Big O notation (uz)

Assalomu alaykum.

Demak, ushbu videoda **Algorithm va Data Structure**'ning eng muhim fundamental mavzularidan biri bo'lgan **Complexity**, ya'ni **murakkabliklar** haqida gaplashamiz.

Eng muhim savol:

> **"Bu algoritm qanchalik tez ishlaydi va qancha xotira ishlatadi?"**

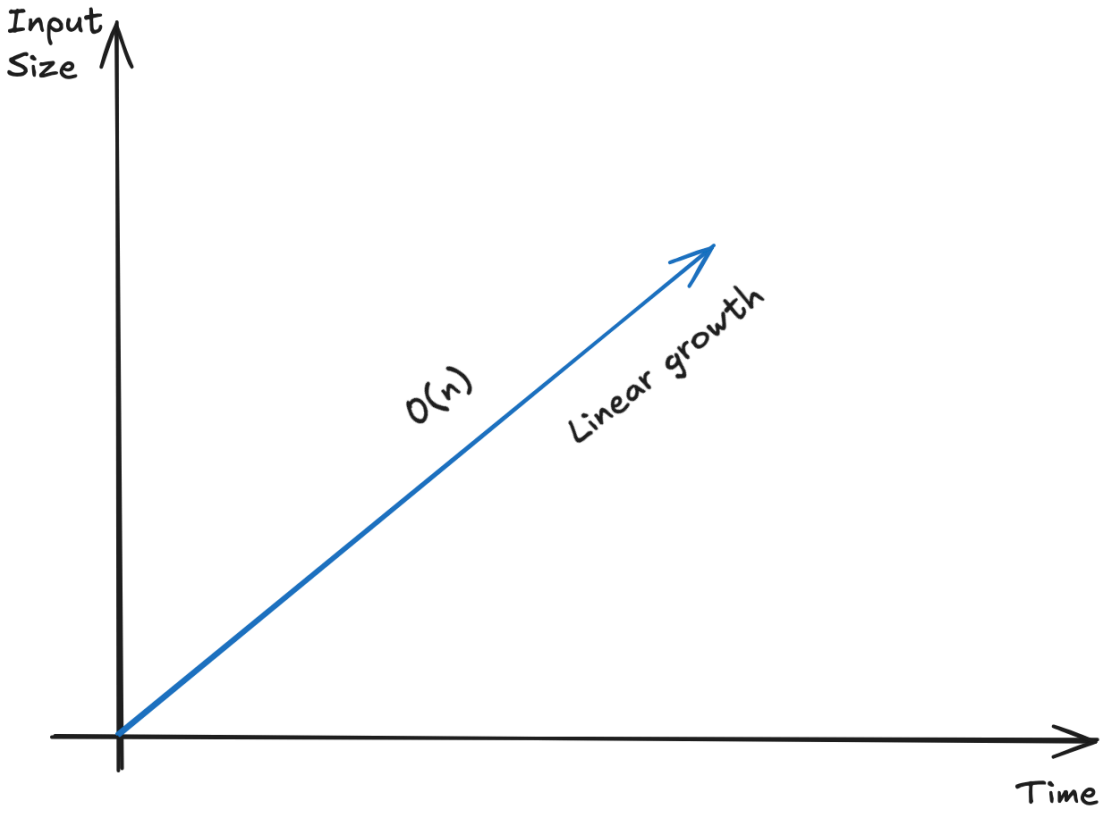
Shu tushuncha orqali:

- **Big O Notation** nima ekanligini,
- **O(1), O(log n), O(√n), O(n), O(n log n), O(n^2), O(2^n), O(n!)** kabi barcha asosiy complexity'larning qanday ishlashini,
- ularning bir-biridan asosiy farqlarini,
- qaysi turdagi muammolarda ishlatilishini,
- har biri uchun real LeetCode masalalarini,
- masalalarni avval matematik va mantiqiy model orqali tahlil qilishni

bosqichma-bosqich ko'rib chiqamiz.Birinchi topic bizda bu:

Big O notation - bu degani biz yechim berayotganda: undagi input hajmi qanchalik o’zgarsa, bajargan vaqt ham shuncha o’zgarishini ifodalaydi. Uni grafik funksiyasini chizsak quyidagicha bo’ladi:

Linear Time



Real life example bilan ko'rsak: Tasavvur qilaylik bizda qoplar bor, qator bo'lib turibdi. Va ichidagi bittasida ticket bor, qaysida ekanini biz bilmaymiz. agarda qidirishni boshlaganimizda birinchisidan chiqsa, demak boshqalarini tekshirishni hojati qolmaydi, birinchi iterationda topildi. Agar oxirigacha tekshirganda ham chiqmasdan, eng oxirida chiqsa, demak 10 ta bo'lsa 10 marta iteration, 1000ta bo'lsa 1000marta iteration. Manashu eng worst case ni hisoblaydigan (vaqt va xotira) bo'yicha jarayon: BigO notationning asosiy concepti. Bu conceptni real problem-solving da qo'llashda odatda `Algorithm analysis` orqali `Time complexity` va `Space complexity` conceptlari orqali yechim beramiz.

Linear time ni biz asosan: Unsorted massivlardan biror elementni qidirishda (Odatda Linear search deb ataymiz), massiv elementlarini yig'ish yoki orasida qaysinidir update qilishda, HashMap Hashset yordamida tekshirishda ishlatamiz.

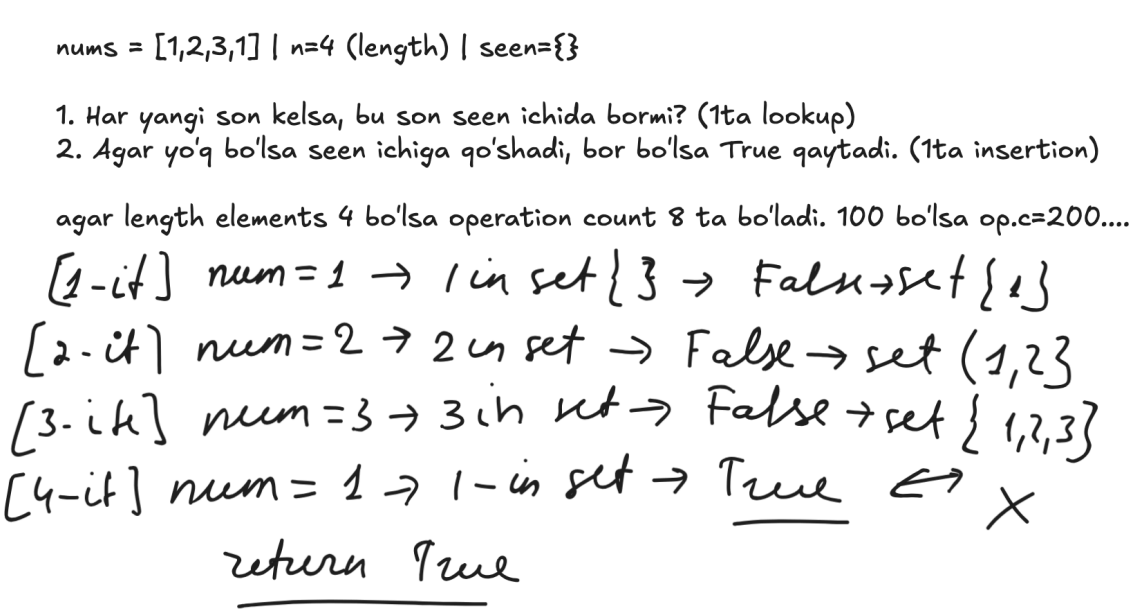
Leetcodedan mavzuga aloqador Contains dublicate masalasini ko'radigan bo'lsak: (https://leetcode.com/problems/contains-duplicate/)

- `Bizga n ta butun sonlardan iborat nums massiv berilgan. Aagar massivda kamida bitta qiymati kamida 2marta takrorlansa True, agar barcha elementlar unique bo'lsa False qaytarsin`

- Misol: nums=[1,2,3,1]
- Output: True (chunki 1soni 2marta takrorlandi)

Demak O(n) Linear time bo'yicha solution berishda: HasSet (to'plam) structuredan foydalansak bo'ladi, chunki biz har bir elementni pair qilib birma bir oxirigacha taqqoslab chiqishimiz shart emas, birorta seen topsak, bo'ldi to'xtatib qoya olamiz. topilmasa eng worst case da hammasini o'tib chiqadi.

