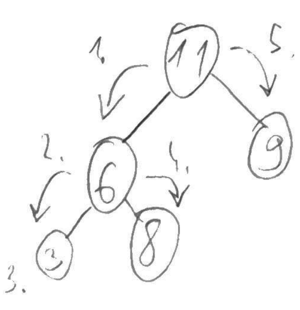
DZ 8

1.

Pseudokod:

```
findPaths(root, X):
  paths = []
  dfs(root, X, [], paths)
  return paths
dfs(node, X, path, paths):
  if node == NIL
    return
  path.append(node.value)
  if node.left == NIL and node.right == NIL
    if sum(path) == X:
      paths.append(path)
  else:
    dfs(node.left, X, path, paths)
    dfs(node.right, X, path, paths)
  path.pop()
```

Primjer: $X = 20$



1.: path = [11]

2.: path = [11,6]

3.: path = [11,6,3], paths = [[11,6,3]]

4.: path = [11,6,8]

5.: path = [11,9], paths = [[11,6,3], [11,9]]

Na kraju smo dobili sve puteve u grafu sa sumom 20.

2. Pseudokod:

```
findPath(Labyrinth):
queue = [(0,0)]
visited = set()
parent = {(0,0): NIL}
while queue:
  (x,y) = queue.pop(0)
  if (x,y) == (len(labyrinth)-1, len(labyrinth[0])-1):
      return constructPath(parent, (x,y))
  visited.add((x,y))
  for dx,dy in [(1,0), (0,1)]:
    new_x, new_y = x+dx, y+dy
    if (new_x, new_y) in visited or new_x >= len(labyrinth) or
    new_y >= len(labyrinth[0]) or labyrinth[new_x][new_y] == 1:
        continue
    queue.append((new_x, new_y))
    parent[(new_x, new_y)] = (x,y)
return NIL

constructPath(parent, end):
  path = [end]
  while path[-1] != (0,0):
     path.append(parent[path[-1]])
  return list(reversed(path))
```

Primjer:

\begin{tikzcd}
0 \arrow[r] & 0 \arrow[d] & 1 \\
1 & 0 \arrow[r] & 0 \arrow[d] \\
1 & 1 & 0
\end{tikzcd}

constructPath vraća:

obrnuto [(2,2), (1,2), (1,1), (0,1), (0,0)],

dobile [(0,0), (0,1), (1,1), (1,2), (2,2)]

(3.) Psevdokod:

graph_cube(vertices, matrix):

    n = len(vertices)

    G3 = {v: set() for v in vertices}

    A2 = [[0 for _ in range(n)] for _ in range(n)] // polnimo A2 z ničlami

    for i in range(n):

        for j in range(n):

            for k in range(n):

                A2[i][j] += matrix[i][k] * matrix[k][j]

    for u in vertices:

        for v in vertices:

            if matrix[u][v] != 0 or A2[u][v] != 0:

                ~~G3.add(v)~~

                G3[u].add(v)

    return G3

Primer:

GRAF:

matrica sosednosti: vozlišča: [0, 1, 2, 3]

0 1 0 0
0 0 1 0
0 0 0 1
0 0 0 0

$(GRAF)^3$:

vozlišča:

matrica:

0 1 1 0
0 0 1 1
0 0 0 1
0 0 0 0

CRTEŽ:

0 → 1 → 2 → 3

1. objašnjenje:
Algoritam prima korijen binarnog stabla i traženu sumu X, te vraca sve putove koji imaju tu sumu. Varijabla paths je lista u kojoj se pohranjuju putovi koji imaju trazenu sumu. Funkcija dfs je rekurzivna funkcija koja obilazi stablo u dubinu. Varijabla path je lista koja predstavlja trenutnu putanju do cvora, a paths je lista svih pronadenih putanja. U dfs-u prilikom svakog prolaska kroz stablo appendamo cvorove u path pa tražimo sumu tog puta. Ako je ona jednaka X, appendamo taj put u varijablu paths. Kada nademo sve pathove u varijabli paths, vratimo ih.

Vremenska slozenost ovog algoritma je O(n), gdje je n broj cvorova u binarnom stablu. To je zato sto algoritam izvodi prvo dfs pretrazivanje binarnog stabla i posjecuje svaki cvor tocno jednom. Vremenska slozenost funkcije dfs pozvane u funkciji findPaths takoder je O(n), budui da rekurzivno posjecuje svaki cvor u binarnom stablu tocno jednom. Medutim, buduci da se funkcija dfs poziva za svaki cvor u binarnom stablu, ukupna vremenska slozenost algoritma je O(n^2).

2. objasnjenje:
Ovaj algoritam prima matricu labyrinth koja predstavlja labirint, a vraca najkrai put od gornjeg lijevog do donjeg desnog dijela labirinta. Varijabla queue je red koji se koristi za BFS pretragu, a visited je skup posjecenih cvorova. Varijabla parent je rjecnik koji biljezi roditeljske cvorove svakog cvora koji je vec posjecen.

