11:15:56 1999
***
```

How to fast multiply a number by 7?

Try

```
(num<<3 - num)
```

This is same as

```
num*8 - num = num * (8-1) = num * 7
```

Write a simple piece of code to split a string at equal intervals

Suppose you have a big string

```
This is a big string which I want to split at equal intervals, without
caring about the words.
```

Now, to split this string say into smaller strings of 20 characters each, try this

```
#define maxLineSize 20

split(char *string)
{
  int i, length;
  char dest[maxLineSize + 1];

  i           = 0;
  length      = strlen(string);

  while((i+maxLineSize) <= length)
  {
     strncpy(dest, (string+i), maxLineSize);
     dest[maxLineSize - 1] = '\0';
     i = i + strlen(dest) - 1;
     printf("\nChunk : [%s]\n", dest);
  }

  strcpy(dest, (string + i));
  printf("\nChunk : [%s]\n", dest);
}
```

Is there a way to multiply matrices in lesser than o(n^3) time complexity?

Yes. Divide and conquer method suggests Strassen's matrix multiplication method to be used. If we follow this method, the time complexity is O(n^2.81) times rather O(n^3) times.

Here are some more details about this method.

Suppose we want to multiply two matrices of size N x N: for example A x B = C

```
[C11 C12]   [A11 A12] [B11 B12]
[C21 C22] = [A21 A22] [B21 B22]
```

Now, this guy called Strassen's somehow :) came up with a bunch of equations to calculate the 4 elements of the resultant matrix

```
C11 = a11*b11 + a12*b21
C12 = a11*b12 + a12*b22
C21 = a21*b11 + a22*b21
C22 = a21*b12 + a22*b22
```

If you are aware, the rudimentary matrix multiplication goes something like this

```
void matrix_mult()
{
  for (i = 1; i <= N; i++)
  {
     for (j = 1; j <= N; j++)
     {
        compute Ci,j;

     }
  }
}
```

So, essentially, a 2x2 matrix multiplication can be accomplished using 8 multiplications. And the complexity becomes

```
2^log 8 =2^3
```

Strassen showed that 2x2 matrix multiplication can be accomplished in 7 multiplications and 18 additions or subtractions. So now the complexity becomes

```
2^log7 =2^2.807
```

This is how he did it

```
P1 = (A11+ A22)(B11+B22)
P2 = (A21 + A22) * B11
P3 = A11 * (B12 - B22)
P4 = A22 * (B21 - B11)
P5 = (A11 + A12) * B22
P6 = (A21 - A11) * (B11 + B12)
P7 = (A12 - A22) * (B21 + B22)

C11 = P1 + P4 - P5 + P7
C12 = P3 + P5
C21 = P2 + P4
C22 = P1 + P3 - P2 + P6
```

Now, there is no need to memorize this stuff!

How do you find out if a machine is 32 bit or 64 bit?

One common answer is that all compilers keep the size of integer the same as the size of the register on a particular architecture. Thus, to know whether the machine is 32 bit or 64 bit, just see the size of integer on it.

I am not sure how true this is!
**

Write a program to have the output go two places at once (to the screen and to a file also) .

You can write a wrapper function for printf() which prints twice.

```
myprintf(...)
{
  // printf();         -> To screen.
  // write_to_file(); -> To file.
}
```

A command in shell, called tee does have this functionality!

Write code to round numbers.

Use something like

```
(int)(num < 0 ? (num - 0.5) : (num + 0.5))
```

How can we sum the digits of a given number in single statement?

Try something like this

```
# include<stdio.h>

void main()
{
  int num=123456;
  int sum=0;

  for(;num>0;sum+=num%10,num/=10);  // This is the "single line".

  printf("\nsum = [%d]\n", sum);
}
```

If there is a simpler way to do this, let me know!

Given two strings A and B, how would you find out if the characters in B were a subset of the characters in A?

Here is a simple, yet efficient C program to accomplish the same...

```
#include <stdio.h>
#include <conio.h>

int isSubset(char *a, char *b);

int main()
{
 char str1[]="defabc";
 char str2[]="abcfed";

 if(isSubset(str1, str2)==0)
 {
   printf("\nYes, characters in B=[%s] are a subset of characters in A=
[%s]\n",str2,str1);
 }
 else
 {
   printf("\nNo, characters in B=[%s] are not a subset of characters in
 A=[%s]\n",str2,str1);
 }

 getch();
 return(0);
}


// Function to check if characters in "b" are a subset
// of the characters in "a"

int isSubset(char *a, char *b)
{
 int letterPresent[256];
 int i;

 for(i=0; i<256; i++)
    letterPresent[i]=0;

 for(i=0; a[i]!='\0'; i++)
    letterPresent[a[i]]++;

 for(i=0; b[i]!='\0'; i++)
    if(!letterPresent[b[i]])
       return(1);

 return(0);
}
```

Write a program to merge two arrays in sorted order, so that if an integer is in both the arrays, it gets added into the final array only once.

Write a program to check if the stack grows up or down.

Try noting down the address of a local variable. Call another function with a local variable declared in it and check the address of that local variable and compare!

```
#include <stdio.h>
#include <stdlib.h>

void stack(int *local1);

int main()
{
  int local1;
  stack(&local1);
  exit(0);
}

void stack(int *local1)
{
   int local2;
   printf("\nAddress of first  local : [%u]", local1);
   printf("\nAddress of second local : [%u]", &local2);
   if(local1 < &local2)
   {
     printf("\nStack is growing downwards.\n");
   }
   else
   {
     printf("\nStack is growing upwards.\n");
   }
   printf("\n\n");
}
```

How to add two numbers without using the plus operator?

Actually,

```
SUM   = A XOR B
CARRY = A AND B
```

On a wicked note, you can add two numbers wihtout using the + operator as follows

```
a - (- b)
```

How to generate prime numbers? How to generate the next prime after a given prime?

This is a very vast subject. There are numerous methods to generate primes or to find out if a given number is a prime number or not. Here are a few of them. I strongly recommend you to search on the Internet for more elaborate information.

Brute Force
Test each number starting with 2 and continuing up to the number of primes we want to generate. We divide each numbr by all divisors upto the square root of that number. If no factors are found, its a prime.

Using only primes as divisors
Test each candidate only with numbers that have been already proven to be prime. To do so, keep a list of already found primes (probably using an array, a file or bit fields).

Test with odd candidates only
We need not test even candidates at all. We could make 2 a special case and just print it, not include it in the list of primes and start our candidate search with 3 and increment by 2 instead of one everytime.

Table method
Suppose we want to find all the primes between 1 and 64. We write out a table of these numbers, and proceed as follows. 2 is the first integer greater than 1, so its obviously prime. We now cross out all multiples of two. The next number we haven't crossed out is 3. We circle it and cross out all its multiples. The next non-crossed number is 5, sp we circle it and cross all its mutiples. We only have to do this for all numbers less than the square root of our upper limit, since any composite in the table must have atleast one factor less than the square root of the upper limit. Whats left after this process of elimination is all the prime numbers between 1 and 64.

Write a program to print numbers from 1 to 100 without using loops!

Another "Yeah, I am a jerk, kick me! kind of a question. I simply dont know why they ask these questions.

Nevertheless, for the benefit of mankind...

Method1 (Using recursion)

```
void printUp(int startNumber, int endNumber)
{
  if (startNumber > endNumber)
    return;
  printf("[%d]\n", startNumber++);
  printUp(startNumber, endNumber);
}
```

Method2 (Using goto)

```
void printUp(int startNumber, int endNumber)
{
start:

    if (startNumber > endNumber)
    {
       goto end;
    }
    else
    {
       printf("[%d]\n", startNumber++);
       goto start;
    }

end:
    return;
}
```

Write your own trim() or squeeze() function to remove the spaces from a string.

Here is one version...

```
#include <stdio.h>

char *trim(char *s);

int main(int argc, char *argv[])
{
  char str1[]=" Hello   I am Good ";
  printf("\n\nBefore trimming : [%s]", str1);
  printf("\n\nAfter trimming  : [%s]", trim(str1));

  getch();
}


// The trim() function...

char *trim(char *s)
{
    char *p, *ps;

    for (ps = p = s; *s != '\0'; s++)
    {
        if (!isspace(*s))
        {
            *p++ = *s;
        }
    }

    *p = '\0';

    return(ps);
}
```

And here is the output...