1. Ya'ni bitta bo'sh to'plam yaratib olamiz. 2. massiv bo'yicha loop qilib o'tib chiqamiz. 3. agar seen bo'lsa takrorlash topiladi va True qaytaramiz. agar bo'lmasa False.

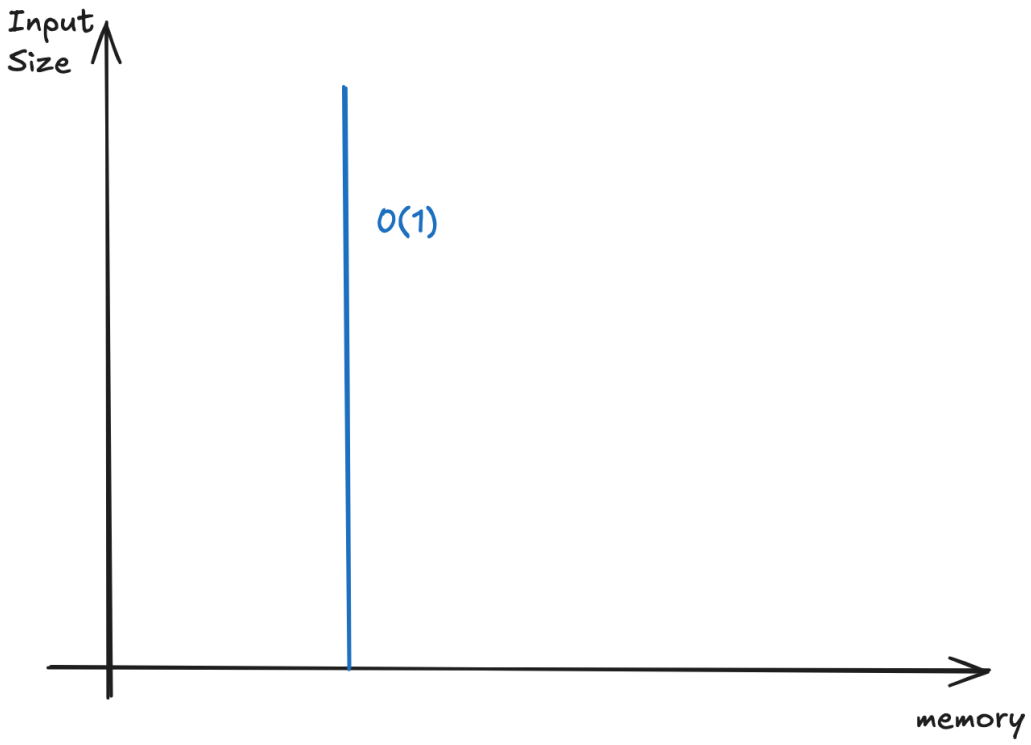



```python
def containsDublicate(nums: list[int]) -> bool:
    seen = set()
    for num in nums:
        if num in seen:
            return True
        seen.add(num)
    return False
```

Demak bizda navbatdagi topic: O(1) Constant time.

- Uni ikkinchi nomi Big O of 1. Ya'ni algorithmda input hajmi qancha kattalashsa ham algorithm bajaradigan operatsiya bir xil qolaveradi,

o’zgarmaydi. Endi uni funksiyasini ko’rsak, quyidagicha bo’ladi:



K chiziq qanchaga oshganda ham, doim bir xil xotira kerak bo’ladi. Masalan:

```
def get_first(numbers: list[int]) -> int:
    return numbers[0]
```

agar array da 5ta, 100ta, yoki 1million element bo’lsa ham, biz faqat 0-indexga murojaat qilamiz. funksiya umumiy 10ta operation bajarsa ham Constant time 1 bo’lib qoladi.

Square Root Time — O($\sqrt{n}$)

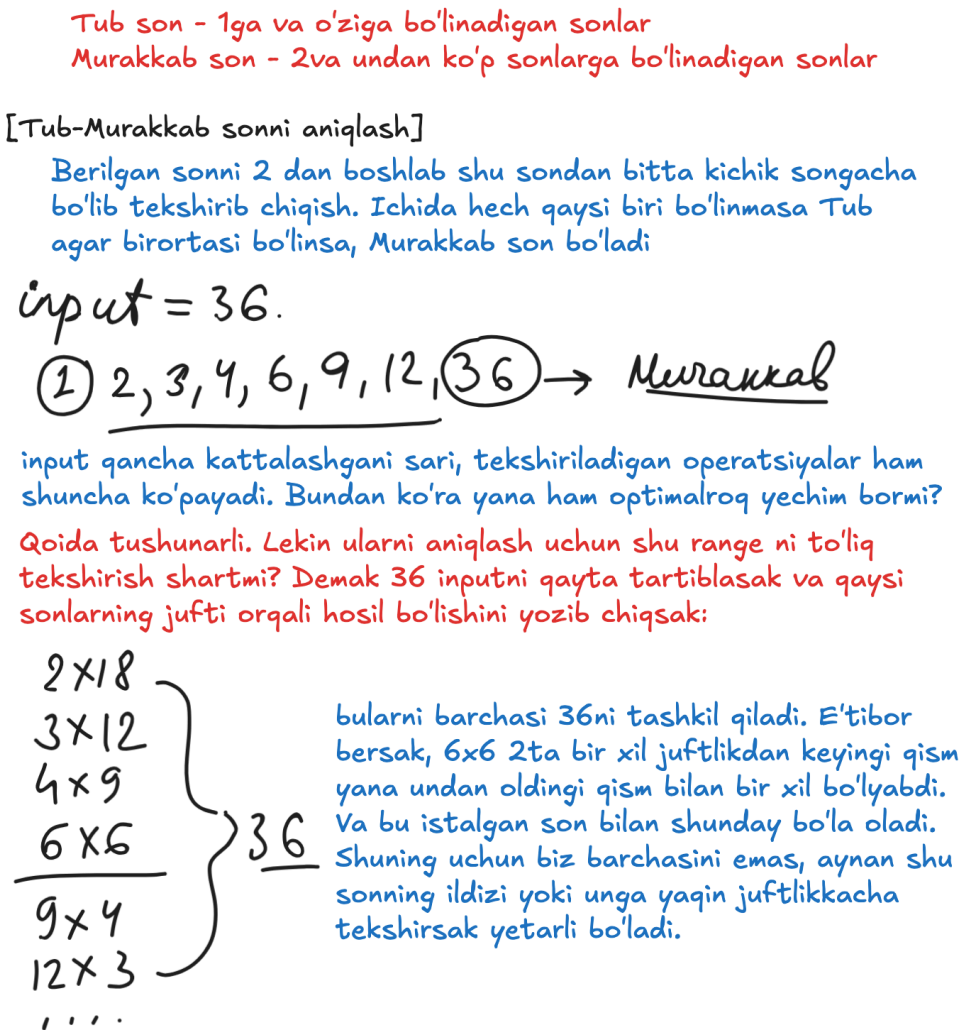
- Algorithmning bajarilish vaqti n ning kvadrat ildiziga ($\sqrt{n}$) mutanosib ravishda o’sadi.
    - Ya’ni inputning qiymati n bo’lsa, algorithm eng yomon holatda taxminan ($\sqrt{n}$) ta qadam bajaradi. Misol uchun:

