**Assignment 2**

**Greedy Algorithms**

**Rehnoor Aulakh**

**102317137**

**2Q16**

**Q1 Activity Selection Problem**

#include<iostream>

#include<vector>

#include<algorithm>

using namespace std;

class Activity

{

public:

int start;

int finish;

Activity(){

start=0;

finish=0;

}

Activity(int start, int finish)

{

this->finish=finish;

this->start=start;

}

void display()

{

cout<<"Start: "<<start<<endl;

cout<<"Finish: "<<finish<<endl;

}

};

bool activityCompare(const Activity &a1, const Activity &a2) {

return a1.finish < a2.finish;

}

vector<Activity> run(vector<Activity> &act)

{

sort(act.begin(),act.end(),activityCompare);

int finish=act[0].finish;

vector<Activity> sol;

sol.push\_back(act[0]);

for(int i=1;i<act.size();i++)

{

if(act[i].start>=finish)

{

sol.push\_back(act[i]);

finish=act[i].finish;

}

}

return sol;

}

int main()

{

Activity act[]={Activity(1,2),Activity(3,4),Activity(0,6),Activity(5,7),Activity(8,9),Activity(5,9)};

//First one will run as it is

vector<Activity> act2;

for(int i=0;i<sizeof(act)/sizeof(Activity);i++)

{

act2.push\_back(act[i]);

}

vector<Activity> sol=run(act2);

for(int i=0;i<sol.size();i++)

{

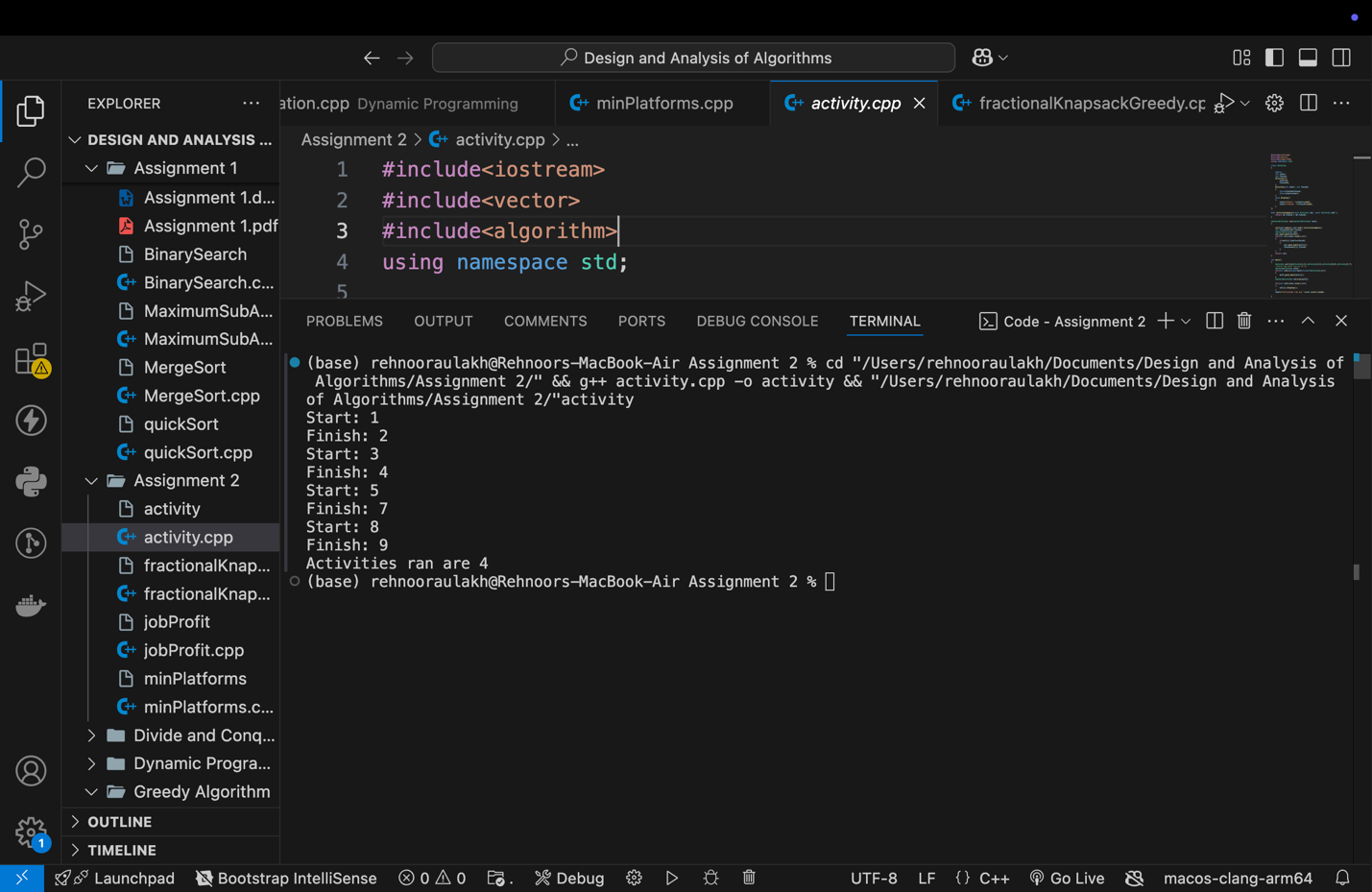
sol[i].display();

