1. Проблема вибору алгориму
2. Асимптотична складність алгоритму
3. Класи складності
4. Деякі підходи до розв'язку задач:
    1. Перебір та його оптимізація
    2. Перебір із поверненням
    3. Евристичні розв'язки

Для розв’язання більшості задач можна застосувати не один, а кілька різних алгоритмів. Якщо ми маємо справу з невеликими об’ємами даних, оптимізація не є суттєвою, так як на малих розмірностях задач різниця в часі виконання різних алгоритмів буде непомітною. Для великих об’ємів даних використання неефективного алгоритму може призвести до надто довгого виконання програми.

Наприклад, той же підбір паролю користувача. Задача може бути розв’язана перебором всіх можливих паролів, але час роботи повного перебору для достатньо довгих паролів на одній машині може становити роки -- отримання відповіді в такий термін є неприйнятним.

Для порівняння ефективності алгоритмів між собою вводиться поняття **обчислювальної складності алгоритму** -- відношення кількості операцій, які необхідно здійснити для розв’язку задачі, до розмірності задачі (найчастіше вона визначається кількістю вхідних даних).

Обчислити точну складність в більшості випадків неможливо, так як його програмна реалізація міститиме готові функції та операції, які відрізняються в різних мовах програмування. Ті ж звернення до елементів списку або полів об’єктів в різних мовах реалізовані по-різному, вимагають різної кількості машинних операцій і, відповідно, можуть працювати довше або швидше. Крім того, із ростом розмірності задачі вплив більшості складових програми на її час виконання значно зменшується.

Тому на практиці розглядається **асимптотична складність**, яка позначає рівень зростання обсягу роботи алгоритму з ростом об'єму вхідних даних. Фактично, замість точної функції, розглядається лише загальна тенденція її росту: складність цього алгоритму зростає лінійно, складність іншого -- як квадратична функція (y = n**2) і т.д.

Крім того, деякі алгоритми є досить складними і по-різному поводять себе в залежності від характеру вхідних даних. Тому розрізняють верхню межу кількості операцій, нижню межу та середнє значення. **Верхня межа** -- це найгірший випадок, який може бути для даного алгоритму (у багатьох алгоритмів є слабкі місця і можуть існувати набори даних, які оброблятимуться повільніше ніж звичайно). **Нижня** -- це навпаки, найкращий випадок, який матиме місце для “зручних” даних, (випадково

чи спеціально) пристосованих для обробки даним алгоритмом. **Середнє значення** -- це та кількість операцій, яка потребуватиметься в більшості випадків для обробки деяких абстрактних даних, і зовсім не обов'язково є середнім арифметичним від двох попередніх показників.

Наприклад, абстрактний алгоритм сортування завершить роботу дуже швидко, якщо дані вже відсортовані і навпаки, потребує найбільшої кількості операцій, якщо дані відсортовані в зворотньому порядку, так як абсолютно всі дані необхідно буде переставити. Зрозуміло, що на практиці рідко доведеться сортувати вже відсортовані послідовності, тому є сенс орієнтуватися на середню кількість операцій, яка потребується для сортування більшості реальних даних. "Найгірший" випадок також є важливим, так як він може потребувати значного часу роботи програми, що буде помітним для користувача і, ймовірно, погіршить його враження від роботи з програмою.

Умовно алгоритми можуть бути класифіковані за своєю асимптотичною складністю, яка в літературі часто позначається як O(g(n)). Запис f(n) є O(g(n)) означає, що складність алгоритму (яка є функцією від n, тобто залежить від n) при збільшенні n зростає не швидше ніж функція g(n), помножена на якесь число. Тобто із збільшенням розмірності задачі кількість операцій зростає приблизно із тією ж швидкістю, що функція g(n).

Константна складність f(n) є O(k) -- притаманна алгоритмам, для яких кількість операцій є постійною величиною, тобто не залежить від об'єму вхідних даних. Звичайно, це значить, що вони не передбачають обробки всіх даних і виконують лише якісь прості операції. Наприклад, функція, яка повертає перший елемент послідовності, помножений на 2:

```
def get_max(seq):
    return seq[0] * 2
```

