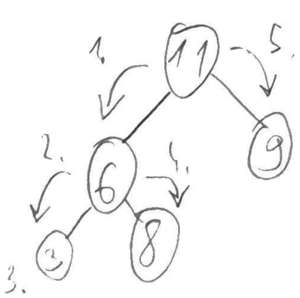
DZ 8

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:

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

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

(3.) Pseudokod:

```
graph_cube(vertices, matrix):
  n = len(vertices)
  G3 = {v: set() for v in vertices}

  A2 = [[0 for _ in range(n)] for _ in range(n)] //punimo A2 s nulama
  for i in range(n):
      for j in range(n):
          for k in range(n):
              A2[i][j] += matrix[i][k] * matrix[k][j]
  A3 = [[0 for _ in range(n)] for _ in range(n)] //punimo A3 s nulama
  for i in range(n):
      for j in range(n):
          for k in range(n):
              A3[i][j] += A2[i][k] * matrix[k][j]

  for u in vertices:
      for v in vertices:
          if A3[u][v] != 0 or matrix[u][v] != 0 or A2[u][v] != 0:
              G3[u].add(v)

  return G3
```

PRIMJER:

matrica susjedstva: vrhovi: [0,1,2,3]

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

matrica susjedstva na kvadrat:
0 0 1 0
0 0 0 1
0 0 0 0
0 0 0 0

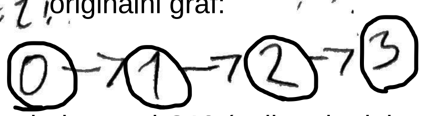
$(GRAF)^3$:

vrhovi isti

matrica susjedstva na treću:

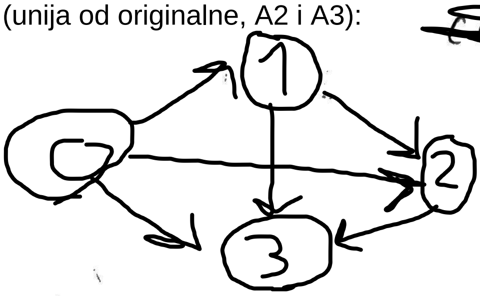
0 0 0 1
0 0 0 0
0 0 0 0
0 0 0 0

CRTEŽ: originalni graf:



matrica susjedstva od G^3 (unija od originalne, A2 i A3):
0 1 1 1
0 0 1 1
0 0 0 1
0 0 0 0

krajnji crtez:



1. zadatak objašnjenje:
Algoritam prima korijen binarnog stabla i traženu sumu X, te vraca sve putove koji imaju tu sumu u obliku liste lista gdje je svaka unutarnja lista jedan put. Varijabla paths je ta lista 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 rekurzivno pozivamo algoritam na lijevo i desno dijete (pretrazujemo po dubini). Kada smo dosli do lista (posto se u zadatku traze putovi tocno do lista) provjerimo je li suma jednaka nasem X-u. Ako je, appendamo nas path u paths i popamo iz njega da bi mogli ici na drugi cvor i njega dodati u path. Na kraju algoritma vracamo paths kao listu lista koja sadrzi sve putove do lista koji imaju sumu X.

Vremenska slozenost ovog algoritma je O(n) jer je dfs u pravilu na grafovima O(V + E), gdje je V broj vrhova a E broj bridova. U binarnom stablu broj vrhova je broj cvorova - 1, a broj cvorova je n, pa dobijemo O(n - 1 + n), sto je i dalje O (n).

2. zadatak objašnjenje:
Ovaj algoritam prima matricu labyrinth koja predstavlja labirint, a vraca najkraci put od gornjeg lijevog do donjeg desnog dijela labirinta. U ovom algoritmu koristimo BFS. Varijabla queue je red koji biljezi koordinate trenutnog cvora, a visited je skup posjecenih cvorova. Varijabla parent je rjecnik koji biljezi roditeljske cvorove svakog cvora koji je vec posjecen kako bi iz tih roditeljskih cvorova mogli reconstructati put do kraja labirinta.

Algoritam poinje s dodavanjem pocetnog cvora (0, 0) u red. Zatim se redom obrauju svi cvorovi u grafu (sve dok queue ne postane potpuno empty na kraju iteracije, onda smo dosli do kraja). Na pocetku dequeueamo pocetnu koordinatu. 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 imamo N * M cvorova, a znamo da BFS ima slozenost O(V + E), gdje je V broj vrhova a E broj bridova u grafu. BFS u najgorem slucaju razmatra svaki cvor u grafu, pa imamo O(N * M) slozenost.