}

cout<<"Activities ran are "<<sol.size()<<endl;

}

**Solution**

****

**Q2 Minimum Platforms Problem**

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

int findPlatform(vector<int> arr, vector<int> dep) {

int n = arr.size();

sort(arr.begin(), arr.end());

sort(dep.begin(), dep.end());

int platforms\_needed = 1, result = 1;

int i = 1, j = 0;

while (i < n && j < n) {

if (arr[i] <= dep[j]) {

platforms\_needed++;

i++;

} else {

platforms\_needed--;

j++;

}

if (platforms\_needed > result) {

result = platforms\_needed;

}

}

return result;

}

int main() {

vector<int> arr = {900, 940, 950, 1100, 1500, 1800};

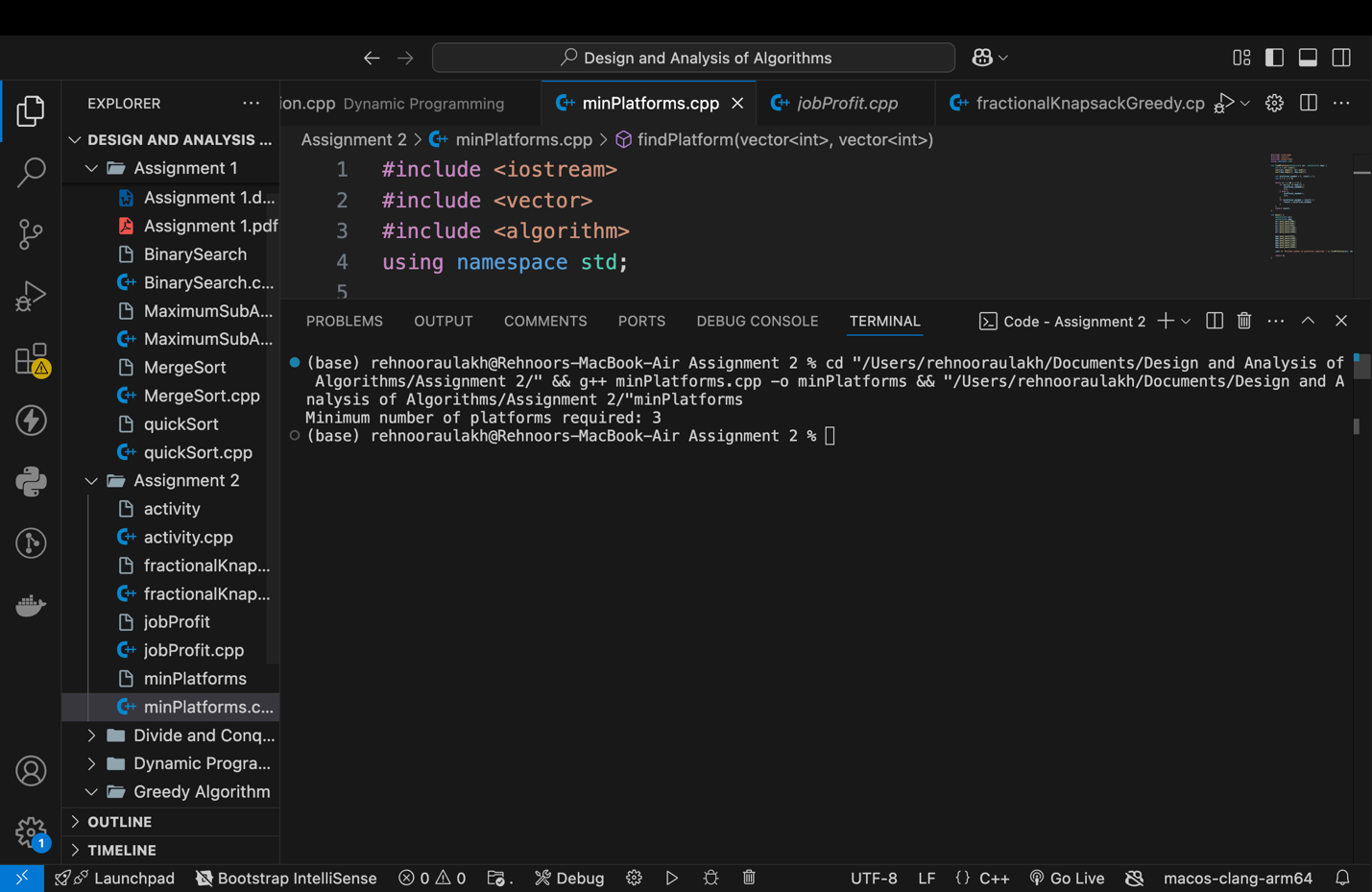
vector<int> dep = {910, 1200, 1120, 1130, 1900, 2000};

cout << "Minimum number of platforms required: " << findPlatform(arr, dep) << endl;

return 0;

}

**Solution**



**Q3 Job Profit Problem**

#include<iostream>

#include<vector>

#include<algorithm>

using namespace std;

class Job

{

private:

char jobId;

public:

int deadline;

int profit;

Job()

{

}

Job(char jobId, int deadline, int profit)

{

this->jobId=jobId;

this->deadline=deadline;