```
Before trimming : [ Hello   I am Good ]
After trimming  : [HelloIamGood]
```

Another version of this question requires one to reduce multiple spaces, tabs etc to single spaces...

Write your own random number generator function in C.
Write your own sqrt() function in C.

Trees

Write a C program to find the depth or height of a tree.

Here is some C code to get the height of the tree

```
tree_height(mynode *p)
{
   if(p==NULL)return(0);
   if(p->left){h1=tree_height(p->left);}
   if(p=>right){h2=tree_height(p->right);}
   return(max(h1,h2)+1);
}
```

The degree of the leaf is zero. The degree of a tree is the max of its element degrees. A binary tree of height n, h > 0, has at least h and at most (2^h -1) elements in it. The height of a binary tree that contains n, n>0, elements is at most n and atleast log(n+1) to the base 2.

Log(n+1) to the base 2 = h

n = (2^h - 1)

Write a C program to determine the number of elements (or size) in a tree.

Try this C program

```
int tree_size(struct node* node)
{
  if (node==NULL)
  {
    return(0);
  }
  else
  {
    return(tree_size(node->left) + tree_size(node->right) + 1);
  }
}
```

Write a C program to delete a tree (i.e, free up its nodes) .

Free up the nodes using Postorder traversal!.

Write C code to determine if two trees are identical.

Here is a C program using recursion

```
int identical(struct node* a, struct node* b)
{
  if (a==NULL && b==NULL){return(true);}
  else if (a!=NULL && b!=NULL)
  {
    return(a->data == b->data &&
           identical(a->left, b->left) &&
           identical(a->right, b->right));
  }
  else return(false);
}
```

Write a C program to find the mininum value in a binary search tree.

Here is some sample C code. The idea is to keep on moving till you hit the left most node in the tree

```
int minValue(struct node* node)
{
  struct node* current = node;

  while (current->left != NULL)
  {
    current = current->left;
  }

  return(current->data);
}
```

On similar lines, to find the maximum value, keep on moving till you hit the right most node of the tree.

Write a C program to compute the maximum depth in a tree?

Here is some C code...

```
int maxDepth(struct node* node)
{
  if (node==NULL)
  {
    return(0);
  }
  else
  {
    int leftDepth  = maxDepth(node->left);
    int rightDepth = maxDepth(node->right);
    if (leftDepth > rightDepth) return(leftDepth+1);
    else return(rightDepth+1);
  }
}
```

Write a C program to create a mirror copy of a tree (left nodes become right and right nodes become left)!

This C code will create a new mirror copy tree.

```
mynode *copy(mynode *root)
{
  mynode *temp;

  if(root==NULL)return(NULL);

  temp = (mynode *) malloc(sizeof(mynode));
  temp->value = root->value;

  temp->left  = copy(root->right);
  temp->right = copy(root->left);

  return(temp);
}
```

This code will will only print the mirror of the tree

```
void tree_mirror(struct node* node)
{
   struct node *temp;

   if (node==NULL)
   {
     return;
   }
   else
   {
      tree_mirror(node->left);
      tree_mirror(node->right);

      // Swap the pointers in this node
      temp = node->left;
      node->left = node->right;
      node->right = temp;
   }
}
```

Write C code to return a pointer to the nth node of an inorder traversal of a BST.

Here is a C program. Study it carefully, it has some Gotchas!

```
typedef struct node
{
  int value;
  struct node *left;
  struct node *right;
}mynode;

mynode *root;
static ctr;

void nthnode(mynode *root, int n, mynode **nthnode /* POINTER TO A POIN
TER! */);

int main()
{
  mynode *temp;
  root = NULL;

  // Construct the tree
  add(19);
  add(20);
  ...
  add(11);

  // Plain Old Inorder traversal
  // Just to see if the next function is really returning the nth node?
  inorder(root);
  // Get the pointer to the nth Inorder node
  nthinorder(root, 6, &temp);

  printf("\n[%d]\n, temp->value);
  return(0);
}

// Get the pointer to the nth inorder node in "nthnode"
void nthinorder(mynode *root, int n, mynode **nthnode)
{
  static whichnode;
  static found;

  if(!found)
  {
    if(root)
    {
       nthinorder(root->left, n , nthnode);
       if(++whichnode == n)
       {
          printf("\nFound %dth node\n", n);
          found = 1;
          *nthnode = root; // Store the pointer to the nth node.
        }
       nthinorder(root->right, n , nthnode);
    }
  }
}


inorder(mynode *root)
{
  // Plain old inorder traversal
}


// Function to add a new value to a Binary Search Tree
add(int value)
{
  mynode *temp, *prev, *cur;

  temp = malloc(sizeof(mynode));
  temp->value = value;
  temp->left  = NULL;
  temp->right = NULL;

  if(root == NULL)
  {
    root = temp;
  }
  else
  {
    prev = NULL;
    cur  = root;

    while(cur)
    {
      prev = cur;
      cur  = (value < cur->value)? cur->left : cur->right;
    }

    if(value > prev->value)
       prev->right = temp;
    else
       prev->left  = temp;
  }
}
```

There seems to be an easier way to do this, or so they say. Suppose each node also has a weight associated with it. This weight is the number of nodes below it and including itself. So, the root will have the highest weight (weight of its left subtree + weight of its right subtree + 1). Using this data, we can easily find the nth inorder node.

Note that for any node, the (weight of the leftsubtree of a node + 1) is its inorder rankin the tree!. Thats simply because of how the inorder traversal works (left->root->right). So calculate the rank of each node and you can get to the nth inorder node easily. But frankly speaking, I really dont know how this method is any simpler than the one I have presented above. I see more work to be done here (calculate thw weights, then calculate the ranks and then get to the nth node!).

Also, if (n > weight(root)), we can error out saying that this tree does not have the nth node you are looking for.

Write C code to implement the preorder(), inorder() and postorder() traversals. Whats their time complexities?

Here are the C program snippets to implement these traversals...

Preorder

```
preorder(mynode *root)
{
  if(root)
  {
    printf("Value : [%d]", root->value);
    preorder(root->left);
    preorder(root->right);
  }
}
```

Postorder

```
postorder(mynode *root)
{
  if(root)
  {
    postorder(root->left);
    postorder(root->right);
    printf("Value : [%d]", root->value);
  }
}
```

Inorder

```
inorder(mynode *root)
{
  if(root)
  {
    inorder(root->left);
    printf("Value : [%d]", root->value);
    inorder(root->right);
  }
}
```

Time complexity of traversals is O(n).

Write a C program to create a copy of a tree.

Here is a C program which does that...

```
mynode *copy(mynode *root)
{
  mynode *temp;

  if(root==NULL)return(NULL);

  temp = (mynode *) malloc(sizeof(mynode));
  temp->value = root->value;

  temp->left  = copy(root->left);
  temp->right = copy(root->right);

  return(temp);
}
```

Write C code to check if a given binary tree is a binary search tree or not?

Here is a C program which checks if a given tree is a Binary Search Tree or not...

```
int isThisABST(struct node* mynode)
{
    if (mynode==NULL) return(true);
    if (node->left!=NULL && maxValue(mynode->left) > mynode->data)
       return(false);
    if (node->right!=NULL && minValue(mynode->right) <= mynode->data)
       return(false);
    if (!isThisABST(node->left) || !isThisABST(node->right))
```

```
        return(false);

    return(true);
}
```

Write C code to implement level order traversal of a tree.

If this is the tree,

```
         1
    2          3
  5   6      7   8
```

its level order traversal would be

```
1 2 3 5 6 7 8
```

You need to use a queue to do this kind of a traversal

```
Let t be the tree root.
while (t != null)
{
  visit t and put its children on a FIFO queue;
  remove a node from the FIFO queue and
  call it t;
  // Remove returns null when queue is empty
}
```

Pseudocode

```
Level_order_traversal(p)
{
   while(p)
   {
     Visit(p);
     If(p->left)Q.Add(p->left);
     If(p->right)Q.Add(p->right);
     Delete(p);
   }
}
```

Here is some C code (working :))..

```
#include <stdio.h>

typedef struct node
{
  int value;
  struct node *right;
  struct node *left;
}mynode;

mynode *root;

add_node(int value);

void levelOrderTraversal(mynode *root);

int main(int argc, char* argv[])
{
  root = NULL;

  add_node(5);
  add_node(1);
  add_node(-20);
  add_node(100);
  add_node(23);
  add_node(67);
  add_node(13);


  printf("\n\n\nLEVEL ORDER TRAVERSAL\n\n");
  levelOrderTraversal(root);

  getch();
}

// Function to add a new node...
add_node(int value)
{
   mynode *prev, *cur, *temp;

   temp        = (mynode *) malloc(sizeof(mynode));
   temp->value = value;
   temp->right = NULL;
   temp->left  = NULL;

   if(root==NULL)
   {
     printf("\nCreating the root..\n");
     root = temp;
     return;
   }

   prev=NULL;
   cur=root;

   while(cur!=NULL)
   {
      prev=cur;
      cur=(value<cur->value)?cur->left:cur->right;
   }

   if(value < prev->value)
     prev->left=temp;
   else
```

```
     prev->right=temp;
}



// Level order traversal..
void levelOrderTraversal(mynode *root)
{
  mynode *queue[100] = {(mynode *)0}; // Important to initialize!
  int size = 0;
  int queue_pointer = 0;

  while(root)
  {
      printf("[%d] ", root->value);

      if(root->left)
      {
        queue[size++] = root->left;
      }

      if(root->right)
      {
        queue[size++] = root->right;
      }

      root = queue[queue_pointer++];
  }
}
```

Write a C program to delete a node from a Binary Search Tree?

The node to be deleted might be in the following states

- The node does not exist in the tree - In this case you have nothing to delete.
- The node to be deleted has no children - The memory occupied by this node must be freed and either the left link or the right link of the parent of this node must be set to NULL.
- The node to be deleted has exactly one child - We have to adjust the pointer of the parent of the node to be deleted such that after deletion it points to the child of the node being deleted.
- The node to be deleted has two children - We need to find the inorder successor of the node to be deleted. The data of the inorder successor must be copied into the node to be deleted and a pointer should be setup to the inorder successor. This inorder successor would have one or zero children. This node should be deleted using the same procedure as for deleting a one child or a zero child node. Thus the whole logic of deleting a node with two children is to locate the inorder successor, copy its data and reduce the problem to a simple deletion of a node with one or zero children.

Here is some C code for these two situations