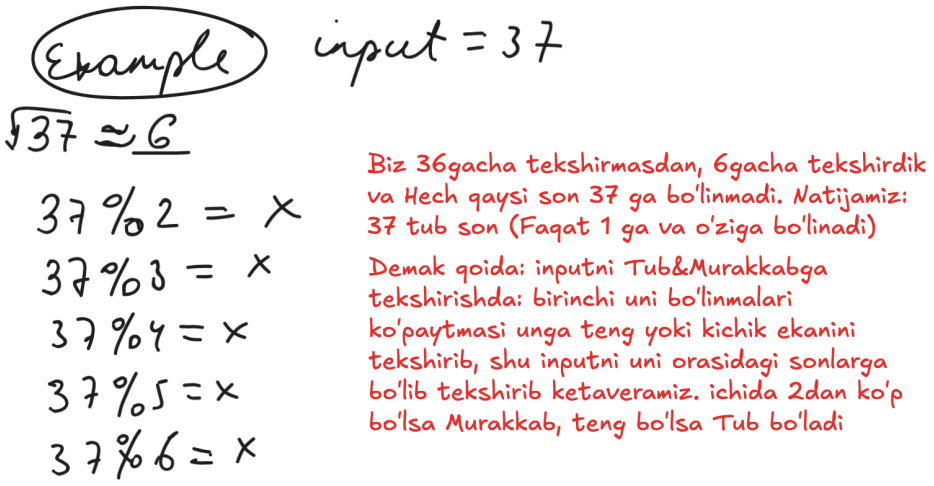
(n)	($\sqrt{n}$)	Taxminiy qadam
16	4	4
100	10	10
10 000	100	100

Shuning uchun bu O(n) dan ancha tezroq, lekin O(log n) dan biroz sekinroq ishlaydi.

  - Real misollardan: odatda Square root time Tub sonlarni topishda, Sqrt decomposition larni topishda foydalaniladi.

- **Masala:**
  - n = 36 sonining tub yoki murakkab son ekanligini tekshirish uchun uning bo'luvchilarini qidirish algoritmini tuzish. 36 ning bo'luvchilarini topish uchun nechta qadam kifoya qiladi?



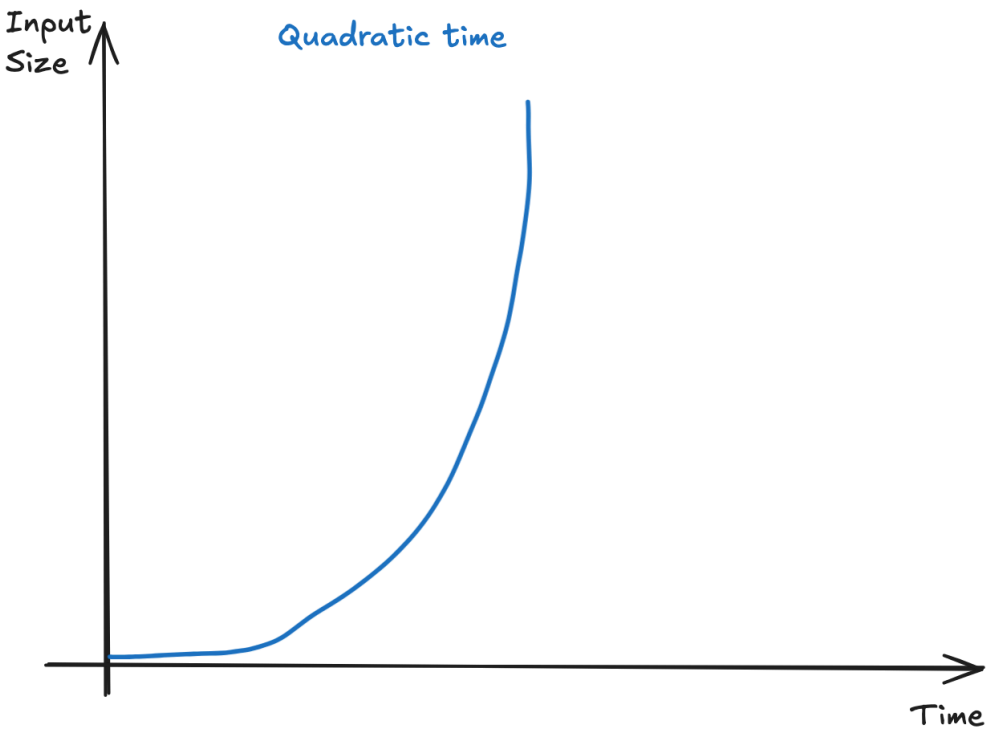



```python
class Solution:
    def isPrime(self, n: int) -> bool:
        if n <= 1:
            return False

        divisor = 2

        while divisor * divisor <= n:
            if n % divisor == 0:
                return False

            divisor += 1

        return True
```

$O(n^2)$ — Quadratic time.

- Quadratic time o’z nomi bilan input size oshgani sari, uni bajarishga ketadigan vaqt, kvadratik ravishda oshib borishini bildiradi. misol uchun elementlar soni 2 baravar oshsa, uni qayta ishlash vaqti 4barobar, 10 baravar oshsa, 100 barobar, oshadi va time space nuqtai nazaridan shuncha marta parllel sekinlashadi. Bizga Quadratic time nimaga kerak: asosiylari:
    - Anti-patternlarni aniqlash, katta hajmdagi ma’lumotlar bilan ishlanganda qaysi approachlar unda lag (qotirib qo’yish) ni oldindan bilan

- Optimization. Qaysi yechim optimalroq $O(n^2)$ yoki $O(n \log n)$, $O(n)$ qaysi optimalroq ekanini aniqlash uchun



Endi umumiy maqsadi nima ekanini tushuntirsak:

- `Brute Force` (Qo'pol yechim) nomli tushuncha bor: ya'ni har qanday murakkab masalani yechishda odatda birinchi bo'lib miyyaga keladigan eng sodda, xatosiz va to'g'ridan to'g'ri yechim sifatida foydalanishdan boshlanadi. Keyin shuni **Baseline** sifatida olib, yechimni optimallashtiriladi. Quadratic time ni asosan massiv sonlar orqali saralash yoki taqqoslash kabi (Bubble, Selection, Insertion sort algorithmlarida ishlatiladi)
- **Coding tomondan misol ko'rsak (Leetcode ni 1-misoli)**: Masala: Bizda n ta butun sonlardan iborat nums massivi va bitta target soni berilgan. Yig'indisi target ga teng bo'lgan 2ta turli indexdagi elementlar indexlarini topish kerak.
  - Misol: nums = [2,7,11,15], target=9
  - Output: [0,1] (**because: nums[0]+nums[1] = 2+7=9**)

  Matematik analyse:

Bizga n ta butun sonlardan iborat nums massivi bor va bitta target son berilgan. Yig'indisi target ga teng bo'lgan 2ta turli indexdagi elementlarning indexlarini toping
Example: nums=[2,7,11,15], target=9
Output: [0,1]
Because nums[0]+nums[1]==9

Yechish:
High level:

1. listdagi bitta elementni ushlab turamiz. (i nomli index misol).
2. i index yonidagi barcha boshqa elementlar bilan solishtirib chiqamiz (har birini j bilan nomlab)
3. Agar tanlab olingan nums[i]+nums[j]==target bo'lsa, demak shu indexdagi sonlarni qaytaramiz.