this->profit=profit;

}

void display()

{

cout<<"JobID: "<<jobId<<endl;

cout<<"Deadline: "<<deadline<<endl;

cout<<"Profit: "<<profit<<endl;

}

};

bool compare(const Job &j1, const Job &j2)

{

return j1.profit>j2.profit;

}

vector<Job> run(vector<Job> job)

{

sort(job.begin(),job.end(),compare);

vector<Job> sol;

sol.push\_back(job[0]);

//find the size of gantt chart

int size=0;

for(int i=0;i<job.size();i++)

{

if(job[i].deadline>size)

{

size=job[i].deadline;

}

}

//make an int array of size to store the values of profit

int gantt[size];

bool isFilled[size];

for(int i=0;i<size;i++)

{

isFilled[i]=false;

}

//maximum profit add to gantt chart

//find the position of job[0] in the gantt chart i.e the deadline

//deadline-1 will be the position

gantt[job[0].deadline-1]=job[0].profit;

isFilled[job[0].deadline-1]=true;

for(int i=1;i<job.size();i++)

{

// add job[i] to the gantt chart if space is available

//first try in the desired position if available

if(!isFilled[job[i].deadline-1])

{

sol.push\_back(job[i]);

isFilled[job[i].deadline-1];

}

else

{

for(int j=job[i].deadline-1;j>=0;j--)

{

if(!isFilled[j])