Algoritam poinje s dodavanjem pocetnog cvora (0, 0) u red. Zatim se redom obrauju svi cvorovi u queueu. Ako je trenutni cvor jednak krajnjem cvoru (len(labyrinth)-1, len(labyrinth[0])-1), tada se vraca put koji se može konstruirati pomou roditeljskih veza u rjecniku parent.

Inace, trenutni cvor se dodaje u skup posjecenih, a zatim se obraduju susjedni cvorovi koji su slobodni i još nisu posjeceni. Izracunamo sto bi se dogodilo da idemo dolje i da idemo desno. (zato for petlja za novi x i y koja prolazi kroz listu uredenih parova (1, 0) i (0, 1)). Ako smo izasli van granica labirinta ili dosli do zida (1), continueamo petlju. Ako su cvorovi slobodni i nisu posjeceni, dodaju se u red, a njihovi roditelji se zabiljezavaju u rjecniku parent.

Funkcija constructPath koristi roditeljske veze u rjecniku parent za konstrukciju puta od kraja do pocetka. Nakon toga reverseamo listu da dobijemo originalni put.

Vremenska složenost ovog algoritma je O(N * M), gdje su N i M dimenzije labirinta.To je zato sto u najgorem slucaju algoritam posjeti svaki cvor.

3. objasnjenje:
Kod kuba grafa bitno je gledati da dodamo brid ako i samo ako G sadrzi put s najvise dva brida izmedu u i v. Napravimo rjecnik G3 koji ce nam predstavljati kub grafa. Izracunamo matricu susjedstava pomnozenu samu sa sobom (A2) i gledamo ima li za dane vrhove bilo gdje jedinica (bilo to u matrici susjedstava ili u A2). Ako ima, to znaci da je izmedu ta dva vrha put koji ima najvise dva brida, pa ga dodamo u graf. Na kraju dobijemo kub grafa.
Vremenska složenost algoritma je O(n^3), s obzirom da smo morali raditi matricno mnozenje.

4. zadatak:

Koristit cu palindromska stabla.Palindromska stabla su usmjereni grafovi s dvije vrste rubova:
Insertion rub(ima tezinu)
Maksimalni palindromski sufiks (nema tezinu)

Insertion rub u -> v s nekom težinom x znaci da je vor v formiran umetanjem x na poetku i kraju niza na u. Kako je 'u' ve palindrom, rezultirajuci niz na cvoru v ce takoder biti palindrom. x ce biti jedan znak za svaki rub. Stoga, cvor može imati najviše 26 insertion rubova.

Maksimalni palindromski sufiks rub ce uvijek pokazivati na svoj string. Stvorit cemo sve palindromske podnizove i zatim vratiti zadnji koji smo dobili jer bi to bio najdulji palindromski podniz do sada. Buduci da palindromsko stablo pohranjuje palindrome prema redoslijedu dolaska znaka, najduzi ce uvijek biti na zadnjem indeksu naseg tree arraya.

Ovako ce izgledati node:

```
struct Node {
// start i end ruba
    int start, end;

    // duljina substringa
    int length;

    // insertion rub za svaki character
    int insertEdg[26];

    // suffix rub za current node
    int suffixEdg;
};
```

```
// current node prilikom insertanja u stablo
int currNode;
string s;
int ptr;

void insert(int idx)
{
    /* trazimo cvor X takav da s[idx] X S[idx]
    je maksimalni palindrom koji zavrsava na poziciji
idx
    iteriramo po suffix rubu od currNode da
pronademo X */
    int tmp = currNode;
    while (true) {
        int curLength = tree[tmp].length;
        if (idx - curLength >= 1
            and s[idx] == s[idx - curLength - 1])
            break;
        tmp = tree[tmp].suffixEdg;
    }

    /* X = string u nodeu tmp
     * provjerimo postoji li if s[idx] X s[idx] vec u stablu
    if (tree[tmp].insertEdg[s[idx] - 'a'] != 0) {
        currNode = tree[tmp].insertEdg[s[idx] - 'a'];
        return;
    }

   // pravimo novi node
      ptr++;

      // novi node je child od X s weightom kao i s[idx]
      tree[tmp].insertEdg[s[idx] - 'a'] = ptr;

      // duljina novog nodea
      tree[ptr].length = tree[tmp].length + 2;

      // end point za novi node
      tree[ptr].start = idx - tree[ptr].length + 1;
```

// setamo suffix rub za novi node. Pronalazimo string Y takav da je s[idx] Y S[idx] najduzi moguci palindromski sufiks za novi node

```
tmp = tree[tmp].suffixEdg;

    // novi node je current node
    currNode = ptr;
    if (tree[currNode].length == 1) {
        tree[currNode].suffixEdg = 2;
        return;
    }
    while (true) {
        int curLength = tree[tmp].length;
        if (idx - curLength >= 1
            and s[idx] == s[idx - curLength - 1])
            break;
        tmp = tree[tmp].suffixEdg;
    }
    tree[currNode].suffixEdg
            = tree[tmp].insertEdg[s[idx] - 'a'];
```

Vremenska slozenost algoritma biti ce O(k * n), gdje je n duljina niza, a k su dodatne iteracije potrebne za pronalaženje niza X i niza Y svaki put kada umetnemo znak u stablo.

npr. za string abba, algoritam vraca abba.