Low level:

1. i=0 index bo'lganda, j elementni (n-1) marta tekshirib chiqadi. (sabab: o'zini o'ziga qo'shib qo'ymasligi uchun)
2. i=1 bo'ganda, j elementni (n-2) mata tekshiradi (oldingini qayta tekshirmaslik uchun)
.....
toki oxirgi elementgacha

Bu matematikada arifmetik progressiya formulaisni beradi:

$$S=(n-1)+(n-2)+\dots(n-3); \quad S=\frac{a_1+a_n}{2}\cdot n$$

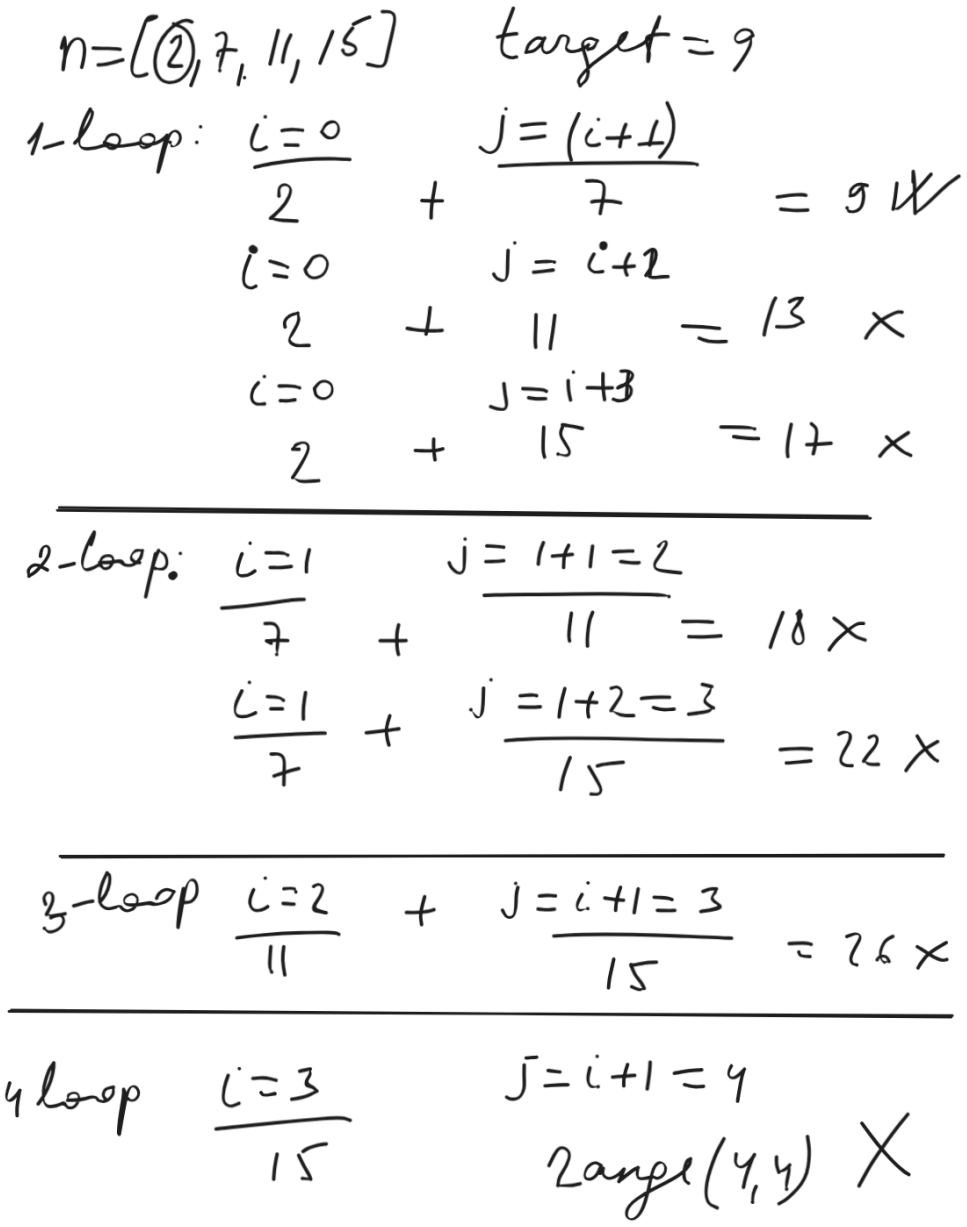
Bizda birinchi had a(1)=1 bo'lgani (ya'ni istalgan yig'indi oxirida 1 gacha borgani uchun) va umumiy hadlar soni (n-1) deb olganimiz uchun formula o'zgardi:

$$S=\frac{1+(n-1)}{2}\cdot(n-1)=\frac{n\cdot(n-1)}{2}=\frac{n^2}{2}-\frac{n}{2}$$

$$O\left(\frac{n^2}{2}-\frac{n}{2}\right)=O(n^2)$$

Quadratic square time ni isbotlangan formulasi bo'ladi

Code based analyse:

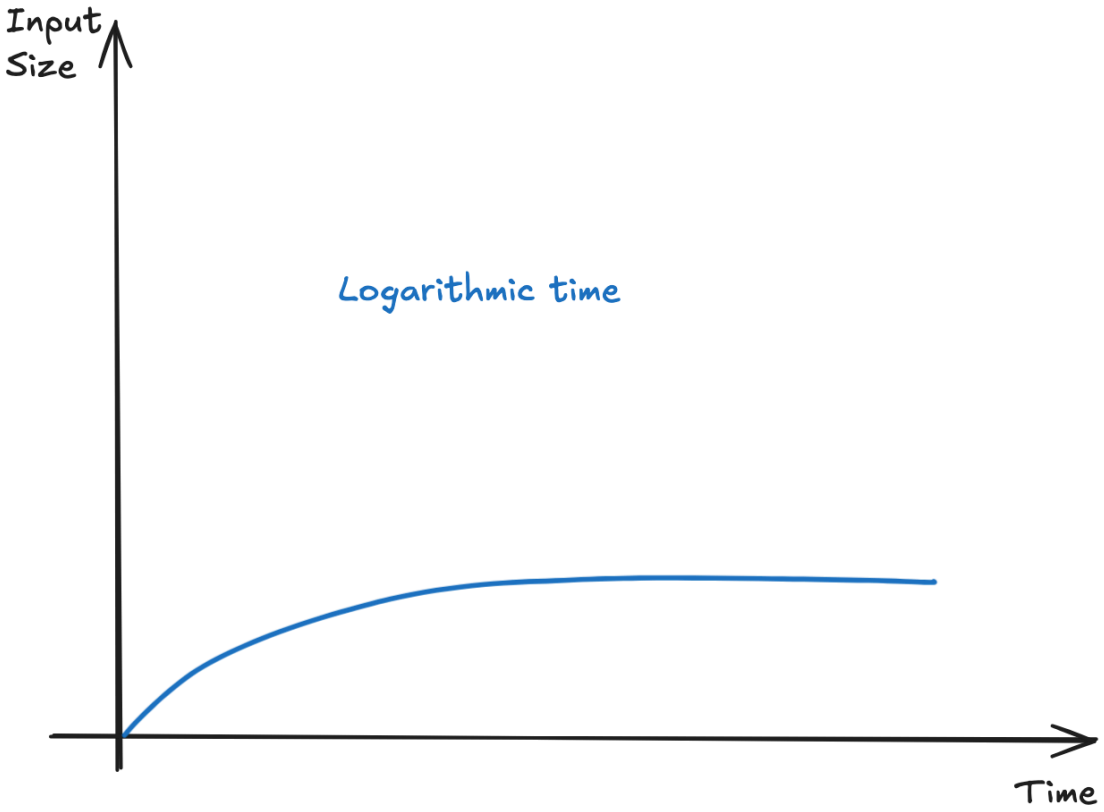



```python
def twoSum(nums: list[int], target: int) -> list[int]:
    n = len(nums) #birinchi nums list ni uzunligini olamiz
    for i in range(n): # outer loop orqali har safar bitta
    # elementni tanlash uchun kerak bo'ladi. i ni oli shda
        for j in range(i + 1, n): # inner loop -> i+1
```