```
Situation 1


     100 (parent)

   50 (cur == psuc)

20    80 (suc)

         90

       85  95


Situation 2

           100 (parent)

        50 (cur)

     20    90

         80

       70 (suc)

          75

        72   76


mynode *delete(int item, mynode *head)
{
     mynode *cur, *parent, *suc, *psuc, q;

     if(head->left==NULL){printf("\nEmpty tree!\n");return(head);}

     parent = head;
     cur    = head->left;

     while(cur!=NULL && item != cur->value)
     {
         parent  =  cur;
         cur     =  (item < cur->next)? cur->left:cur->right;
     }

     if(cur == NULL)
     {
        printf("\nItem to be deleted not found!\n");
        return(head);
     }


     // Item found, now delete it
     if(cur->left == NULL)
         q = cur->right;
     else if(cur->right == NULL)
         q = cur->left;
     else
     {
           // Obtain the inorder successor and its parent

           psuc = cur;
           cur  = cur->left;

           while(suc->left!=NULL)
           {
               psuc = suc;
               suc  = suc->left;
           }


           if(cur==psuc)
           {
                // Situation 1

                suc->left = cur->right;
           }
           else
           {
                // Situation 2

                suc->left  = cur->left;
                psuc->left = suc->right;
                suc->right = cur->right;
           }

           q = suc;

     }


     // Attach q to the parent node

     if(parent->left == cur)
        parent->left=q;
     else
        parent->rlink=q;

     freeNode(cur);

     return(head);
}
```

Write C code to search for a value in a binary search tree (BST).

Here are a few C programs....

```
mynode *search(int value, mynode *root)
{
   while(root!=NULL && value!=root->value)
   {
     root = (value < root->value)?root->left:root->right;
   }

   return(root);
}
```

Here is another way to do the same

```
mynode *recursive_search(int item, mynode *root)
{
  if(root==NULL || item == root->value){return(root);}
  if(item<root->info)return{recursive_search(item, root->left);}
  return{recursive_search(item, root->right);}
}
```

Write C code to count the number of leaves in a tree.

Here is a C program to do the same...

```
void count_leaf(mynode *root)
{
   if(root!=NULL)
   {
      count_leaf(root->left);

      if(root->left == NULL && root->right==NULL)
      {
        // This is a leaf!
        count++;
      }

      count_leaf(root->right);
   }
}
```

Write C code for iterative preorder, inorder and postorder tree traversals.

Here is a complete C program which prints a BST using both recursion and iteration. The best way to understand these algorithms is to get a pen and a paper and trace out the traversals (with the stack or the queue) alongside. Dont even try to memorize these algorithms!

```
#include <stdio.h>

typedef struct node
{
  int value;
  struct node *right;
  struct node *left;
}mynode;

mynode *root;

add_node(int value);
void postorder(mynode *root);
void inorder(mynode *root);
void preorder(mynode *root);

void iterativePreorder(mynode *root);
void iterativeInorder (mynode *root);
void iterativePostorder(mynode *root);


int main(int argc, char* argv[])
{
  root = NULL;

  add_node(5);
  add_node(1);
  add_node(-20);
  add_node(100);
  add_node(23);
  add_node(67);
  add_node(13);

  printf("\nPreorder (R)    : ");
  preorder(root);
  printf("\nPreorder (I)    : ");
  iterativePreorder(root);

  printf("\n\nPostorder (R)   : ");
  postorder(root);
  printf("\nPostorder (R)   : ");
  iterativePostorder(root);


  printf("\n\nInorder (R)     : ");
  inorder(root);
  printf("\nInorder (I)     : ");
  iterativeInorder(root);

}

// Function to add a new node to the BST
add_node(int value)
{
   mynode *prev, *cur, *temp;

   temp        = (mynode *) malloc(sizeof(mynode));
   temp->value = value;
   temp->right = NULL;
   temp->left  = NULL;

   if(root==NULL)
   {
     printf("\nCreating the root..\n");
     root = temp;
     return;
   }

   prev=NULL;
   cur=root;

   while(cur!=NULL)
   {
      prev=cur;
      cur=(value<cur->value)?cur->left:cur->right;
   }

   if(value < prev->value)
     prev->left=temp;
   else
     prev->right=temp;
}


// Recursive Preorder
void preorder(mynode *root)
{
  if(root)
  {
    printf("[%d] ", root->value);
    preorder(root->left);
    preorder(root->right);
  }
}

// Iterative Preorder
void iterativePreorder(mynode *root)
{
  mynode *save[100];
  int top = 0;

  if (root == NULL)
  {
    return;
  }

  save[top++] = root;
  while (top != 0)
  {
    root = save[--top];

    printf("[%d] ", root->value);

    if (root->right != NULL)
      save[top++] = root->right;
    if (root->left != NULL)
      save[top++] = root->left;
  }
}

// Recursive Postorder
void postorder(mynode *root)
{
  if(root)
  {
    postorder(root->left);
    postorder(root->right);
    printf("[%d] ", root->value);
  }
}

// Iterative Postorder
void iterativePostorder(mynode *root)
{
  struct
  {
    mynode *node;
    unsigned vleft :1;   // Visited left?
    unsigned vright :1;  // Visited right?
  }save[100];

  int top = 0;

  save[top++].node = root;

  while ( top != 0 )
  {
      /* Move to the left subtree if present and not visited */
      if(root->left != NULL && !save[top].vleft)
      {
          save[top].vleft = 1;
          save[top++].node = root;
          root = root->left;
          continue;
      }

      /* Move to the right subtree if present and not visited */
      if(root->right != NULL && !save[top].vright )
      {
          save[top].vright = 1;
          save[top++].node = root;
          root = root->right;
          continue;
      }

      printf("[%d] ", root->value);

      /* Clean up the stack */
      save[top].vleft = 0;
      save[top].vright = 0;

      /* Move up */
      root = save[--top].node;
   }
}


// Recursive Inorder
void inorder(mynode *root)
{
  if(root)
  {
    inorder(root->left);
    printf("[%d] ", root->value);
    inorder(root->right);
  }
}


// Iterative Inorder..
void iterativeInorder (mynode *root)
{
  mynode *save[100];
  int top = 0;

  while(root != NULL)
  {
      while (root != NULL)
      {
           if (root->right != NULL)
           {
             save[top++] = root->right;
           }
           save[top++] = root;
           root = root->left;
      }

      root = save[--top];
      while(top != 0 && root->right == NULL)
      {
          printf("[%d] ", root->value);
          root = save[--top];
      }

      printf("[%d] ", root->value);
      root = (top != 0) ? save[--top] : (mynode *) NULL;
  }
}
```

And here is the output...

```
Creating the root..

Preorder (R)     : [5] [1] [-20] [100] [23] [13] [67]
Preorder (I)     : [5] [1] [-20] [100] [23] [13] [67]

Postorder (R)    : [-20] [1] [13] [67] [23] [100] [5]
Postorder (R)    : [-20] [1] [13] [67] [23] [100] [5]

Inorder (R)      : [-20] [1] [5] [13] [23] [67] [100]
Inorder (I)      : [-20] [1] [5] [13] [23] [67] [100]
```

Can you construct a tree using postorder and preorder traversal?

No

Consider 2 trees below

```
Tree1

  a
b

Tree 2

a
  b

preorder  = ab
postorder = ba
```

Preorder and postorder do not uniquely define a binary tree. Nor do preorder and level order (same example). Nor do postorder and level order.

Construct a tree given its inorder and preorder traversal strings. Similarly construct a tree given its inorder and post order traversal strings.

For Inorder And Preorder traversals

```
inorder  = g d h b e i a f j c
preorder = a b d g h e i c f j
```

Scan the preorder left to right using the inorder sequence to separate left and right subtrees. For example, "a" is the root of the tree; "gdhbei" are in the left subtree; "fjc" are in the right subtree. "b" is the next root; "gdh" are in the left subtree; "ei" are in the right subtree. "d" is the next root; "g" is in the left subtree; "h" is in the right subtree.

For Inorder and Postorder traversals

Scan postorder from right to left using inorder to separate left and right subtrees.

```
inorder   = g d h b e i a f j c
postorder = g h d i e b j f c a
```

Tree root is "a"; "gdhbei" are in left subtree; "fjc" are in right subtree.

For Inorder and Levelorder traversals

Scan level order from left to right using inorder to separate left and right subtrees.

```
inorder     = g d h b e i a f j c
level order = a b c d e f g h i j
```

Tree root is "a"; "gdhbei" are in left subtree; "fjc" are in right subtree.

Here is some working code which creates a tree out of the Inorder and Postorder traversals. Note that here the tree has been represented as an array. This really simplifies the whole implementation.

Converting a tree to an array is very easy

Suppose we have a tree like this

```
      A
  B       C
 D E     F G
```

The array representation would be

```
a[1] a[2] a[3] a[4] a[5] a[6] a[7]
  A    B    C    D    E    F    G
```

That is, for every node at position j in the array, its left child will be stored at position (2*j) and right child at (2*j + 1). The root starts at position 1.

