CPM:

```
#include<stdio.h>
#include<conio.h>
int main()
{
 int i,j,k,m,p,count,n,f[10];
 float a[10][10],e[10],l[10],w,t;
 printf("Enter the no. of nodes:");
 scanf("%d",&n);
 printf("\nEnter network diagram and -1 to exit:");
 for(i=0;i<n;i++)
  for(j=0;j<n;j++)
       a[i][j]=0;
 while(1)
 {
  scanf("%d%d%f",&j,&k,&w);

  if(j==-1)
  break;
  a[j][k]=w;
 }
 e[0]=0;
 for(k=1;k<n;k++)
 {
  count=0;
  for(i=0;i<n;i++)
  {
       if(a[i][k]!=0)
       {
        count++;
        m=i;
       }
  }
  if(count==1)
       e[k]=e[m]+a[m][k];
  else
  {
  e[k]=0;
  for(i=0;i<n;i++)
       {
        if(a[i][k]!=0)
        {
         t=e[i]+a[i][k];
         if(t>e[k])
              e[k]=t;
        }
       }
  }
 }
 for(i=0;i<n;i++)
  printf("  %f",e[i]);
 l[n-1]=e[n-1];
 for(k=n-2;k>=0;k--)
 {
  count=0;
  for(j=0;j<n;j++)
  {
       if(a[k][j]!=0)
       {
        count++;
```

```
          p=j;
        }
   }
   if(count==1)
        l[k]=l[p]-a[k][p];
   else
   {
        l[k]=1000;
        for(j=0;j<n;j++)
        {
         if(a[k][j]!=0)
         {
          t=l[j]-a[k][j];
          if(t<l[k])
               l[k]=t;
         }
        }
   }
 }
 for(i=0;i<n;i++)
  printf("  %f",l[i]);
 for(i=1,k=0;i<n;i++)
 {
  if(e[i]==l[i])
        f[k++]=i;
 }
 printf("\nThe critical path is: 0");
 for(i=0;i<k;i++)
 printf("->%d",f[i]);
 getch();
 return 0;
}
```

CPM ON JAVA

```
public class CriticalPath {

  public static void main(String[] args) {
    //The example dependency graph from
    //http://www.ctl.ua.edu/math103/scheduling/scheduling_algorithms.htm
    HashSet<Task> allTasks = new HashSet<Task>();
    Task end = new Task("End", 0);
    Task F = new Task("F", 2, end);
    Task A = new Task("A", 3, end);
    Task X = new Task("X", 4, F, A);
    Task Q = new Task("Q", 2, A, X);
    Task start = new Task("Start", 0, Q);
    allTasks.add(end);
    allTasks.add(F);
    allTasks.add(A);
    allTasks.add(X);
    allTasks.add(Q);
    allTasks.add(start);
    System.out.println("Critical Path:
"+Arrays.toString(criticalPath(allTasks)));
  }

  //A wrapper class to hold the tasks during the calculation
  public static class Task{
    //the actual cost of the task
    public int cost;
    //the cost of the task along the critical path
    public int criticalCost;
    //a name for the task for printing
    public String name;
    //the tasks on which this task is dependant
    public HashSet<Task> dependencies = new HashSet<Task>();
    public Task(String name, int cost, Task... dependencies) {
      this.name = name;
      this.cost = cost;
      for(Task t : dependencies){
        this.dependencies.add(t);
      }
    }
    @Override
    public String toString() {
      return name+": "+criticalCost;
    }
    public boolean isDependent(Task t){
      //is t a direct dependency?
      if(dependencies.contains(t)){
        return true;
      }
      //is t an indirect dependency
      for(Task dep : dependencies){
        if(dep.isDependent(t)){
          return true;
        }
      }
      return false;
    }
  }
```

```
  public static Task[] criticalPath(Set<Task> tasks){
    //tasks whose critical cost has been calculated
    HashSet<Task> completed = new HashSet<Task>();
    //tasks whose ciritcal cost needs to be calculated
    HashSet<Task> remaining = new HashSet<Task>(tasks);

    //Backflow algorithm
    //while there are tasks whose critical cost isn't calculated.
    while(!remaining.isEmpty()){
      boolean progress = false;

      //find a new task to calculate
      for(Iterator<Task> it = remaining.iterator();it.hasNext();){
        Task task = it.next();
        if(completed.containsAll(task.dependencies)){
          //all dependencies calculated, critical cost is max dependency
          //critical cost, plus our cost
          int critical = 0;
          for(Task t : task.dependencies){
            if(t.criticalCost > critical){
              critical = t.criticalCost;
            }
          }
          task.criticalCost = critical+task.cost;
          //set task as calculated an remove
          completed.add(task);
          it.remove();
          //note we are making progress
          progress = true;
        }
      }
      //If we haven't made any progress then a cycle must exist in
      //the graph and we wont be able to calculate the critical path
      if(!progress) throw new RuntimeException("Cyclic dependency, algorithm
stopped!");
    }

    //get the tasks
    Task[] ret = completed.toArray(new Task[0]);
    //create a priority list
    Arrays.sort(ret, new Comparator<Task>() {

      @Override
      public int compare(Task o1, Task o2) {
        //sort by cost
        int i= o2.criticalCost-o1.criticalCost;
        if(i != 0)return i;

        //using dependency as a tie breaker
        //note if a is dependent on b then
        //critical cost a must be >= critical cost of b
        if(o1.isDependent(o2))return -1;
        if(o2.isDependent(o1))return 1;
        return 0;
      }
    });

    return ret;
  }
}
```