{

sol.push\_back(job[i]);

isFilled[j]=true;

break;

}

}

}

}

return sol;

}

int main()

{

Job job[]={Job('a',4,20),Job('b',1,10),Job('c',1,40),Job('d',1,30)};

vector<Job> j;

for(int i=0;i<sizeof(job)/sizeof(Job);i++)

{

j.push\_back(job[i]);

}

vector<Job> sol=run(j);

int profit=0;

for(int i=0;i<sol.size();i++)

{

sol[i].display();

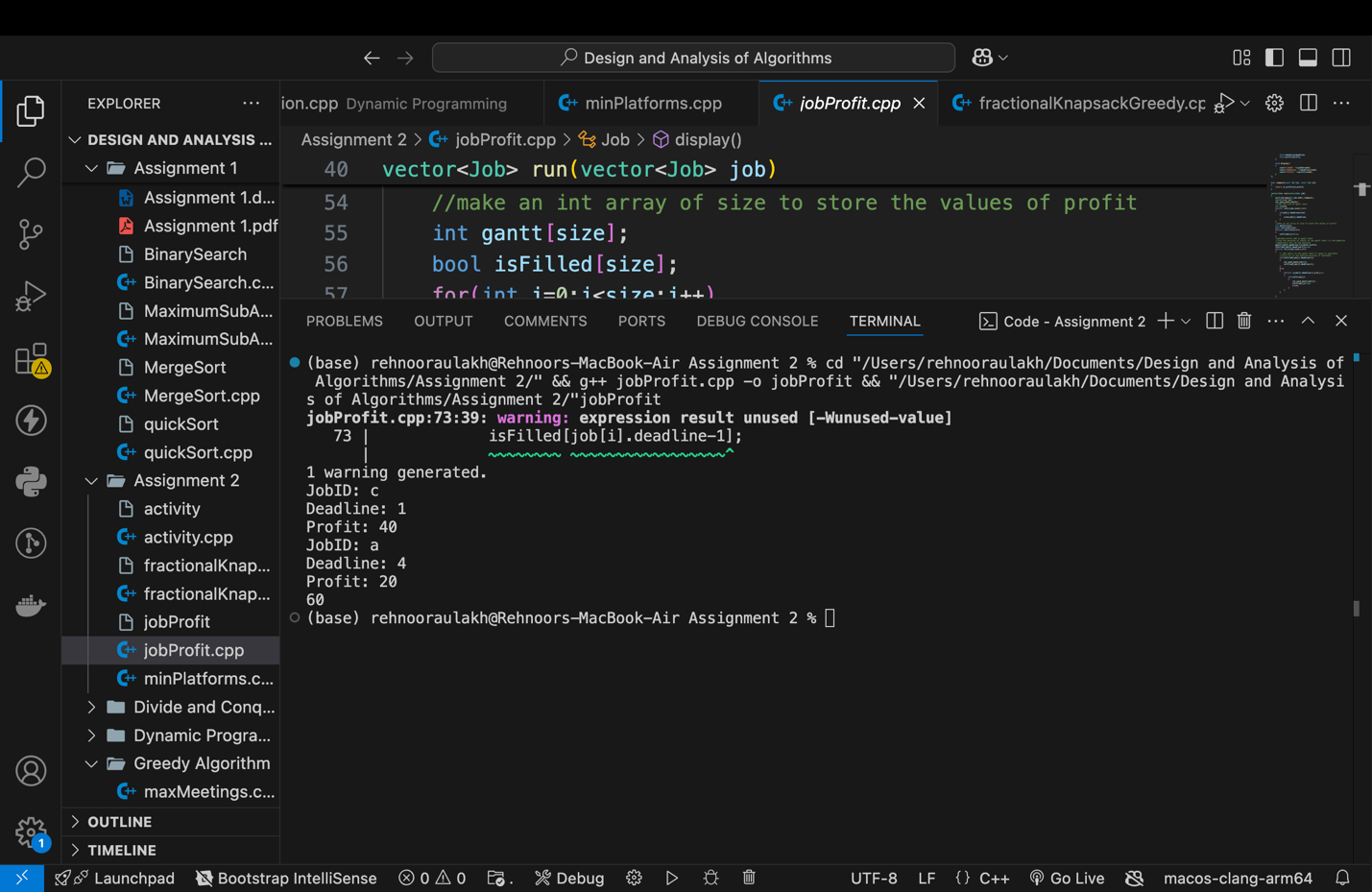
profit+=sol[i].profit;