```
// CONSTRUCTING A TREE GIVEN THE INORDER AND PREORDER SEQUENCE

#include<stdio.h>
```

```
#include<string.h>
#include<ctype.h>

/*-------------------------------------------------------------
* Algorithm
*
* Inorder And Preorder
* inorder = g d h b e i a f j c
* preorder = a b d g h e i c f j
* Scan the preorder left to right using the inorder to separate left
* and right subtrees. a is the root of the tree; gdhbei are in the
* left subtree; fjc are in the right subtree.
*------------------------------------------------------------*/

static char io[]="gdhbeiafjc";
static char po[]="abdgheicfj";
static char t[100][100]={'\0'};  //This is where the final tree will be
 stored
static int hpos=0;

void copy_str(char dest[], char src[], int pos, int start, int end);
void print_t();

int main(int argc, char* argv[])
{
  int i,j,k;
  char *pos;
  int posn;

  // Start the tree with the root and its
  // left and right elements to start off

  for(i=0;i<strlen(io);i++)
  {
     if(io[i]==po[0])
     {
        copy_str(t[1],io,1,i,i);             // We have the root here
        copy_str(t[2],io,2,0,i-1);           // Its left subtree
        copy_str(t[3],io,3,i+1,strlen(io));  // Its right subtree
        print_t();
     }
  }

  // Now construct the remaining tree
  for(i=1;i<strlen(po);i++)
  {
     for(j=1;j<=hpos;j++)
     {
        if((pos=strchr((const char *)t[j],po[i]))!=(char *)0 && strlen(
t[j])!=1)
        {
            for(k=0;k<strlen(t[j]);k++)
            {
               if(t[j][k]==po[i]){posn=k;break;}
            }
            printf("\nSplitting [%s] for po[%d]=[%c] at %d..\n", t[j],i
,po[i],posn);
```

```
            copy_str(t[2*j],t[j],2*j,0,posn-1);
            copy_str(t[2*j+1],t[j],2*j+1,posn+1,strlen(t[j]));
            copy_str(t[j],t[j],j,posn,posn);
            print_t();
        }
     }
  }
}

// This function is used to split a string into three seperate strings
// This is used to create a root, its left subtree and its right subtre
e

void copy_str(char dest[], char src[], int pos, int start, int end)
{
  char mysrc[100];
  strcpy(mysrc,src);
  dest[0]='\0';
  strncat(dest,mysrc+start,end-start+1);
  if(pos>hpos)hpos=pos;
}


void print_t()
{
  int i;
  for(i=1;i<=hpos;i++)
  {
    printf("\nt[%d] = [%s]", i, t[i]);
  }
  printf("\n");
}
```

Find the closest ancestor of two nodes in a tree.

Here is some working C code...

```
#include <stdio.h>

typedef struct node
{
  int value;
  struct node *right;
  struct node *left;
}mynode;

mynode *root;

mynode *add_node(int value);
void levelOrderTraversal(mynode *root);
mynode *closestAncestor(mynode* root, mynode* p, mynode* q);
```

```
int main(int argc, char* argv[])
{
  mynode *node_pointers[7], *temp;
  root = NULL;

  // Create the BST.
  // Store the node pointers to use later...
  node_pointers[0] = add_node(5);
  node_pointers[1] = add_node(1);
  node_pointers[2] = add_node(-20);
  node_pointers[3] = add_node(100);
  node_pointers[4] = add_node(23);
  node_pointers[5] = add_node(67);
  node_pointers[6] = add_node(13);


  printf("\n\n\nLEVEL ORDER TRAVERSAL\n\n");
  levelOrderTraversal(root);


  // Calculate the closest ancestors of a few nodes..

  temp = closestAncestor(root, node_pointers[5], node_pointers[6]);
  printf("\n\nClosest ancestor of [%d] and [%d] is [%d]\n\n",
         node_pointers[5]->value,
         node_pointers[6]->value,
         temp->value);


  temp = closestAncestor(root, node_pointers[2], node_pointers[6]);
  printf("\n\nClosest ancestor of [%d] and [%d] is [%d]\n\n",
         node_pointers[2]->value,
         node_pointers[6]->value,
         temp->value);


  temp = closestAncestor(root, node_pointers[4], node_pointers[5]);
  printf("\n\nClosest ancestor of [%d] and [%d] is [%d]\n\n",
         node_pointers[4]->value,
         node_pointers[5]->value,
         temp->value);


  temp = closestAncestor(root, node_pointers[1], node_pointers[3]);
  printf("\n\nClosest ancestor of [%d] and [%d] is [%d]\n\n",
         node_pointers[1]->value,
         node_pointers[3]->value,
         temp->value);


  temp = closestAncestor(root, node_pointers[2], node_pointers[6]);
  printf("\n\nClosest ancestor of [%d] and [%d] is [%d]\n\n",
         node_pointers[2]->value,
         node_pointers[6]->value,
         temp->value);
```

```
}


// Function to add a new node to the tree..
mynode *add_node(int value)
{
   mynode *prev, *cur, *temp;

   temp        = (mynode *) malloc(sizeof(mynode));
   temp->value = value;
   temp->right = NULL;
   temp->left  = NULL;

   if(root==NULL)
   {
     printf("\nCreating the root..\n");
     root = temp;
     return;
   }

   prev=NULL;
   cur=root;

   while(cur!=NULL)
   {
      prev=cur;
      cur=(value<cur->value)?cur->left:cur->right;
   }

   if(value < prev->value)
     prev->left=temp;
   else
     prev->right=temp;

   return(temp);

}



// Level order traversal..
void levelOrderTraversal(mynode *root)
{
  mynode *queue[100] = {(mynode *)0};
  int size = 0;
  int queue_pointer = 0;

  while(root)
  {
      printf("[%d] ", root->value);

      if(root->left)
      {
        queue[size++] = root->left;
      }

      if(root->right)
      {
        queue[size++] = root->right;
      }

      root = queue[queue_pointer++];
  }
}


// Function to find the closest ancestor...
mynode *closestAncestor(mynode* root, mynode* p, mynode* q)
{
   mynode *l, *r, *tmp;

   if(root == NULL)
   {
      return(NULL);
   }

   if(root->left==p || root->right==p || root->left==q || root-
>right==q)
   {
     return(root);
   }
   else
   {
      l = closestAncestor(root->left, p, q);
      r = closestAncestor(root->right, p, q);

      if(l!=NULL && r!=NULL)
      {
        return(root);
      }
      else
      {
         tmp = (l!=NULL) ? l : r;
         return(tmp);
      }
   }
}
```

Here is the tree for you to visualize...

```
                        5 (node=0)

       1 (node=1)                    100 (node=3)

-20 (node=2)                  23 (node=4)

                   13 (node=5)   67 (node=6)
```

Here is the output...

```
LEVEL ORDER TRAVERSAL
```

```
[5] [1] [100] [-20] [23] [13] [67]


Closest ancestor of [67] and [13] is [23]

Closest ancestor of [-20] and [13] is [5]

Closest ancestor of [23] and [67] is [100]

Closest ancestor of [1] and [100] is [5]

Closest ancestor of [-20] and [13] is [5]
```

Given an expression tree, evaluate the expression and obtain a paranthesized form of the expression.

The code below prints the paranthesized form of a tree.

```
infix_exp(p)
{
   if(p)
   {
      printf("(");
      infix_exp(p->left);
      printf(p->data);
      infix_exp(p->right);
      printf(")");
   }
}
```

Creating a binary tree for a postfix expression

```
mynode *create_tree(char postfix[])
{
   mynode *temp, *st[100];
   int i,k;
   char symbol;

   for(i=k=0; (symbol = postfix[i])!='\0'; i++)
   {
        temp = (mynode *) malloc(sizeof(struct node));
        temp->value = symbol;
        temp->left = temp->right = NULL;

        if(isalnum(symbol))
              st[k++] = temp;
        else
        {
              temp->right = st[--k];
              temp->left  = st[--k];
              st[k++]     = temp;
        }
   }
   return(st[--k]);
}
```

Evaluate a tree

```
float eval(mynode *root)
{
   float num;

   switch(root->value)
   {
        case '+' : return(eval(root->left) + eval(root->right)); break;
        case '-' : return(eval(root->left) - eval(root->right)); break;
        case '/' : return(eval(root->left) / eval(root->right)); break;
        case '*' : return(eval(root->left) * eval(root->right)); break;
        case '$' : return(eval(root->left) $ eval(root->right)); break;
        default  : if(isalpha(root->value))
                   {
                      printf("%c = ", root->value);
                      scanf("%f", &num);
                      return(num);
                   }
                   else
                   {
                      return(root->value - '0');
                   }

   }
}
```

How do you convert a tree into an array?

The conversion is based on these rules

```
If i > 1, i/2 is the parent
If 2*i > n, then there is no left child, else 2*i is the left child.
If (2*i + 1) > n, then there is no right child, else (2*i + 1) is the r
ight child.
```

Converting a tree to an array is very easy

Suppose we have a tree like this

```
      A
  B       C
 D E     F G
```

```
The array representation would be
```

```
a[1] a[2] a[3] a[4] a[5] a[6] a[7]
  A    B    C    D    E    F    G
```

That is, for every node at position i in the array, its left child will be stored at position (2*i) and right child at (2*i + 1). The root starts at position 1.

What is an AVL tree?

AVL trees are self-adjusting, height-balanced binary search trees and are named after the inventors: Adelson-Velskii and Landis. A balanced binary search tree has O(log n) height and hence O(log n) worst case search and insertion times. However, ordinary binary search trees have a bad worst case. When sorted data is inserted, the binary search tree is very unbalanced, essentially more of a linear list, with O(n) height and thus O(n) worst case insertion and lookup times. AVL trees overcome this problem.

An AVL tree is a binary search tree in which every node is height balanced, that is, the difference in the heights of its two subtrees is at most 1. The balance factor of a node is the height of its right subtree minus the height of its left subtree (right minus left!). An equivalent definition, then, for an AVL tree is that it is a binary search tree in which each node has a balance factor of -1, 0, or +1. Note that a balance factor of -1 means that the subtree is left-heavy, and a balance factor of +1 means that the subtree is right-heavy. Each node is associated with a Balancing factor.