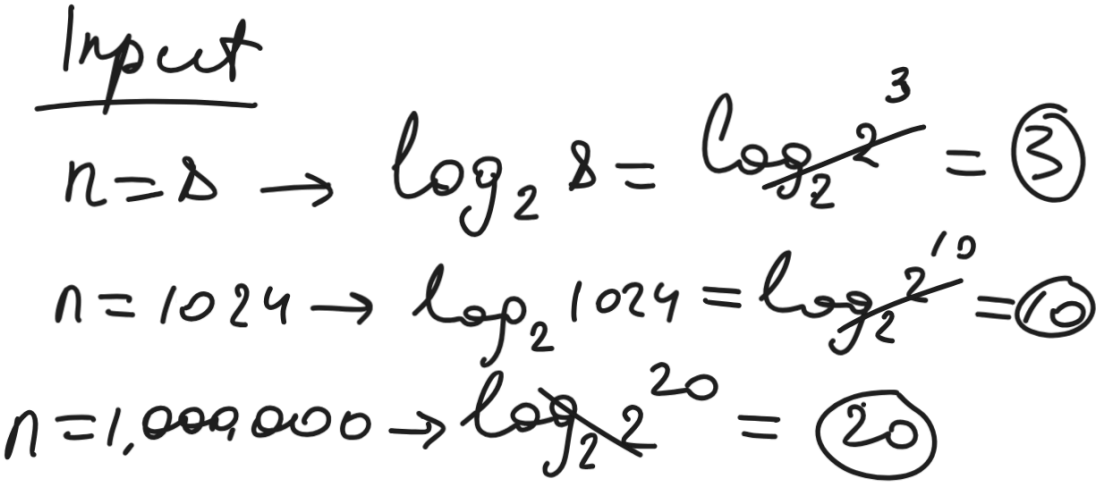
```
qilamiz, sabab index o'zini o'zi qayta hisoblab qo'ym
asligi uchun.(misol uchun 2+7 ham 7+2 ham bir xil nat
ija beradi) va n ni yuqoridagi massiv indexlar bilan
birma bir ko'rib chiqaveradi.
            if nums[i] + nums[j] == target: # olingan
indexlarni qo'shib target bilan tekshiramiz.
                return [i, j]  # qaysi teng bo'lgan i
va j indexlarni yig'amiz.
    return []   #ikkinchi return esa ifga kirmasa, y
a'ni yechim chiqmasa birortasida ham shunda bo'sh qay
taradi.

# Local Test 1
print(twoSum([2,7,11,15],9))

# Local Test 2
print(twoSum([1,2,3],100))
```

O(log n) — Logarithmic time

Bunda berilgan input size har bir iterationda ma’lum bir nisbatda qisqarib borish jarayonini ko’rsatadi. Bu odatda optimal complexitylardan biri hisoblanadi:



chunki input size oshsa ham operation countlar sekin o’sib boradi. Misol uchun matematik holat:

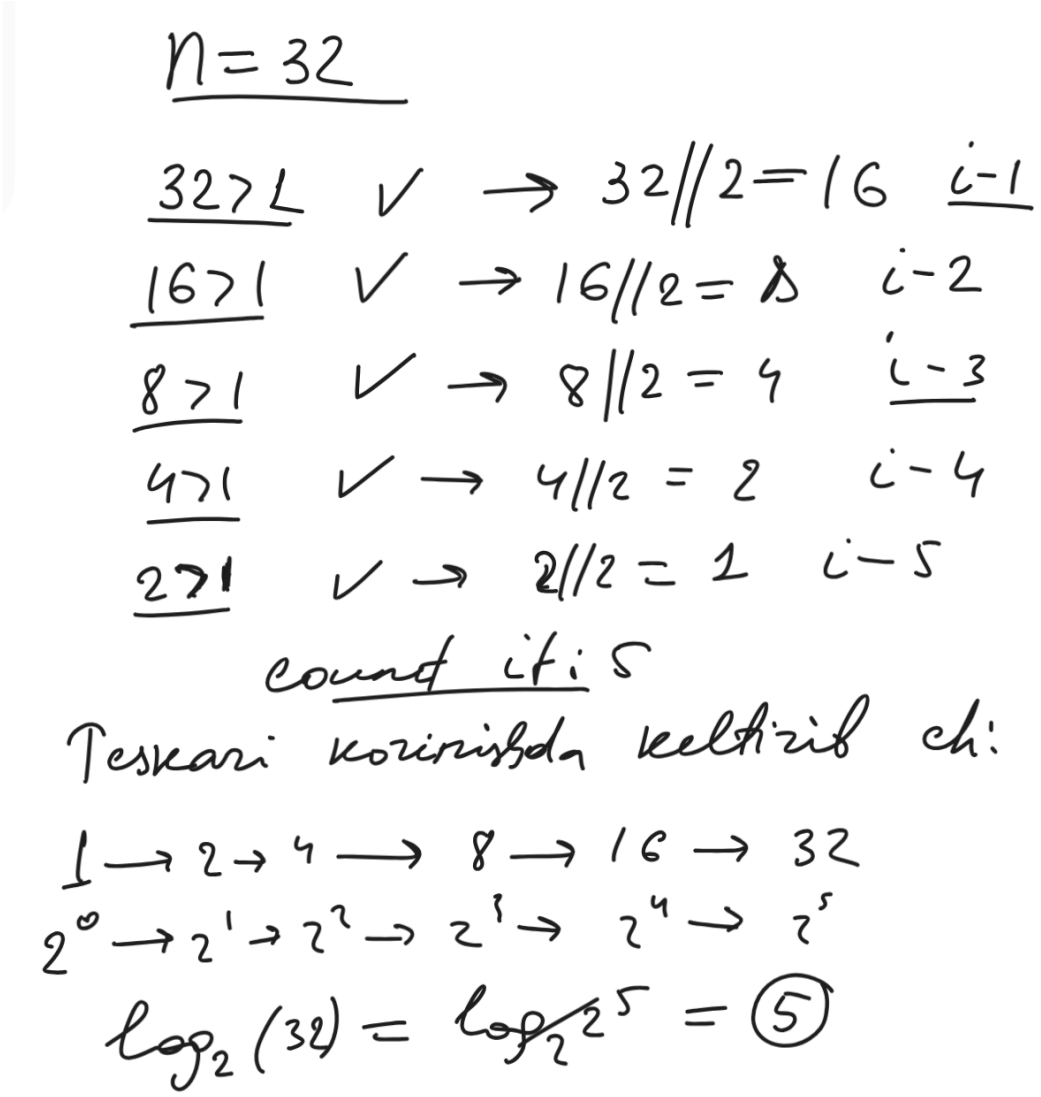


Bunda odatda biz logarifm ni 2asosga ko’ra holati bo’yicha hisoblasak bo’ladi. agar har qadamda 3ga bo’linsa, unda qadamlar soni ham shunga mos log 3 asosga bog’liq holda hisoblanadi. . Endi bularni ko’rsak: input size 8 bo’lsa

operatsiyalar soni 3ta bo’ldi. input size 1024 ta bo’lsa 1024 emas, 10 ta bo’ldi. 1M bo’lsa 20ta operatsiya bilan yechimga yetib bordi. Bu ko’pchiligimiz bilgan Binary searching ham asosiy poydevori hisoblanadi. Real life misollarda qaysi jarayonlarda ishlatiladi:

- Masalan Contact bookdan qaysidir Ism-familiya ni harfi bo’yicha qidirish. Biz M harfi bilan boshlanaidgan so’zni topish uchun 1-sahifadan boshlab birma-bir varaqlash (BigO bo’yicha) yurish shart emas. Shunchaki o’rtasidan ochamiz, qaysi tomonga qarab yurishni taqqoslab ketaveramiz. (Ya’ni Binary Search, Binary Tree algorithmlari shu concept bo’yicha ishlaydi)

- Bitta sodda algorithmic masala ko’rsak: Berilish sharti: **Berilgan musbat son n, toki butun qismi 1ga teng bo’lmaguncha 2ga bo’lib chiqqanda, necha qadam bilan bajariladi ?**
  - Matematik analyse:




```python
def divide_until_one(n:int) -> int:
    steps = 0
    while n>1:
        n//=2  #nni 2ga bo'lib butun qismini qayta n ga ber
ish
        steps += 1  # qadamlar sonini +1ga oshiramiz
    return steps
```

- Endi Leetcodedan bitta example ko’rsak bu bo’yicha (https://leetcode.com/problems/binary-search/) : `Sizga o'sish tartibida saralangan (sorted) n ta butun sonlardan iborat nums massivi va bitta target soni berilgan.Agar target massiv ichida bo'lsa, uning indeksini qaytaring. Agar bo'lmasa, -1 qaytaring.`

  `bajarilish vaqti O(log n) bo'lishi shart.`

- Example:
  - Input: `nums = [-1, 0, 3, 5, 9, 12]`, `target = 9`
  - Output: `4` (chunki `9` soni 4-indeksda joylashgan)

Matematik analyse:

Buni ishlash uchun birinchi bo'lib Binary searchni umumiy tushunishimiz kerak. Demak real life example bilan: qo'limizda 1million betlik lug'at bor. Index siz. Men **Apple** nomli so'zni topmoqchiman. Birinchi betdan boshlab qarasam 1,2,3.....1.000.000 gacha borishim mumkin. Bu O(n) bo'ladi. 2-usul: shunchaki o'rtasini ochib ko'raman, qaysi harf bor ekan. Deylik **M** harflik so'zlar, demak men oldinga o'tib ketibman, qolgan qismni ignore qilib, undan oldingi qismni yana shunday tekshirib boraman. Har safargi qidirish maydonning qolgan qismining yarmini qisqartirib boriladi.

$$n = 2^k \implies \mathbf{k = \log_2 n}$$

Shu orqali effective qidirish bo'ladi. Manashu Binary search. Sharti faqat albatta u sort qilingan bo'lishi kerak.

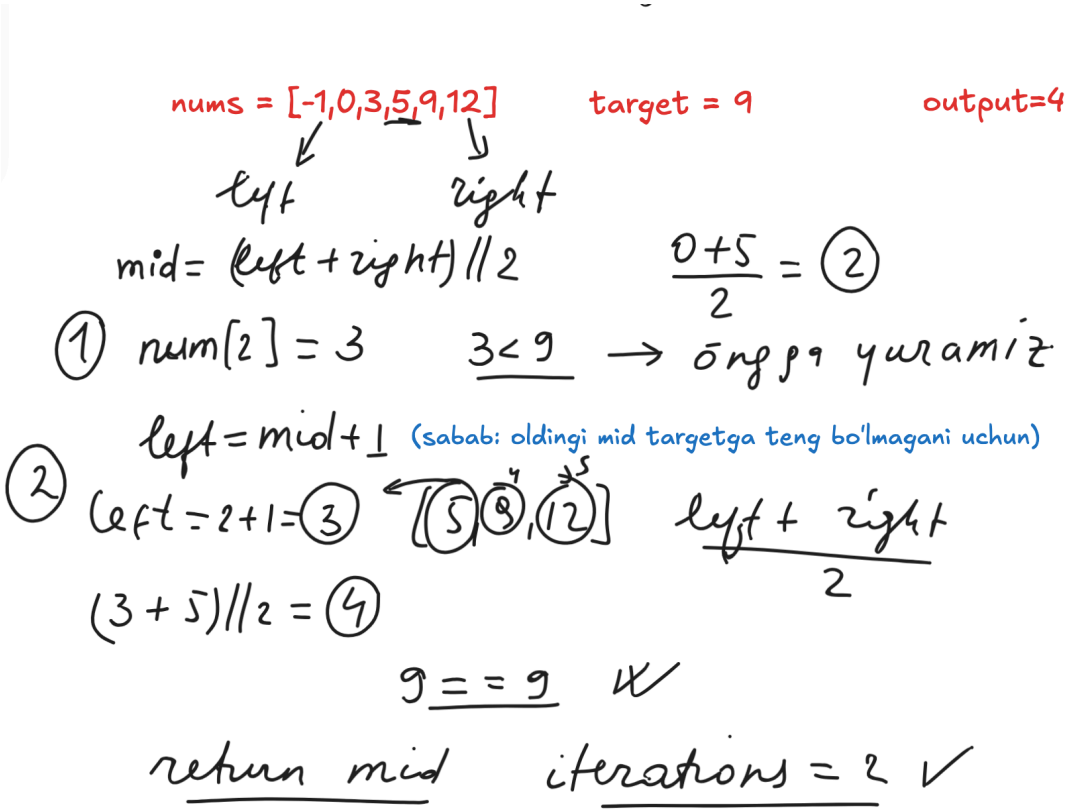
Code anlayse tomondan:

```
for i in range(len(nums)):
    if nums[i]==target:
        return i
```

desak bo'ladi, lekin bu yuqoridagidek O(n) bo'lib qidiradi. Bizga optimalroq yo'l kerak:

Demak yechim: **Binary search**

> 1-ish: nums arraydagi sonlarni indexi bo'yicha olamiz, o'rtasidan ajratib, target bilan taqqoslaymiz, katta bo'lsa o'nngga kichik bo'lsa chapga qarab. qolganini skip qilib yuboraveramiz.




```python
def search(nums: list[int], target: int) -> int:
    left = 0          #left tomon doim 0-indexdan bosh
lanadi
    right = len(nums) - 1 #rangde index oxirgisini ol
inadi

    while left <= right:#toki chap tomon o'ng tomonda
n
    # o'tib ketmaguncha [element tugamaguncha] tekshi
radi)
        mid = (left + right) // 2        # 4-qator

        if nums[mid] == target:          # 5-qator
            return mid                   # 6-qator
        elif nums[mid] < target:         # 7-qator
            left = mid + 1               # 8-qator
        else:                            # 9-qator
            right = mid - 1              # 10-qator
```

```
    return -1   # agar sikl tugab, element topilmasa
-1
```

- **Exponential Time $O(2^n)$**
  - bunda input size har safar 1taga oshganda, algorithm bajarish vaqt va operation count **2barobarga** ko'payishini biltiradigan complexity (Sodda tilda aytganda Binary search ni terkari holatda ishlashi). shuning uchun nomi Exponential deyiladi.

