# 700803401908

# **ТУРГИНБАЕВА Кулжанат Абдраимовна,**

**Саттар Ерубаев атындағы №24 ІТ мектеп-лицейінің**

**информатика пәні мұғалімі.**

**Шымкент қаласы**

# **ПРОГРАММАЛАУДАҒЫ КҮРДЕЛІ ЕСЕПТЕР: PYTHON БАҒДАРЛАМАЛАУ ТІЛІНДЕ КҮРДЕЛІ ЕСЕПТЕРДІ ШЕШУ ЖОЛДАРЫ**

**Аңдатпа**

Бұл мақалада Python бағдарламалау тілін қолдана отырып, күрделі есептерді шешу әдістері қарастырылады. Әртүрлі алгоритм мен деректер құрылымдарын пайдалану арқылы есептердің тиімді шешімін табу жолдары талданады. Сонымен қатар, есептеу тиімділігін арттыру тәсілдері мен оңтайландыру әдістері ұсынылған.

**Тірек сөздер:** Python, олимпиадалық есептер, алгоритм, деректер құрылымы, графтар.

**Кіріспе**

Python - қарапайымдылығымен және қуаттылығымен ерекшеленетін бағдарламалау тілі. Бұл тілде алгоритмдік және математикалық күрделі есептерді тиімді шешуге болады. Python бағдарламалау тілінің көмегімен әр түрлі күрделі есептерді шешу жолдарын қарастырамыз және бірнеше мысал келтіреміз.



## **Негізгі бөлім.**

## 1. Алгоритмдердің тиімділігін бағалау

Күрделі есептерді шешкен кезде алгоритмнің тиімділігі маңызды рөл атқарады. Алгоритмнің уақыт пен жады күрделілігін (Big-O нотациясы) бағалау қажет. Мысалы:

* O (1) – тұрақты уақыт
* O (log n) – логарифмдік уақыт
* O(n) – сызықтық уақыт
* O(n^2) – квадраттық уақыт
* O(2^n) – экспоненциалдық уақыт

**Мысал:**

import time

def factorial(n):

if n == 0:

return 1

return n \* factorial (n - 1)

start\_time = time.time()

print (factorial (10))

end\_time = time.time()

print ("Execution Time:", end\_time - start\_time)

Бұл рекурсивті тәсіл, оның уақыт күрделілігі O(n) болады.

## 2. Динамикалық бағдарламалау

Күрделі есептерді шешудің бір тәсілі – динамикалық бағдарламалау. Бұл әдіс қайталанатын есептеулерді болдырмау арқылы тиімділікті арттырады.

**Мысал:** Фибоначчи сандарын есептеу

def fibonacci(n, memo={}):

if n in memo:

return memo[n]

if n <= 2:

return 1

memo[n] = fibonacci(n-1, memo) + fibonacci(n-2, memo)

return memo[n]

print(fibonacci(50))

Бұл әдістің уақыт күрделілігі O(n), ал жады күрделілігі O(n).

## 3. Графтар және жол табу алгоритмдері

Графтар – көптеген нақты өмірлік есептерді шешуге арналған қуатты құрал. Ең жиі қолданылатын алгоритмдер:

* Dijkstra алгоритмі (ең қысқа жол)
* Floyd-Warshall алгоритмі (барлық төбелер арасындағы ең қысқа жол)
* A\* алгоритмі (эвристикалық жол табу)

**Мысал:** Dijkstra алгоритмі

import heapq

def dijkstra(graph, start):

queue = [(0, start)]

distances = {node: float('inf') for node in graph}

distances[start] = 0

while queue:

current\_distance, current\_node = heapq.heappop(queue)

if current\_distance > distances[current\_node]:

continue

for neighbor, weight in graph[current\_node]. items ():

distance = current\_distance + weight

if distance < distances[neighbor]:

distances[neighbor] = distance

heapq.heappush(queue, (distance, neighbor))

return distances

graph = {

'A': {'B': 1, 'C': 4},

'B': {'A': 1, 'C': 2, 'D': 5},

'C': {'A': 4, 'B': 2, 'D': 1},

'D': {'B': 5, 'C': 1}}

print (dijkstra(graph, 'A'))

Бұл алгоритм O((V+E) log V) уақыт күрделілігімен жұмыс істейді, мұнда V – төбелер саны, ал E – қабырғалар саны.

## 4. Ақпараттарды өңдеу және үлкен деректермен жұмыс істеу

Python үлкен деректерді өңдеуге арналған қуатты құралдар ұсынады: NumPy, Pandas, Dask, және т.б.

**Мысал:** Pandas көмегімен үлкен CSV файлын оқу

import pandas as pd

df = pd.read\_csv("large\_data.csv", chunksize=10000)

for chunk in df:

print(chunk.head())

Бұл код үлкен көлемдегі мәліметтерді тиімді өңдеуге көмектеседі.

## Қорытынды

Python күрделі есептерді шешуге арналған күшті құралдар мен кітапханаларға ие. Алгоритмдердің тиімділігін түсіну және дұрыс әдістерді таңдау есептерді жылдам әрі оңай шешуге көмектеседі.