}

cout<<profit<<endl;

}

**Solution**

****

**Q4 Fractional Knapsack**

#include<iostream>

#include<vector>

#include<algorithm>

using namespace std;

bool compare(vector<int>& a, vector<int> &b)

{

double r1=(double)a[0]/a[1];

double r2=(double)b[0]/b[1];

return r1>r2;

}

int maxProfit(vector<vector<int> >arr, int W)

{

//Sort these items according to profit-to-weight ratios

sort(arr.begin(),arr.end(),compare);

//Getting the number of items

int n=arr.size();

int currWeight=0;

double finalProfit=0;

for(int i=0;i<n;i++)

{

//If the item's weight can be fully accomplished

if(currWeight+arr[i][1]<=W)

{

currWeight+=arr[i][1];

finalProfit+=arr[i][0];

}

else

{

//Get the fraction of the next item that can be accomplished

int remaining=W-currWeight;

finalProfit+=arr[i][0]\*((double)remaining/arr[i][1]);

break;

}

}

return finalProfit;

}

int main()

{

// {Profit,Weight}

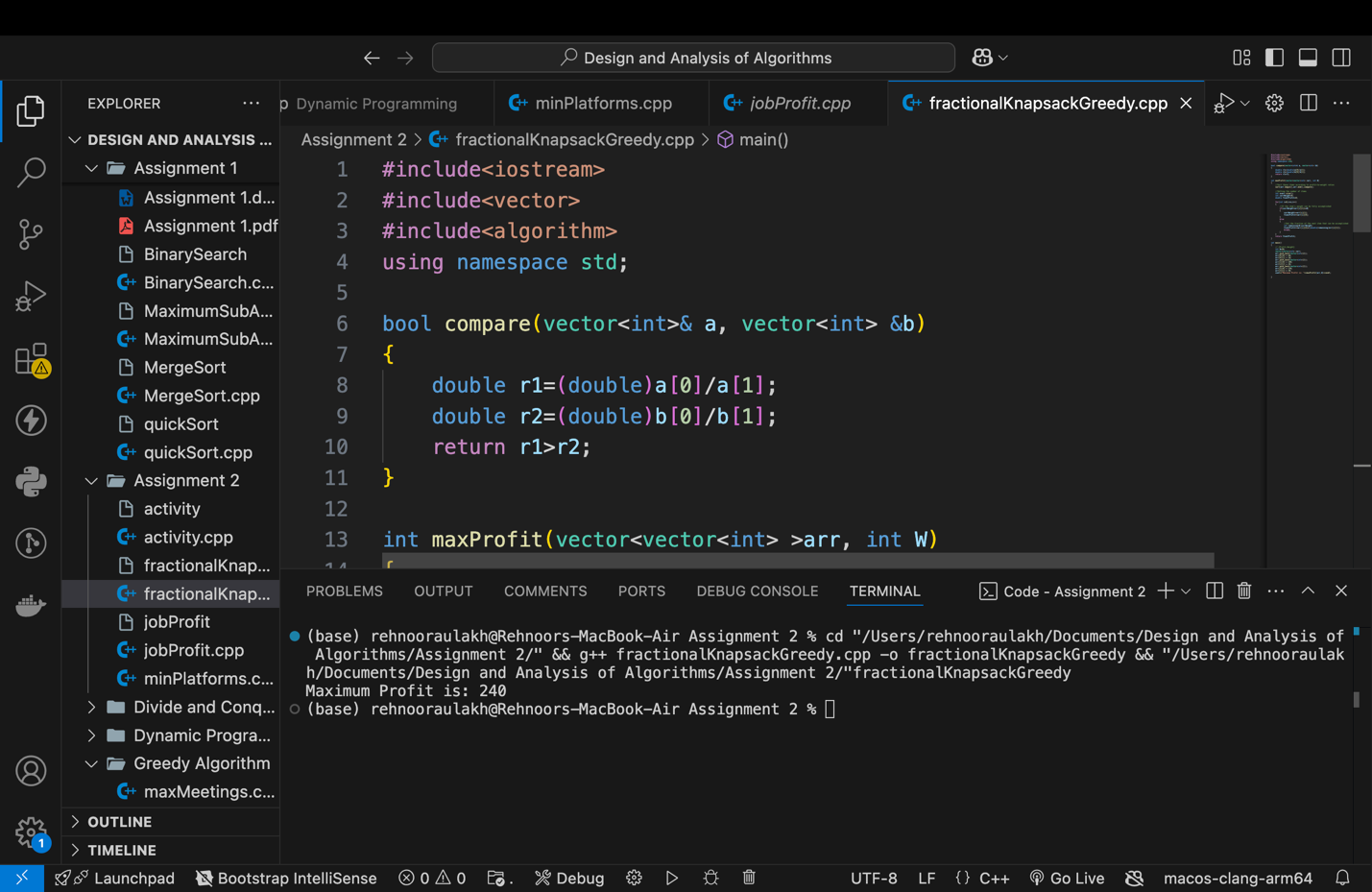
int W=50;