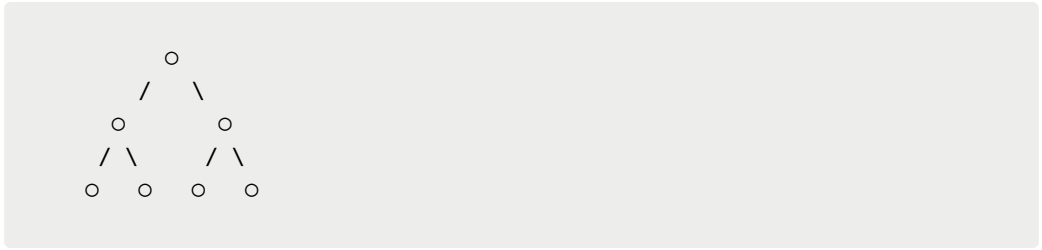
```
2¹=2

2²=4

2³=8

2⁴=16

2⁵=32
```

  - Bu complexitylar odatda har bosqichda rekursiya yoki har bir element uchun olish yoki olmaslik qarori qabul qilinganda paydo bo'ladi.
  - Shunday muammolar borki, ularni har bir caselarini ko'rmasdan turib to'liq solution berib bo'lmaydi. Masalan bizda 5ta lampochka bor. Har birida **ON** yoki **OFF** buyruqlari bor. Endi savol: umumiy nechta holat chiqadi barchasida. 2**5=32. Algorithm hamma holatni ko'rishga majbur. shuning uchun O(2**n)
  - Yana bitta na'munaviy misol:
    - n berilgan, bu n ta bit degani. Hamma mumkin bo'lgan binary stringlarni chiqarishimiz kerak. masalan: n=2 bo'lsa output:

```
00

01

10

11
```

      Versiyalar 4ta.

n=3 bo’lsa huddi shunday: [000,001,010,,011,100,101,110,111] 8ta = 2**3 degani.



- Leetcodeda #LeetCode #78 masalasini ko’rsak bo’ladi (https://leetcode.com/problems/subsets/)
  - `Sizga unikal butun sonlardan iborat nums massivi berilgan. Uning barcha mumkin bo'lgan` **`qism to'plamlarini (Power Set)`** `qaytaring. Javobda takroriy to'plamlar bo'lmasligi kerak`
  - Input: nums = [1,2]
  - output: [[], [1], [2], [1,2]]
    - Ya’ni birinchi listda: hech qaysi, ikkinchi-uchinchida: alohida va to’rtinchida: ikkalasi ham.

  Demak subset degani o’zi - original to’plam ichidagi elementlarning ayrimlarini tanlab hosil qilingan to’plamdir.

  - misol uchun Original: [1, 2, 3]
  - subsets:

```
[]
[1]
[2]
[3]
[1, 2]
[1, 3]
[2, 3]
[1, 2, 3]
```

To move canvas, hold (Scroll wheel) or (Space) while dragging, or use the hand tool

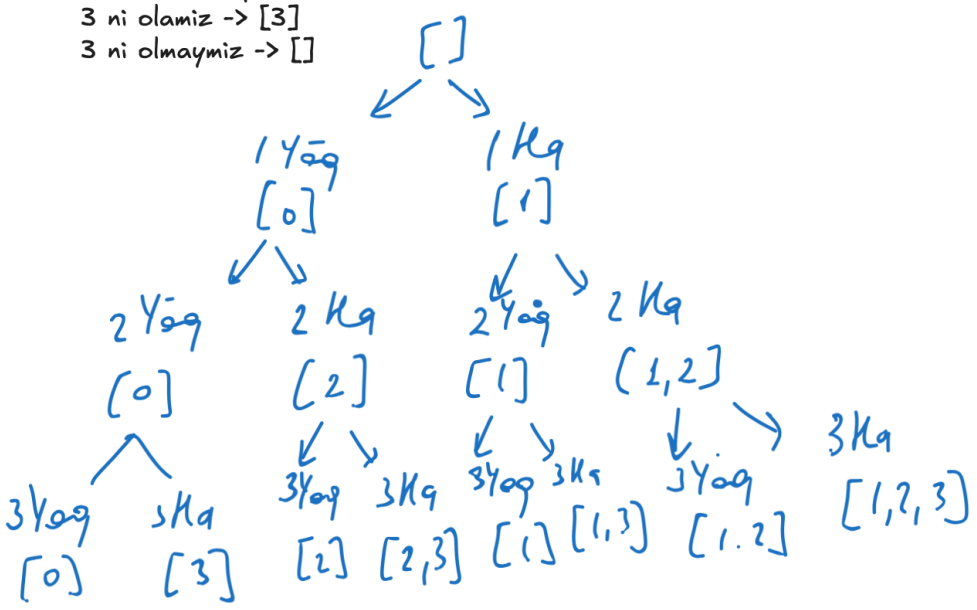
input=[1,2,3]
output=[[], [1], [2], [3], [1,2], [1,3], [2,3], [1,2,3]]

Demak bizda 3ta element bor, 1,2 va 3; 1 uchun alohida 2ta qaror. 2 uchun alohida 2 ta qaror, 3 uchun ham 2ta alohida qaror ko'riladi.

#1 uchun 2ta qaror:
1 ni olmaymiz -> []
1 ni olamiz -> [1]

#2 uchun 2ta qaror:
2ni olmaymiz -> []
2ni olamiz -> [2]

3 uchun 2ta qaror
3 ni olamiz -> [3]
3 ni olmaymiz -> []



Jami leaflar 8ta

Shu orqali biz current da vaqtinchalik subsetlarni saqlab ularni keyin yig'ib olamiz: bu jarayon **Backtracking** deb nomlanadi.

```
def subsets(nums: list[int]) -> list[list[int]]:
    result = [] # oxirgi subset resultni yig'ish uchun b
o'sh list

    def backtrack(index: int, path: list[int]):
    # recursive function yaratib, har bir index va path
orqali qarorlarni  qaysi element bo'yicha qilayotganimiz
ni yig'ib boradi. index listdagi raqamlarni oladi, path
- taqqoslanayotgan elementlarni yig'adi.
        if index == len(nums): # bu recursiondagi Base c
```

```
ase. qachonki index listdagi eng katta indexga yetganda
to'xtaydi barchasini chiqaradi.
            result.append(path)
            return
        # 1-Qaror: Hozirgi elementni OLMASLIK (Exclude d
ecision)
        backtrack(index + 1, path) # har bir indexni bir
ma bir o'tib chiqadi +1 bilan va faqat exclude bo'yicha
ishlaydi

        # 2-Qaror: Hozirgi elementni OLISH (Include deci
sion)
        backtrack(index + 1, path + [nums[index]]) # ele
mentlarni subsetga qo'shib boradi. path+nums[index] bund
a o'ziing alohida listi bilan ishlaydi va eski pathni
o'zgartirmaydi

    backtrack(0, []) #bu recursionni boshlash (ya'ni bos
hlang'ich holat:    index=0, path[] bo'sh).
    return result      #Barchasi tugagandan keyin subset
lar yig'ilgan bo'ladi
```

Factorial Time O(n!)