3. zadatak objašnjenje:
Algoritam radi tako sto prvo napravi rjecnik koji ce predstavljati nas kubirani graf (moze se i reprezentirati matricom susjedstva, ja sam napravio rjecnik vrhova i vrhova s kojima su spojeni). Napravi A2 matricu koju cemo na pocetku popuniti nulama, a onda u nju spremiti produkt nase originalne matrice susjedstva sa samom sobom. Ta matrica A2 je matrica susjedstva originalnog grafa na kvadrat. Ona ce i posluziti da bi napravili matricu A3 koja ce biti A2 * matrica susjedstva, dakle nasa originalna matrica susjedstva na trecu. Kubirani graf se reprezentira kao unija matrice susjedstva originalnog grafa s matricom A2 i A3. Dakle, ako u toj uniji izmedu neka dva vrha u i v postoji brid, taj brid biti ce i u kubiranom grafu. Na kraju dobijemo taj kubirani graf. Ovaj algoritam ima kompleksnost O(n^3) zbog kubiranja matrice.

4. zadatak:

Prvo moramo napraviti funkciju koja testira je li dani string palindrom:

```
isPal(s):
  i = 0
  j = s.size() - 1
  while (i < j):
    if s[i] != s[j] return false
    i++
    j--
  return true
```

Uz pomoc dfs-a mozemo napraviti algoritam koji skace do sljedece pozicije u particiji stringa
ako je trenutni dio palindrom. Pseudokod algoritma:

```
ans = [[]]
dfs(s, solution):
  if (s.empty())
    ans.push_back(solution)
    return
  for (i = 0; i < s.size(); ++i)
    first = s.substr(0, i + 1)
    if (isPal(first)):
      solution.push_back(first)
      dfs(s.substr(i + 1), solution)
      solution.pop_back()
```

inicijalno solution = {}, na kraju samo provjerimo koji stringovi u ans stringovima imaju najvecu duljinu i njih vratimo (svejedno je koji ako ih ima vise).

Algoritam radi tako da provjerava je li string empty (jesmo li dosli do kraja). Ako je, u listu lista ans stavljamo nase palindrome (prvo ce biti jednoslovni itd). Ulazimo u for petlju i radimo prvo jednoslovni substring (koji je automatski palindrom) pa pushamo taj substring u solution. Ulazimo u rekurzivni poziv no bez prvog znaka. Dalje idemo po rekurziji (pri cemu ignoriramo grane koje nisu palindromi, zato if uvjet) itd. U biti, gradimo stablo u kojem cemo dfsom gledati sve moguce jednoslove, dvoslove itd. u stringu i ako su palindromi, pushamo ih u solution. Svi palindromi jednoslovi, dvoslovi itd. ce biti u svojoj solution listi jer cemo doci do kraja stringa, pa u ans listi lista saveamo te palindrome (prva lista u ansu ce biti jednoslovi (automatski palindromi), druga dvoslovi palindromi itd). Na kraju gledamo koja lista u ansu ima string najdulje duljine, njega vratimo.

Posto za svako slovo pokusavamo napraviti svaku kombinaciju jednoslova, dvoslova itd. vremenska slozenost ovog algoritma biti ce O(n * 2^n), gdje je n duljina originalnog stringa.

npr. za aba, prvo nije empty pa ulazimo u for petlju. first je sada samo slovo 'a'. Pushamo ga u solution i pozivamo dfs na 'ba'. Ulazimo opet u for petlju i first je sada slovo 'b'. Pushamo to u solution i pozivamo dfs za 'a'. Pushamo ponovo a u solution i sada imamo empty string u dfs pozivu, pa listu ['a', 'b', 'a'] pushamo u ans. Posto smo dosli do kraja u dfsu, idemo nazad na for petlju. Sada imamo substring 'ab' kao first. On nije palindrom. Imamo substring 'aba' kao first, on je palindrom i pushamo ga u solution. Sada zovemo dfs na s.substr(3), sto je prazan string pa pushamo u ans ['aba']. Dalje nemamo palindroma pa ce nas ans biti [['a', 'b', 'a'], ['aba']]. Nademo string max duljine, sto ce uvijek biti string u zadnjem clanu liste lista, u ovom slucaju 'aba'.