vector<vector<int> >arr={{60, 10}, {100, 20}, {120, 30}};

cout<<"Maximum Profit is: "<<maxProfit(arr,W);

}

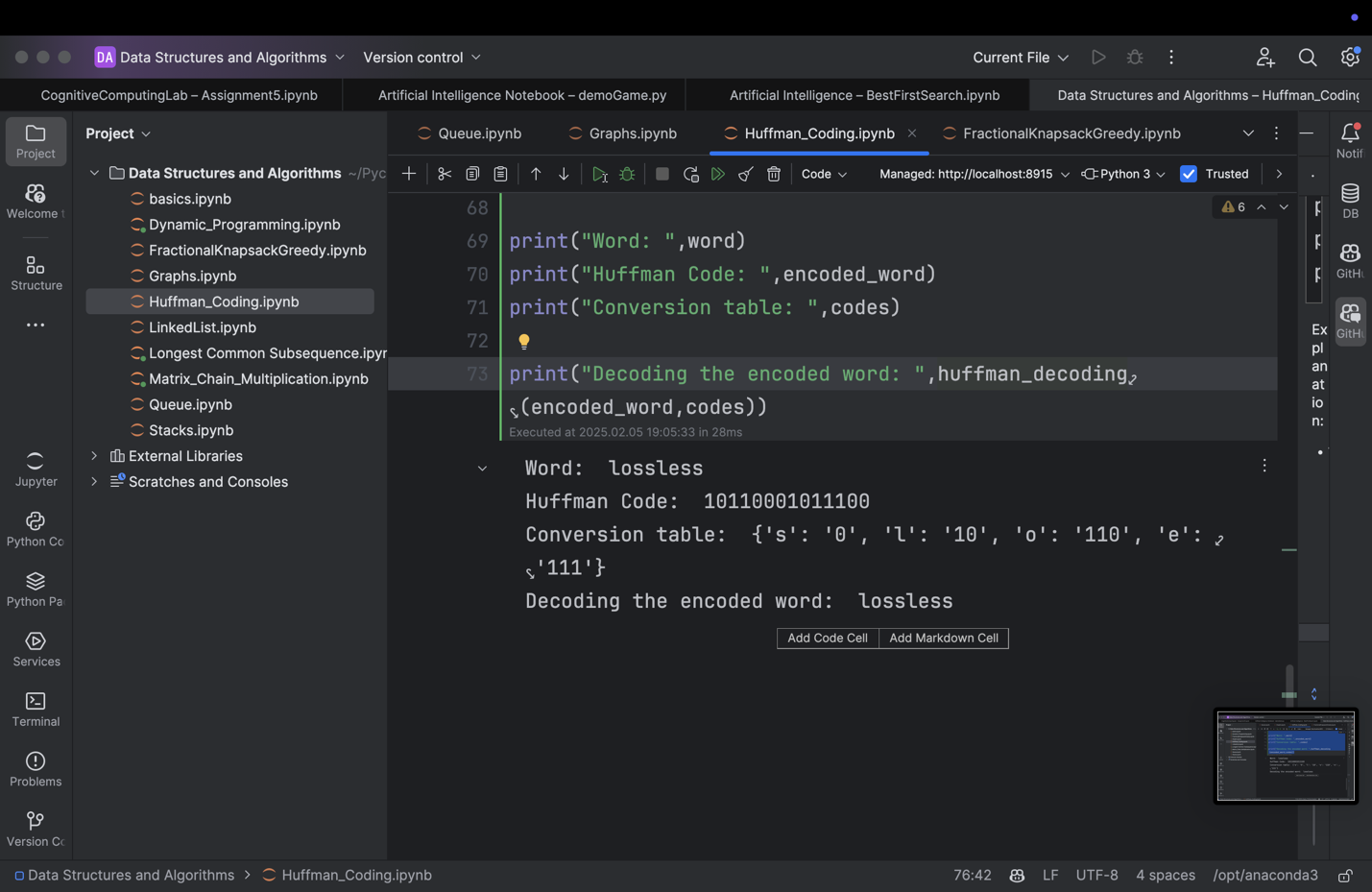
**Solution**

****

**Q5 Huffman Coding**

class Node:  
 def \_\_init\_\_(self,char=None,freq=0):  
 self.char=char  
 self.freq=freq  
 self.left=None  
 self.right=None  
   
nodes=[]  
  
def calculate\_frequencies(word):  
 frequencies={}  
 for char in word:  
 if char not in frequencies:  
 freq=word.count(char)  
 frequencies[char]=freq  
 nodes.append(Node(char,freq))  
   
def build\_huffman\_tree():  
 while len(nodes)>1:  
 #sort the nodes according to frequencies  
 nodes.sort(key= lambda x:x.freq)  
 #take out the first two least frequent entries   
 left=nodes.pop(0)  
 right=nodes.pop(0)  
 merged=Node(freq=left.freq+right.freq)  
 merged.left=left  
 merged.right=right  
   
 nodes.append(merged)  
 return nodes[0]  
  
def generate\_huffman\_codes(node,current\_code,codes):  
 if node is None:  
 return  
 if node.char is not None:  
 codes[node.char]=current\_code  
 generate\_huffman\_codes(node.left,current\_code+'0',codes)  
 generate\_huffman\_codes(node.right,current\_code+'1',codes)  
  
  
def huffman\_encoding(word):  
 global nodes  
 nodes=[]  
 calculate\_frequencies(word)  
 root=build\_huffman\_tree()  
 codes={}  
 generate\_huffman\_codes(root,'',codes)  
 return codes  
  
def huffman\_decoding(encoded\_word,codes):  
 current\_code=''  
 decoded\_chars=[]  
 #invert the codes dictionary to get the reverse mapping  
   
 code\_to\_char={v:k for k,v in codes.items()}  
   
 for bit in encoded\_word:  
 current\_code+=bit  
 if current\_code in code\_to\_char:  
 decoded\_chars.append(code\_to\_char[current\_code])  
 current\_code=''  
 return ''.join(decoded\_chars)  
   
  
word='lossless'  
codes=huffman\_encoding(word)  
encoded\_word=''.join(codes[char] for char in word)  
   
print("Word: ",word)  
print("Huffman Code: ",encoded\_word)  
print("Conversion table: ",codes)  
  
print("Decoding the encoded word: ",huffman\_decoding(encoded\_word,codes))

**Solution**

****