- Demak Factorial time - har bir keyingi pozitsiyada qolgan barcha elementlardan birini tanlash kerak bo‘lganda paydo bo‘ladi. Tanlovlar soni `n, n-1, n-2, ..., 1` tarzida kamayadi va jami kombinatsiyalar `n!` bo‘ladi. U nomi bilan factorial orqali hisoblanadi:

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1$$

Dasturlashda u ko'pincha elementlarning **barcha mumkin bo'lgan o'rin almashtirishlarini (Permutations)** yoki barcha tartiblarini birma-bir hosil qilib ko mezon ko'rib chiqishda paydo bo'ladi:

- $n = 1 \implies 1! = 1$ operatsiya
- $n = 3 \implies 3! = 3 \times 2 \times 1 = 6$ operatsiya
- $n = 5 \implies 5! = 120$ operatsiya
- $n = 10 \implies 10! = 3,628,800$ operatsiya
- $n = 20 \implies 20! \approx 2.43 \times 10^{18}$ (kompyuter yillab hisoblaydi!)

- Qanday holatlarda foydalaniladi:
  - Asosan Permutations (o’rin almashtirishlar) ya’ni n ta buyumni bir qatorga necha xil tartibda joylashtirish kabi masalalar. Yoki Brute Force attack (Parollarni almashtirish) belgilar o’rnini takrorlamasdan kabi.
  - Practice uchun masala ko’rsak: **LeetCode #172: Factorial Trailing Zeroes (**https://leetcode.com/problems/factorial-trailing-zeroes/)

Sizga bitta butun son n berilgan. $n!$ (n faktorial) natijasining **oxirida nechta nol (zero)** borligini aniqlang.
Eslatma: Funksiyangiz $O(\log n)$ vaqt murakkabligida ishlashi kerak.

Misollar:

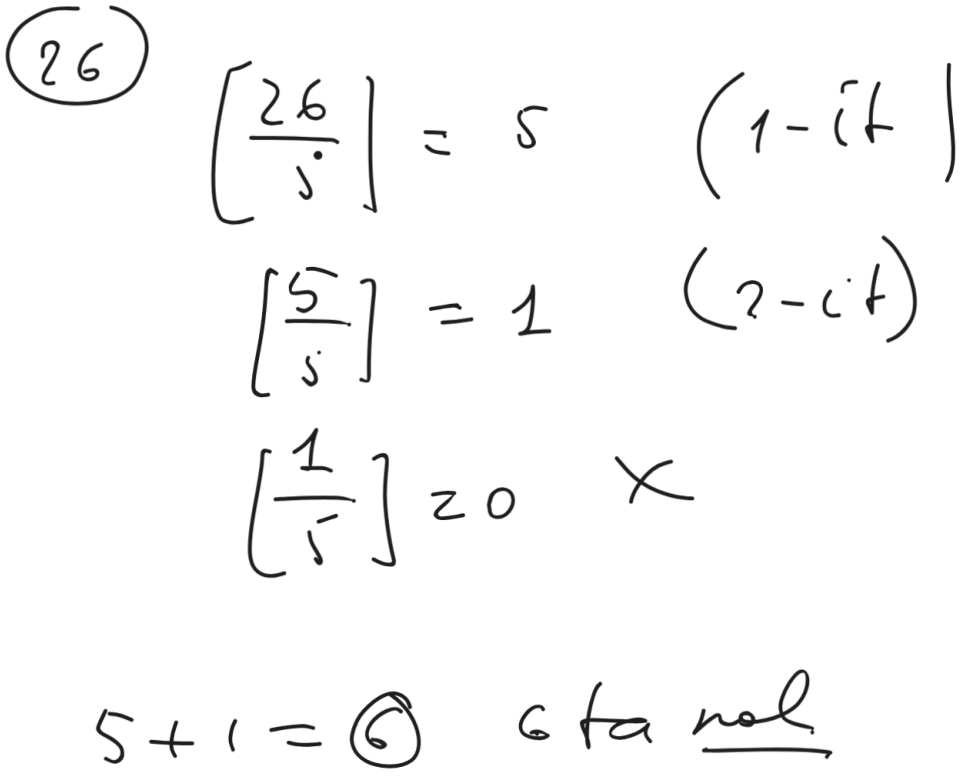
- **Input:** $n = 3 \to 3! = 6 \to$ **Output:** `0` (oxirida nol yo'q)
- **Input:** $n = 5 \to 5! = 120 \to$ **Output:** `1` (oxirida 1 ta nol bor)
- **Input:** $n = 10 \to 10! = 3628800 \to$ **Output:** `2` (oxirida 2 ta nol bor)

  - Yechim sifatida agar biz shunchaki birinchi factorial ni hisoblab, keyin undagi nechta nol borligini aniqlash orqali ishlashimiz mumkin. lekin masalani asosiy shartida trailing zeroes, nechta nol bor deb so’ragan. Kichik sonlarda faktorial hisoblash muammosiz bo’lishi mumkin lekin katta sonlarda overflow bo’lishi mumkin. shuning uchun boshqacha tomondan qarasak bo’ladi:
    Masalan: ko’paytmada qachon nollik son payd bo’ladi:

- 2ta holatda: 1-2ta son ko’paytmasi 0 bilan tugaydigan son chiqqanda; va 2-si: 10ga ko’paytirganda. Endi: xulosa faktorial tarkibida odatda 2, 5 sonlari ko’p uchraydi 10 hosil qilish uchun. Shunda tahlil qilsak:
- n=5 bo’lsa: 1×2×3×4×5 → 1ta 5 bor demak: 1 ta nol lik son bor
- n=10 bo’lsa: 5va10 (2×5) → 2ta 5 bor, demak: 2ta nol lik son bor
- Xulosa formulasi

$$\text{Nollar soni} = \lfloor \frac{n}{5} \rfloor + \lfloor \frac{n}{25} \rfloor + \lfloor \frac{n}{125} \rfloor + \dots$$

Demak biz n inputni 5ga bo’lib borsak va butun bo’linmalar sonini chiqarishimiz, 5ga karrali sonlarni beradi, nechta 5ga karrali sonlar bo’lsa shuncha 0 qatnashishini anglatadi deb xulosa qilsak bo’ladi




```python
def trailingZeroes(n: int) -> int:
    count = 0          # 1-qator: nollar sonini hisob
lagich
```

```
    while n > 0:        # 2-qator: n hali 0 ga teng b
o'lgunicha
        n //= 5         # 3-qator: n ni 5 ga bo'lamiz
        count += n      # 4-qator: chiqqan bo'linmani
count'ga qo'shamiz

    return count        # 5-qator: natijani qaytaramiz
```

Xulosa o’rnida Complexitylarni farqlarini ko’radigan bo’lsak:

- **O(1). Constant time** - da input size qancha katta bo’lishidan qat’iy nazar, bajaradigan operation count o’zgarmaydi
- **O(log n). Logarithmic time -** da har qadamda search katta qismga filterlab boriladi (Binary Search).
- **O(√n). Square root time -** da algoritm n gacha emas, faqat √n chegarasigacha yuradi.
- **O(n^2). Quadratic time -** da har bir element boshqa elementlar bilan solishtirish yoki juftliklar hosil qilish muammolarida ishlatiladi
- **O(2^n). Exponential time -** Har bir element uchun odatda 2ta decision mavjud bo’ladi va kombinatsiyalar tekshiriladi
- **O(n!). Factorial time -** da barcha mumkin bo’lgan tartiblar (permutationlar) tekshiriladi yoki hosil qilinadi.