```
Balance factor of each node = height of right subtree at that node -
 height of left subtree at that node.
```

Please be aware that we are talking about the height of the subtrees and not the weigths of the subtrees. This is a very important point. We are talking about the height!.

Here is some recursive, working! C code that sets the Balance factor for all nodes starting from the root....

```
#include <stdio.h>

typedef struct node
{
  int value;
  int visited;
  int bf;
  struct node *right;
  struct node *left;
}mynode;

mynode *root;

mynode *add_node(int value);
void levelOrderTraversal(mynode *root);
int setbf(mynode *p);


// The main function
int main(int argc, char* argv[])
{
  root = NULL;

  // Construct the tree..
  add_node(5);
  add_node(1);
  add_node(-20);
  add_node(100);
  add_node(23);
  add_node(67);
  add_node(13);

  // Set the balance factors
  setbf(root);

  printf("\n\n\nLEVEL ORDER TRAVERSAL\n\n");
  levelOrderTraversal(root);
  getch();
}

// Function to add a new node to the tree...
mynode *add_node(int value)
{
   mynode *prev, *cur, *temp;

   temp        = (mynode *) malloc(sizeof(mynode));
   temp->value = value;
   temp->visited = 0;
   temp->bf = 0;
   temp->right = NULL;
   temp->left  = NULL;

   if(root==NULL)
   {
     //printf("\nCreating the root..\n");
     root = temp;
     return;
   }

   prev=NULL;
   cur=root;

   while(cur!=NULL)
   {
      prev=cur;
      cur=(value<cur->value)?cur->left:cur->right;
   }

   if(value < prev->value)
     prev->left=temp;
   else
     prev->right=temp;

   return(temp);

}


// Recursive function to set the balancing factor
// of each node starting from the root!
int setbf(mynode *p)
{
   int templ, tempr;
   int count;
   count = 1;

   if(p == NULL)
   {
     return(0);
   }
   else
   {
       templ = setbf(p->left);
       tempr = setbf(p->right);

       if(templ < tempr)
         count = count + tempr;
       else
         count = count + templ;
   }

   // Set the nodes balancing factor.
   printf("\nNode = [%3d], Left sub-tree height = [%1d], Right sub-
tree height = [%1d], BF = [%1d]\n",
          p->value, templ, tempr, (tempr - templ));
   p->bf = tempr - templ;
   return(count);
}



// Level order traversal..
void levelOrderTraversal(mynode *root)
{
  mynode *queue[100] = {(mynode *)0};
  int size = 0;
  int queue_pointer = 0;

  while(root)
  {
      printf("\n[%3d] (BF : %3d) ", root->value, root->bf);

      if(root->left)
      {
        queue[size++] = root->left;
      }

      if(root->right)
      {
        queue[size++] = root->right;
      }

      root = queue[queue_pointer++];
  }
}
```

And here is the output...

```
Node = [-20], Left sub-tree height = [0], Right sub-
tree height = [0], BF = [0]
Node = [  1], Left sub-tree height = [1], Right sub-
tree height = [0], BF = [-1]
Node = [ 13], Left sub-tree height = [0], Right sub-
tree height = [0], BF = [0]
Node = [ 67], Left sub-tree height = [0], Right sub-
tree height = [0], BF = [0]
Node = [ 23], Left sub-tree height = [1], Right sub-
tree height = [1], BF = [0]
Node = [100], Left sub-tree height = [2], Right sub-
tree height = [0], BF = [-2]
Node = [  5], Left sub-tree height = [2], Right sub-
tree height = [3], BF = [1]


LEVEL ORDER TRAVERSAL

[  5] (BF :   1)
[  1] (BF :  -1)
[100] (BF :  -2)
[-20] (BF :   0)
[ 23] (BF :   0)
[ 13] (BF :   0)
[ 67] (BF :   0)
```

Here is the tree which we were dealing with above

```
            5
       1         100
-20           23
           13   67
```

After insertion, the tree might have to be readjusted as needed in order to maintain it as an AVL tree. A node with balance factor -2 or 2 is considered unbalanced and requires rebalancing the tree. The balance factor is either stored directly at each node or computed from the heights of the subtrees, possibly stored at nodes. If, due to an instertion or deletion, the tree becomes unbalanced, a corresponding left rotation or a right rotation is performed on that tree at a particular node. A balance factor > 1 requires a left rotation (i.e. the right subtree is heavier than the left subtree) and a balance factor < -1 requires a right rotation (i.e. the left subtree is heavier than the right subtree).

Here is some pseudo code to demonstrate the two types of rotations...

Left rotation

```
BEFORE

        0 (par)

   0          0 (p)

         0         0 (tmp)

       0   0     0   0
                (a) (b)
```

Here we left rotate the tree around node p

```
tmp         = p->right;
p->right    = tmp->left;
tmp->left   = p;

if(par)
{
   if(p is the left child of par)
   {
     par->left=tmp;
   }
   else
   {
     par->right=tmp;
   }
}
else
{
   root=tmp;
}

// Reclaculate the balance factors
setbf(root);
```

```
AFTER
        0 (par)

   0             0
               (tmp)

          0           0
         (p)         (b)

      0      0
            (a)

   0     0
```

Right rotation

```
BEFORE

        0 (par)

   0            0 (p)

         0 (tmp)      0

      0    0       0     0
     (a)  (b)
```

Here we right rotate the tree around node p

```
tmp         = p->left;
p->left     = tmp->right;
tmp->right  = p;

if(par)
{
   if(p is the left child of par)
   {
     par->left=tmp;
   }
   else
   {
     par->right=tmp;
   }
}
else
{
   root=tmp;
}

// Recalculate the balancing factors...
setbf(root);
```

```
AFTER

        0 (par)

   0            0 (tmp)

          0           0
         (a)         (p)

                 0         0
                (b)

                        0     0
```

How many different trees can be constructed using n nodes?

Good question!

Its

```
2^n - n
```

So, if there are 10 nodes, you will have (1024 - 10) = 1014 different trees!! Confirm it yourself with a small number if you dont believe the formula.

A full N-ary tree has M non-leaf nodes, how many leaf nodes does it have?

Use Geometric progression.

```
M + (N ^ (n-1)) = (1 - (N ^ n)) / (1 - N)

Here (N ^ (n-1)) is the number of leaf-nodes.

Solving for this leads to the answer

Leaf nodes = M * (N - 1) + 1
```

Suppose you have a 3-ary tree

```
                A
      B         C         D
   E  F  G   H  I  J   K  L  M
```

So, here M=4 and N=3. So using the formula above,

```
Leaf nodes = M * (N - 1) + 1 = 4 * (3 - 1) + 1 = 9
```

Implement Breadth First Search (BFS) and Depth First Search (DFS).

Depth first search (DFS)

Depth First Search (DFS) is a generalization of the preorder traversal. Starting at some arbitrarily chosen vertex v, we mark v so that we know we've visited it, process v, and then recursively traverse all unmarked vertices adjacent to v (v will be a different vertex with every new method call). When we visit a vertex in which all of its neighbors have been visited, we return to its calling vertex, and visit one of its unvisited neighbors, repeating the recursion in the same manner. We continue until we have visited all of the starting vertex's neighbors, which means that we're done. The recursion (stack) guides us through the graph.

```
public void depthFirstSearch(Vertex v)
{
        v.visited = true;

        // print the node

        for(each vertex w adjacent to v)
        {
           if(!w.visited)
           {
              depthFirstSearch(w);
```

```
            }
        }
}
```

Here is some working C code for a DFS on a BST..