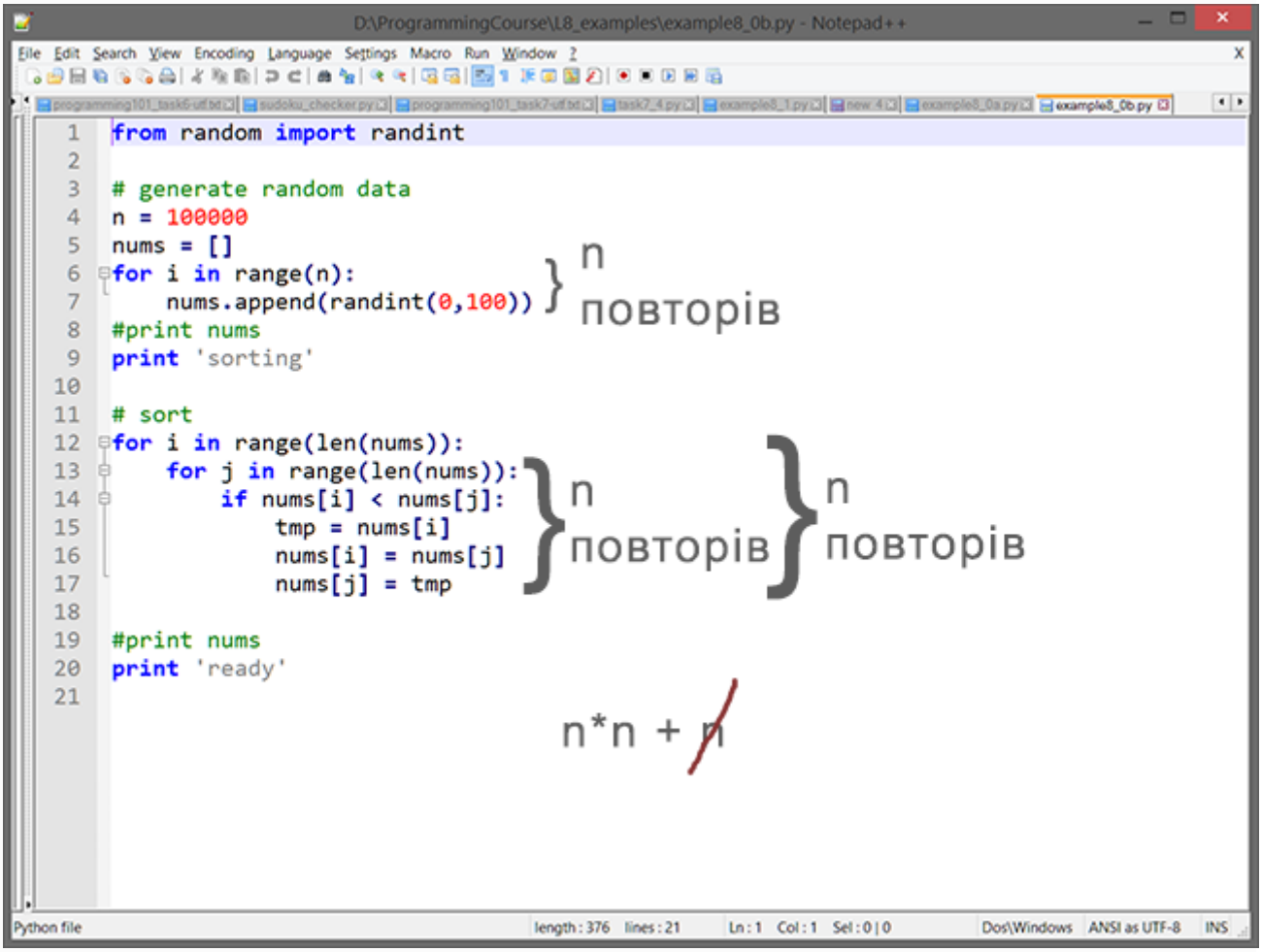
Нас не цікавить вся послідовність, ми не збираємося обробляти всі її елементи і кількість операцій для будь-якої довжини послідовності в такому випадку буде однаковою.

Логарифмічна складність f(n) є O(logkn) в чистому вигляді виникає рідко. На практиці це здебільшого рекурсивні задачі, які дозволяють розділяти послідовність на частини, зменшуючи таким чином розмірність задачі. Як правило, таке розбиття виникає внаслідок деякої оптимізації і розв'язок потребує ще інших дій, крім власне розбиття, тому виникають алгоритми із складністю n*logkn, n2*logkn та інші. Найбільш типовими представниками є бінарний пошук да "швидке" сортування, докладно розглянуті у відео-лекції.

Лінійна складність f(n) є O(n) виникає в простих задачах, для розв'язку яких достатньо пройти послідовність 1 або декілька разів. Наприклад, знайти найбільший елемент в послідовності, або знайти суму всіх елементів, або практичне завдання 4.1 цього курсу про перевірку паліндрома. Навіть, якщо ви використовуєте готові функції мови

програмування і з вашої точки зору, як користувача, це одна операція, всередині функції все одно знаходиться повний прохід послідовності, який так чи інакше вимагатиме близько n операцій.