```
#include <stdio.h>

typedef struct node
{
  int value;
  int visited;
  struct node *right;
  struct node *left;
}mynode;

mynode *root;

mynode *add_node(int value);
void treeDFS(mynode *root);


int main(int argc, char* argv[])
{
  root = NULL;

  // Construct the tree..
  add_node(5);
  add_node(1);
  add_node(-20);
  add_node(100);
  add_node(23);
  add_node(67);
  add_node(13);

  // Do a DFS..
  printf("\n\nDFS : ");
  treeDFS(root);

  getch();
}

// Function to add a new node to the tree...
mynode *add_node(int value)
{
   mynode *prev, *cur, *temp;

   temp        = (mynode *) malloc(sizeof(mynode));
   temp->value = value;
   temp->visited = 0;
   temp->right = NULL;
   temp->left  = NULL;

   if(root==NULL)
   {
     printf("\nCreating the root..\n");
```

```
      root = temp;
      return;
    }

    prev=NULL;
    cur=root;

    while(cur!=NULL)
    {
       prev=cur;
       cur=(value<cur->value)?cur->left:cur->right;
    }

    if(value < prev->value)
      prev->left=temp;
    else
      prev->right=temp;

    return(temp);

}


// DFS..
void treeDFS(mynode *root)
{
     printf("[%d] ", root->value);
     root->visited = 1;

     if (root->left)
     {
       if(root->left->visited==0)
       {
         treeDFS(root->left);
       }
     }

     if (root->right)
     {
       if(root->right->visited==0)
       {
         treeDFS(root->right);
       }
     }
}
```

Breadth First Search

Breadth First Search (BFS) searches the graph one level (one edge away from the starting vertex) at a time. In this respect, it is very similar to the level order traversal that we discussed for trees. Starting at some arbitrarily chosen vertex v, we mark v so that we know we've visited it, process v, and then visit and process all of v's neighbors. Now that we've visited and processed all of v's neighbors, we need to visit and process all of v's neighbors neighbors. So we go to the first neighbor we visited and visit all of its neighbors, then the second neighbor we visited, and so on. We continue this process until we've visited all vertices in the graph. We don't use recursion in a BFS because we don't want to traverse recursively. We want to traverse one level at a time. So imagine that you visit a vertex v, and then you visit all of v's neighbors w. Now you need to visit each w's neighbors. How are you going to remember all of your w's so that you can go back and visit their neighbors? You're already marked and processed all of the w's. How are you going to find each w's neighbors if you don't remember where the w's are? After all, you're not using recursion, so there's no stack to keep track of them. To perform a BFS, we use a queue. Every time we visit vertex w's neighbors, we dequeue w and enqueue w's neighbors. In this way, we can keep track of which neighbors belong to which vertex. This is the same technique that we saw for the level-order traversal of a tree. The only new trick is that we need to makr the verticies, so we don't visit them more than once -- and this isn't even new, since this technique was used for the blobs problem during our discussion of recursion.

```
public void breadthFirstSearch(vertex v)
{
   Queue q = new Queue();

   v.visited = true;
   q.enQueue(v);

   while( !q.isEmpty() )
   {
       Vertex w = (Vertex)q.deQueue();

       // Print the node.

       for(each vertex x adjacent to w)
       {
           if( !x.visited )
           {
              x.visited = true;
              q.enQueue(x);
           }
        }
    }
}
```

BFS traversal can be used to produce a tree from a graph.

Here is some C code which does a BFS (level order traversal) on a BST...

```
#include <stdio.h>

typedef struct node
{
  int value;
  struct node *right;
  struct node *left;
}mynode;

mynode *root;
```

```
add_node(int value);

void levelOrderTraversal(mynode *root);

int main(int argc, char* argv[])
{
  root = NULL;

  add_node(5);
  add_node(1);
  add_node(-20);
  add_node(100);
  add_node(23);
  add_node(67);
  add_node(13);


  printf("\n\n\nLEVEL ORDER TRAVERSAL\n\n");
  levelOrderTraversal(root);

  getch();
}

// Function to add a new node...
add_node(int value)
{
   mynode *prev, *cur, *temp;

   temp        = (mynode *) malloc(sizeof(mynode));
   temp->value = value;
   temp->right = NULL;
   temp->left  = NULL;

   if(root==NULL)
   {
     printf("\nCreating the root..\n");
     root = temp;
     return;
   }

   prev=NULL;
   cur=root;

   while(cur!=NULL)
   {
      prev=cur;
      cur=(value<cur->value)?cur->left:cur->right;
   }

   if(value < prev->value)
     prev->left=temp;
   else
     prev->right=temp;
}
```

```
// Level order traversal..
void levelOrderTraversal(mynode *root)
{
  mynode *queue[100] = {(mynode *)0}; // Important to initialize!
  int size = 0;
  int queue_pointer = 0;

  while(root)
  {
      printf("[%d] ", root->value);

      if(root->left)
      {
        queue[size++] = root->left;
      }

      if(root->right)
      {
        queue[size++] = root->right;
      }

      root = queue[queue_pointer++];
  }
}
```

Write pseudocode to add a new node to a Binary Search Tree (BST).

Here is a C code to construct a BST right from scratch...

```
#include <stdio.h>

typedef struct node
{
  int value;
  struct node *right;
  struct node *left;
}mynode;

mynode *root;

add_node(int value);
void postorder(mynode *root);
void inorder(mynode *root);
void preorder(mynode *root);

int main(int argc, char* argv[])
{
  root = NULL;

  add_node(5);
  add_node(1);
  add_node(-20);
  add_node(100);
  add_node(23);
  add_node(67);
  add_node(13);

  printf("\nPreorder    : ");
  preorder(root);
  printf("\n\nPostorder : ");
  postorder(root);
  printf("\n\nInorder   : ");
  inorder(root);

  return(0);
}

// Function to add a new node...
add_node(int value)
{
   mynode *prev, *cur, *temp;

   temp        = (mynode *) malloc(sizeof(mynode));
   temp->value = value;
   temp->right = NULL;
   temp->left  = NULL;

   if(root==NULL)
   {
     printf("\nCreating the root..\n");
     root = temp;
     return;
   }

   prev=NULL;
   cur=root;

   while(cur!=NULL)
   {
      prev=cur;
      cur=(value<cur->value)?cur->left:cur->right;
   }

   if(value < prev->value)
     prev->left=temp;
   else
     prev->right=temp;
}



void preorder(mynode *root)
{
  if(root)
  {
    printf("[%d] ", root->value);
    preorder(root->left);
    preorder(root->right);
  }
}
```

```
void postorder(mynode *root)
{
  if(root)
  {
    postorder(root->left);
    postorder(root->right);
    printf("[%d] ", root->value);
  }
}
```

```
void inorder(mynode *root)
{
  if(root)
  {
    inorder(root->left);
    printf("[%d] ", root->value);
    inorder(root->right);
  }
}
```

What is a threaded binary tree?

Since traversing the three is the most frequent operation, a method must be devised to improve the speed. This is where Threaded tree comes into picture. If the right link of a node in a tree is NULL, it can be replaced by the address of its inorder successor. An extra field called the rthread is used. If rthread is equal to 1, then it means that the right link of the node points to the inorder success. If its equal to 0, then the right link represents an ordinary link connecting the right subtree.

```
struct node
{
  int value;
  struct node *left;
  struct node *right;
  int rthread;
}
```

Function to find the inorder successor

```
mynode *inorder_successor(mynode *x)
{
  mynode *temp;

  temp = x->right;
  if(x->rthread==1)return(temp);

  while(temp->left!=NULL)temp = temp->left;

  return(temp);
}
```

Function to traverse the threaded tree in inorder

```
void inorder(mynode *head)
{
   mynode *temp;

   if(head->left==head)
   {
      printf("\nTree is empty!\n");
      return;
   }

   temp = head;

   for(;;)
   {
       temp = inorder_successor(temp);
       if(temp==head)return;
       printf("%d ", temp->value);
   }

}
```

Inserting toward the left of a node in a threaded binary tree.

```
void insert(int item, mynode *x)
{
   mynode *temp;
   temp = getnode();
   temp->value = item;
   x->left = temp;
   temp->left=NULL;
   temp->right=x;
   temp->rthread=1;
}
```

Function to insert towards the right of a node in a threaded binary tree.

```
void insert_right(int item, mynode *x)
{
   mynode *temp, r;

   temp=getnode();
   temp->info=item;
   r=x->right;
   x->right=temp;
   x->rthread=0;
   temp->left=NULL;
   temp->right=r;
   temp->rthread=1;
}
```

Function to find the inorder predecessor (for a left threaded binary three)

```
mynode *inorder_predecessor(mynode *x)
{
     mynode *temp;

     temp = x->left;

     if(x->lthread==1)return(temp);

     while(temp->right!=NULL)
       temp=temp->right;

     return(temp);
}
```

Bit Fiddling

Write a C program to count bits set in an integer?

This is one of the most frequently asked interview questions of all times...

There are a number of ways to count the number of bits set in an integer. Here are some C programs to do the same.

Method1

This is the most basic way of doing it.

```
#include<stdio.h>
int main()
{
  unsinged int num=10;
  int ctr=0;

  for(;num!=0;num>>=1)
  {
    if(num&1)
    {
      ctr++;
    }
  }
  printf("\n Number of bits set in [%d] = [%d]\n", num, ctr);
  return(0);

}
```

Method2

This is a faster way of doing the same thing. Here the control goes into the while loop only as many times as the number of bits set to 1 in the integer!.

```
#include<stdio.h>
int main()
{
  int num=10;
  int ctr=0;

  while(num)
  {
    ctr++;
    num = num & (num -
 1); // This clears the least significant bit set.
  }

  printf("\n Number of bits set in [%d] = [%d]\n", num, ctr);
  return(0);

}
```

Method3

This method is very popular because it uses a lookup table. This speeds up the computation. What it does is it keeps a table which hardcodes the number of bits set in each integer from 0 to 256.

For example

```
0 - 0 Bit(s) set.
1 - 1 Bit(s) set.
2 - 1 Bit(s) set.
3 - 2 Bit(s) set.
...
```

So here is the code...

```
const unsigned char LookupTable[]     = {  0, 1, 1, 2, 1, 2, 2, 3, 1, 2
, 2, 3, 2, 3, 3, 4,
                                           1, 2, 2, 3, 2, 3, 3, 4, 2, 3
, 3, 4, 3, 4, 4, 5,
                                           1, 2, 2, 3, 2, 3, 3, 4, 2, 3
, 3, 4, 3, 4, 4, 5,
                                           2, 3, 3, 4, 3, 4, 4, 5, 3, 4
, 4, 5, 4, 5, 5, 6,
                                           1, 2, 2, 3, 2, 3, 3, 4, 2, 3
, 3, 4, 3, 4, 4, 5,
                                           2, 3, 3, 4, 3, 4, 4, 5, 3, 4
, 4, 5, 4, 5, 5, 6,
                                           2, 3, 3, 4, 3, 4, 4, 5, 3, 4
, 4, 5, 4, 5, 5, 6,
                                           3, 4, 4, 5, 4, 5, 5, 6, 4, 5
, 5, 6, 5, 6, 6, 7,
                                           1, 2, 2, 3, 2, 3, 3, 4, 2, 3
, 3, 4, 3, 4, 4, 5,
                                           2, 3, 3, 4, 3, 4, 4, 5, 3, 4
, 4, 5, 4, 5, 5, 6,
                                           2, 3, 3, 4, 3, 4, 4, 5, 3, 4
, 4, 5, 4, 5, 5, 6,
                                           3, 4, 4, 5, 4, 5, 5, 6, 4, 5
, 5, 6, 5, 6, 6, 7,
                                           2, 3, 3, 4, 3, 4, 4, 5, 3, 4
, 4, 5, 4, 5, 5, 6,
                                           3, 4, 4, 5, 4, 5, 5, 6, 4, 5
, 5, 6, 5, 6, 6, 7,
                                           3, 4, 4, 5, 4, 5, 5, 6, 4, 5
, 5, 6, 5, 6, 6, 7,
                                           4, 5, 5, 6, 5, 6, 6, 7, 5, 6
, 6, 7, 6, 7, 7, 8};

unsigned int num;
unsigned int ctr;   // Number of bits set.

ctr = LookupTable[num & 0xff] +
      LookupTable[(num >> 8) & 0xff] +
      LookupTable[(num >> 16) & 0xff] +
      LoopupTable[num >> 24];
```

What purpose do the bitwise and, or, xor and the shift operators serve?

The AND operator

```
Truth Table
-----------
0 AND 0 = 0
0 AND 1 = 0
1 AND 0 = 0
1 AND 1 = 1
```

```
x AND 0 = 0
x AND 1 = x
```

We use bitwise "and" to test if certain bit(s) are one or not. And'ing a value against a pattern with ones only in the bit positions you are interested in will give zero if none of them are on, nonzero if one or more is on. We can also use bitwise "and" to turn off (set to zero) any desired bit(s). If you "and" a pattern against a variable, bit positions in the pattern that are ones will leave the target bit unchanged, and bit positions in the pattern that are zeros will set the target bit to zero.

The OR operator

```
Truth Table
-----------
0 OR 0 = 0
0 OR 1 = 1
1 OR 0 = 1
1 OR 1 = 1

x OR 0 = x
x OR 1 = 1
```

Use bitwise "or" to turn on (set to one) desired bit(s).

The XOR operator

```
0 XOR 0 = 0
0 XOR 1 = 1
1 XOR 0 = 1
1 XOR 1 = 0

x XOR 0 = x
x XOR 1 = ~x
```

Use bitwise "exclusive or" to flip or reverse the setting of desired bit(s) (make it a one if it was zero or make it zero if it was one).

The >>, <<, >>=, <<= operators

Operators >> and << can be used to shift the bits of an operand to the right or left a desired number of positions. The number of positions to be shifted can be specified as a constant, in a variable or as an expression. Bits shifted out are lost. For left shifts, bit positions vacated by shifting always filled with zeros. For right shifts, bit positions vacated by shifting filled with zeros for unsigned data type and with copy of the highest (sign) bit for signed data type. The right shift operator can be used to achieve quick multiplication by a power of 2. Similarly the right shift operator can be used to do a quick division by power of 2 (unsigned types only). The operators >> and <<, dont change the operand at all. However, the operators >>= and <=< also change the operand after doing the shift operations.

```
x << y  -
 Gives value x shifted left y bits (Bits positions vacated by shift are
 filled with zeros).
x <<= y -
 Shifts variable x left y bits (Bits positions vacated by shift are fil
led with zeros).

A left shift of n bits multiplies by 2 raise to n.

x >> y  - Gives value x shifted right y bits.
x >>= y - Shifts variable x right y bits.

For the right shift, All bits of operand participate in the shift. For
unsigned data type,
bits positions vacated by shift are filled with zeros. For signed data
type, bits positions
vacated by shift are filled with the original highest bit (sign bit). R
ight shifting n bits
divides by 2 raise to n. Shifting signed values may fail because for ne
gative values the result never
gets past -1:

-5 >> 3 is -1 and not 0 like -5/8.
```

Good interview question

A simple C command line utility takes a series of command line options. The options are given to the utility like this : <utility_name> options=[no]option1,[no]options2,[no]option3?... Write C code using bitwise operators to use these flags in the code.

```
//Each option will have a bit reserved in the global_options_bits integ
er. The global_options_bits
// integer will have a bit set or not set depending on how the option w
```

```
as specified by the user.
// For example, if the user said nooption1, the bit for OPTION1 in glob
al_options_bits
// will be 0. Likewise, if the user specified option3, the bit for OPTI
ON3 in global_options_bits
// will be set to 1.

#define OPTION1 "option1" // For strcmp() with the passed arguments.
#define OPTION1_BITPOS (0x00000001) // Bit reserved for this option.
#define OPTION2 "option2"
#define OPTION2_BITPOS (0x00000002)
#define OPTION3 "option3"
#define OPTION3_BITPOS (0x00000004)

//Required to do the bit operations.
#define ALL_BITPOS (0x0001ffff)

// Assume you have already parsed the command line option and that
// parsed_argument_without_no has option1 or option2 or option3 (depend
ing on what has
// been provided at the command line) and the variable negate_argument
says if the
// option was negated or not (i.e, if it was option1 or nooption1)

if (strcmp((char *) parsed_argument_without_no, (char *) OPTION1) == 0)
{
    // Copy the global_options_bits to a temporary variable.
    tmp_action= global_options_bits;

    if (negate_argument)
    {
        // Setting the bit for this particular option to 0 as the optio
n has
        // been negated.
        action_mask= ~(OPTION1_BITPOS) & ALL_BITPOS;
        tmp_action= tmp_action & action_mask;
    }
    else
    {
        //Setting the bit for this particular option to 1.
        action_mask= (OPTION1_BITPOS);
        tmp_action= tmp_action | action_mask;
    }

    // Copy back the final bits to the global_options_bits variable
    global_options_bits= tmp_action;
}
else if (strcmp((char *) parsed_argument_without_no, (char *) OPTION2)
== 0)
{
    //Similar code for OPTION2
}
else if (strcmp((char *) parsed_argument_without_no, (char *) OPTION3)
== 0)
{
    //Similar code for OPTION3
}
```

```
//Now someone who wishes to check if a particular option was set or not
 can use the
// following type of code anywhere else in the code.
if(((global_options_bits & OPTION1_BITPOS) == OPTION1_BITPOS)
{
    //Do processing for the case where OPTION1 was active.
}
else
{
    //Do processing for the case where OPTION1 was NOT active.
}
```

How to reverse the bits in an interger?

Here are some ways to reverse the bits in an integer.

Method1

```
unsigned int num;            // Reverse the bits in this number.
unsigned int temp = num;     // temp will have the reversed bits of num
.

int i;

for (i = (sizeof(num)*8-1); i; i--)
{
 temp = temp | (num & 1);
 temp <<= 1;
 num  >>= 1;
}

temp = temp | (num & 1);
```

Method2

In this method, we use a lookup table.