Поліноміальна складність f(n) ϵ O(nk) відповідає великому класу алгоритмів, всередині якого розрізняють підкласи залежно від значення k: O(n**2), O(n**3), O(n**5), O(n**10) та інші. Такі алгоритми легко розрізнити, так як число k відповідає найбільшій вкладеності циклів у програмі.



На практиці це вже досить неприємний випадок, так як із збільшенням об’єму даних навіть для n2 кількість операцій зростає неймовірно швидко і для великих об’ємів даних (n≈10000 і більше) час роботи програми вже помітний неозброєним оком.

Ще гіршою є **експоненційна складність** f(n) ϵ O(kn). Це так само можна представити у вигляді циклів -- лише уявіть, що з розмірністю задачі росте не кількість ітерацій, а кількість циклів. Для того, щоб час роботи такої програми вийшов за межі допустимого навіть не потрібні великі розмірності задачі.

Відповідно, якщо ви зустрічаєтесь із алгоритмом, що має поліноміальну або експоненційну складність, пам’ятайте, що його ефективність є відносно низькою і це може викликати проблеми при подальшій роботі програми з реальними даними. В таких випадках будь-які додаткові відомості про

характер задачі та особливості даних, які доведеться обробляти, можуть бути використані для оптимізації програми і, таким чином, зменшення часу її роботи.

Повний перебір може застосовуватися для розв'язку задач, в яких можна сформулювати точний критерій правильності відповіді. Далі для пошуку правильної відповіді якимось чином (наприклад, за допомогою циклів) послідовно генеруються всі можливі варіанти і перевіряються за допомогою визначеного критерію.

Перебір часто є найбільш простим в реалізації, може застосовуватися без глибокого розуміння задачі і взагалі існують задачі, для яких немає відомих точних розв'язків крім застосування перебору. З іншого боку, це завжди алгоритми поліноміальної або експоненціальної складності, а отже найповільніші з усіх можливих і не можуть гарантувати отримання результату за адекватний час.

Наприклад, наступна задача: нехай задана послідовність натуральних чисел від 1 до n і ціле число m. Необхідно розставити знаки між числами послідовності (не змінюючи їх порядку) таким чином, щоб результатом отриманого арифметичного виразу було m.

Звичайно, можна перебрати всі можливі комбінації знаків, але, представляючи задачу "в лоб", для цього нам знадобиться n циклів, а значення n не зафіксоване. Цей один із випадків, коли для полегшення сприйняття коду замість циклів зручніше використати рекурсію. Кожному рівню заглиблення рекурсії відповідатиме позиція між деякими двома членами послідовності і в рекурсивній функції ми робитимемо вибір між двома варіантами знака, який можна там поставити: "+" чи "-". В напрямі "туди" передаватимемо номер наступної позиції і суму чисел, яку вже набрано в поточній розстановці знаків, "назад" повертатимемо згенеровану частину математичного виразу, якщо в поточній в кінці рекурсії отримано значення m, і False, якщо цей напрям не досяг необхідного результату.

```
def signs1(m, n):
    def calc_step(level, current):
        if level > n:
            return current == m
        res = calc_step(level + 1, current + level)
        if res:
            return '+' + str(level) + str(res)
        res = calc_step(level + 1, current - level)
        if res:
            return '-' + str(level) + str(res)

    res = calc_step(2, 1)
    if res:
        return '1' + res[:-4]
    return False
```

Мій варіант рекурсії виглядає приблизно так. Для n=20, m=2 розв'язок знаходиться швидко, для n=30, m=3 -- досить довго. При цьому рекурсія фактично завершує роботу, знайшовши хоч одну відповідь. Але у випадку, коли відповіді немає, відбувається повний перебір всіх варіантів, що

потребує ще більше часу. Такий “безрезультатний перебір” для n=30 потребував 6.5 хвилин.

```
print signs1(3, 5)  # 1-2+3-4+5
print signs1(2, 20) # 1+2+3+4+5+6+7+8+9+10+11+12+13-14+15-16-17-18-19-20
print signs1(2, 30) # False
print signs1(3, 30) # 1+2+3+4+5+6+7+8+9+10+11+12+ 13+14+15+16+17+18+19+20-21-22-
23+24-25-26-27-28-29-30
```

Найкращим способом оптимізувати перебір є організація **перебору (або пошуку) з поверненням**.