```
const unsigned char ReverseLookupTable[] =
{
     0x00, 0x80, 0x40, 0xC0, 0x20, 0xA0, 0x60, 0xE0, 0x10, 0x90, 0x50,
0xD0, 0x30, 0xB0, 0x70, 0xF0,
     0x08, 0x88, 0x48, 0xC8, 0x28, 0xA8, 0x68, 0xE8, 0x18, 0x98, 0x58,
0xD8, 0x38, 0xB8, 0x78, 0xF8,
     0x04, 0x84, 0x44, 0xC4, 0x24, 0xA4, 0x64, 0xE4, 0x14, 0x94, 0x54,
0xD4, 0x34, 0xB4, 0x74, 0xF4,
     0x0C, 0x8C, 0x4C, 0xCC, 0x2C, 0xAC, 0x6C, 0xEC, 0x1C, 0x9C, 0x5C,
0xDC, 0x3C, 0xBC, 0x7C, 0xFC,
     0x02, 0x82, 0x42, 0xC2, 0x22, 0xA2, 0x62, 0xE2, 0x12, 0x92, 0x52,
0xD2, 0x32, 0xB2, 0x72, 0xF2,
     0x0A, 0x8A, 0x4A, 0xCA, 0x2A, 0xAA, 0x6A, 0xEA, 0x1A, 0x9A, 0x5A,
0xDA, 0x3A, 0xBA, 0x7A, 0xFA,
     0x06, 0x86, 0x46, 0xC6, 0x26, 0xA6, 0x66, 0xE6, 0x16, 0x96, 0x56,
0xD6, 0x36, 0xB6, 0x76, 0xF6,
     0x0E, 0x8E, 0x4E, 0xCE, 0x2E, 0xAE, 0x6E, 0xEE, 0x1E, 0x9E, 0x5E,
0xDE, 0x3E, 0xBE, 0x7E, 0xFE,
     0x01, 0x81, 0x41, 0xC1, 0x21, 0xA1, 0x61, 0xE1, 0x11, 0x91, 0x51,
0xD1, 0x31, 0xB1, 0x71, 0xF1,
     0x09, 0x89, 0x49, 0xC9, 0x29, 0xA9, 0x69, 0xE9, 0x19, 0x99, 0x59,
0xD9, 0x39, 0xB9, 0x79, 0xF9,
     0x05, 0x85, 0x45, 0xC5, 0x25, 0xA5, 0x65, 0xE5, 0x15, 0x95, 0x55,
0xD5, 0x35, 0xB5, 0x75, 0xF5,
     0x0D, 0x8D, 0x4D, 0xCD, 0x2D, 0xAD, 0x6D, 0xED, 0x1D, 0x9D, 0x5D,
0xDD, 0x3D, 0xBD, 0x7D, 0xFD,
     0x03, 0x83, 0x43, 0xC3, 0x23, 0xA3, 0x63, 0xE3, 0x13, 0x93, 0x53,
0xD3, 0x33, 0xB3, 0x73, 0xF3,
     0x0B, 0x8B, 0x4B, 0xCB, 0x2B, 0xAB, 0x6B, 0xEB, 0x1B, 0x9B, 0x5B,
0xDB, 0x3B, 0xBB, 0x7B, 0xFB,
     0x07, 0x87, 0x47, 0xC7, 0x27, 0xA7, 0x67, 0xE7, 0x17, 0x97, 0x57,
0xD7, 0x37, 0xB7, 0x77, 0xF7,
     0x0F, 0x8F, 0x4F, 0xCF, 0x2F, 0xAF, 0x6F, 0xEF, 0x1F, 0x9F, 0x5F,
0xDF, 0x3F, 0xBF, 0x7F, 0xFF
};

unsigned int num; // Reverse the bits in this number.
unsigned int temp; // Store the reversed result in this.

temp = (ReverseLookupTable[num & 0xff] << 24) |
       (ReverseLookupTable[(num >> 8) & 0xff] << 16) |
       (ReverseLookupTable[(num >> 16) & 0xff] << 8) |
       (ReverseLookupTable[(num >> 24) & 0xff]);
```

Check if the 20th bit of a 32 bit integer is on or off?

AND it with x00001000 and check if its equal to x00001000

```
if((num & x00001000)==x00001000)
```

Note that the digits represented here are in hex.

```
     0      0      0      0      1      0      0      0
                                 ^
                                 |

 x0000   0000   0000   0000   0001   0000   0000   0000 = 32 bits

  ^                              ^                    ^
  |                              |                    |

  0th bit                        20th bit             32nd bit
```

How to reverse the odd bits of an integer?

XOR each of its 4 hex parts with 0101.

How would you count the number of bits set in a floating point number?

Sorting Techniques

What is heap sort?

A Heap is an almost complete binary tree.In this tree, if the maximum level is i, then, upto the (i-1)th level should be complete. At level i, the number of nodes can be less than or equal to 2^i. If the number of nodes is less than 2^i, then the nodes in that level should be completely filled, only from left to right.

The property of an ascending heap is that, the root is the lowest and given any other node i, that node should be less than its left child and its right child. In a descending heap, the root should be the highest and given any other node i, that node should be greater than its left child and right child.

To sort the elements, one should create the heap first. Once the heap is created, the root has the highest value. Now we need to sort the elements in ascending order. The root can not be exchanged with the nth element so that the item in the nth position is sorted. Now, sort the remaining (n-1) elements. This can be achieved by reconstructing the heap for (n-1) elements.

A highly simplified pseudocode is given below

```
heapsort()
{
   n = array();   // Convert the tree into an array.
   makeheap(n);   // Construct the initial heap.

   for(i=n; i>=2; i--)
   {
      swap(s[1],s[i]);
      heapsize--;
      keepheap(i);
   }
}

makeheap(n)
{
   heapsize=n;
   for(i=n/2; i>=1; i--)
     keepheap(i);
}

keepheap(i)
{
   l = 2*i;
   r = 2*i + 1;

   p = s[l];
   q = s[r];
   t = s[i];

   if(l<=heapsize && p->value > t->value)
     largest = l;
   else
     largest = i;

   m = s[largest];

   if(r<=heapsize && q->value > m->value)
     largest = r;

   if(largest != i)
   {
      swap(s[i], s[largest]);
      keepheap(largest);
   }
}
```

What is the difference between Merge Sort and Quick sort?

Both Merge-sort and Quick-sort have same time complexity i.e. O(nlogn). In merge sort the file a[1:n] was divided at its midpoint into sub-arrays which are independently sorted and later merged. Whereas, in quick sort the division into two sub-arrays is made so that the sorted sub-arrays do not need to be merged latter.

Give pseudocode for the mergesort algorithm.

```
Mergesort(a, left, right)
{
   if(left<right)
   {
     I=(left+right)/2;
     Mergesort(a,left, I);
     Mergesort(a,I+1,right);
     Merge(a,b,left,I,right);
     Copy(b,a,left,right);
   }
}
```

The merge would be something liks this

```
merge(int a[], int b[], int c[], int m, int n)
{
  int i, j, k;
  i = 0;
  j = 0;
  k = 0;

  while(i<=m && j<=n)
  {
    if(a[i] < a[j])
    {
        c[k]=a[i];
        i=i+1;
    }
    else
    {
        c[k]=a[j];
        j=j+1;
    }
    k=k+1;
  }

  while(i<=m)
    c[k++]=a[i++];

  while(j<=n)
    c[k++]=a[j++];
}
```

Implement the bubble sort algorithm. How can it be improved? Write the code for selection sort, quick sort, insertion sort.

Here is the Bubble sort algorithm

```
void bubble_sort(int a[], int n)
{
  int i, j, temp;

  for(j = 1; j < n; j++)
  {
     for(i = 0; i < (n - j); i++)
     {
        if(a[i] >= a[i + 1])
        {
           //Swap a[i], a[i+1]
        }
     }
  }
}
```

To improvise this basic algorithm, keep track of whether a particular pass results in any swap or not. If not, you can break out without wasting more cycles.

```
void bubble_sort(int a[], int n)
{
  int i, j, temp;
  int flag;

  for(j = 1; j < n; j++)
  {
     flag = 0;
     for(i = 0; i < (n - j); i++)
     {
        if(a[i] >= a[i + 1])
        {
           //Swap a[i], a[i+1]
           flag = 1;
        }
     }

     if(flag==0)break;
  }
}
```

This is the selection sort algorithm

```
void selection_sort(int a[], int n)
{
   int i, j, small, pos, temp;

   for(i = 0; i < (n - 1); i++)
   {
      small = a[i];
      pos   = i;

      for(j = i + 1; j < n; j++)
      {
         if(a[j] < small)
         {
            small = a[j];
            pos   = j;
         }
      }

      temp   = a[pos];
      a[pos] = a[i];
      a[i]   = temp;
   }
}
```

Here is the Quick sort algorithm

```
int partition(int a[], int low, int high)
{
   int i, j, temp, key;

   key = a[low];
   i   = low + 1;
   j   = high;

   while(1)
   {
       while(i < high && key >= a[i])i++;

       while(key < a[j])j--;

       if(i < j)
       {
           temp = a[i];
           a[i] = a[j];
           a[j] = temp;
       }
       else
       {
           temp   = a[low];
           a[low] = a[j];
           a[j]   = temp;
           return(j);
       }
   }
}


void quicksort(int a[], int low, int high)
{
   int j;

   if(low < high)
   {
       j = partition(a, low, high);
       quicksort(a, low, j - 1);
       quicksort(a, j + 1, high);
   }
}


int main()
{
  // Populate the array a
  quicksort(a, 0, n - 1);
}
```

Here is the Insertion sort algorithm

```
void insertion_sort(int a[], int n)
{
   int i, j, item;

   for(i = 0; i < n; i++)
   {
       item = a[i];
       j    = i - 1;

       while(j >=0 && item < a[j])
       {
          a[j + 1] = a[j];
          j--;
       }

       a[j + 1] = item;
   }
}
```

How can I sort things that are too large to bring into memory?

A sorting program that sorts items that are on secondary storage (disk or tape) rather than primary storage (memory) is called an external sort. Exactly how to sort large data depends on what is meant by ?too large to fit in memory.? If the items to be sorted are themselves too large to fit in memory (such as images), but there aren?t many items, you can keep in memory only the sort key and a value indicating the data?s location on disk. After the key/value pairs are sorted, the data is rearranged on disk into the correct order. If ?too large to fit in memory? means that there are too many items to fit into memory at one time, the data can be sorted in groups that will fit into memory, and then the resulting files can be merged. A sort such as a radix sort can also be used as an external sort, by making each bucket in the sort a file. Even the quick sort can be an external sort. The data can be partitioned by writing it to two smaller files. When the partitions are small enough to fit, they are sorted in memory and concatenated to form the sorted file.