Майже завжди перебирається значення не однієї, а декількох змінних, отже розв’язок генерується не одномоментно, а по частинах: виставляється значення одного параметру, це значення фіксується і перебираються (за допомогою рекурсії або вкладених циклів) варіанти наступної змінної, на кожному з варіантів її значення фіксується і перебираються наступні змінні і т.д. І якщо на будь-якому з проміжних етапів ми можемо передбачити, що всі наступні “вкладені” варіанти будуть програшними (а це можливо для більшості задач), така перевірка і своєчасне припинення генерації беззмістовних варіантів суттєво пришвидшить роботу. Це і є “повернення” -- замість занурення у вкладені перебори ми додаємо якусь часткову умову і, якщо вона не справджується, одразу відсікаємо цілу групу розв’язків, повертаючись на крок назад і випробовуючи групу розв’язків, що базуються на іншому, більш перспективному значенні.

Отже, нам необхідна додаткова перевірка, яка б дозволила відсікти частину варіантів. Спочатку я пошкодував про невдалий вибір прикладу, але ні -- можна перевірити наступний випадок: якщо на поточному кроці ми набрали надто малий результат і суми всіх наступних чисел не вистачить, щоб дотягнути його до m, і навпаки -- надто великий результат, такий, що, віднявши всі наступні числа, все одно отримаємо більше за m. На щастя, додати всі наступні елементи послідовності нескладно, так як вони являють собою арифметичну прогресію і для визначення їх суми навіть цикл не потрібний. В результаті вийшло наступне:

```
def signs2(m, n):
    def calc_step(level, current):
        if level > n:
            return current == m
        sum_of_next = 1.0 * (level + n) / 2 * (n - level + 1)
        if m > current + sum_of_next or m < current - sum_of_next :
            return False
        res = calc_step(level + 1, current + level)
        if res:
            return '+' + str(level) + str(res)
        res = calc_step(level + 1, current - level)
        if res:
            return '-' + str(level) + str(res)

    res = calc_step(2, 1)
    if res:
        return '1' + res[:-4]
    return False
```

Відповіді повертаються такі ж, як і в попередній версії функції. “Порожній” перебір по всіх варіантах signs2(2, 30) завершився майже на хвилину швидше. Це приємно, але не достатньо.

Проблема такої проміжної умови полягає в тому, що останніми ми додаємо чи віднімаємо найбільші числа, які найбільше впливають на кінцеву суму, і, таким чином, “повернення” в нашому переборі завжди відбуватиметься десь на останніх етапах. Підвищити ефективність повернення можна дуже просто: складати числа, починаючи не з початку, а з кінця -- тоді невдало складені великі числа дуже швидко приведуть нас у ситуацію, коли переглядати наступні елементи вже не має сенсу. Так і зробимо:

```
def signs3(m, n):
    def calc_step(level, current):
        if level < 1:
            return current == m
        sum_of_next = 1.0 * (level + 1) / 2 * (level)
        if m > current + sum_of_next or m < current - sum_of_next :
            return False
        res = calc_step(level - 1, current + level)
        if res:
            return '+' + str(level) + str(res)
        res = calc_step(level - 1, current - level)
        if res:
            return '-' + str(level) + str(res)

    res = calc_step(n-1, n)
    if res:
        return str(n) + res[:-4]
    return False
```

Звичайно, порядок виведення доданків змінився. За необхідності рядок-відповідь можна було б перевернути назад або формувати його дещо по-іншому, але для нашого прикладу це не принципово. Виграш у часі на “порожньому” переборі виявився суттєвим. Виклик signs3(2, 30) завершив свою роботу замість кількох хвилин вже за 20 секунд.

Для тих, кому цікаво, коли на кожному етапі рекурсії перебирається небагато значень, замість неї для генерації відповідей можна генерувати числа у альтернативних системах числення. Наприклад, в нашому випадку на кожному етапі робиться вибір з 2 значень “+” та “-” -- отже для імітації вибору нам знадобиться двійкова система числення. Глибина рекурсії n-1 -- отже необхідно згенерувати всі двійкові числа довжиною n-1; в кожному числі 1 нехай відповідатиме “+”, а 0 -- “-”. Сам генератор нескладний: починаємо з нуля і додаємо одиницю на кожному кроці, поки не досягнемо 11...111, в процесі отримаємо всі можливі комбінації 0 та 1, які відповідатимуть всім можливим комбінаціям знаків:

```
def signs4(m, n):
    def add_1_to_binary_number(s):
        digit = n-2
        while s[digit] != '0':
            if s[digit] == '1':
                s = s[:digit] + '0' + s[digit+1:]
            digit -= 1
        s = s[:digit] + '1' + s[digit+1:]
```

```
        return s

    s = '0' * (n-1)
    s_final = '1' * (n-1)
    while s != s_final:
        # generate next position
        s = add_1_to_binary_number(s)
        # calculate sum
        res = n
        res_str = str(n)
        for j in range(0, n-1):
            sequence_element = n-1-j
            string_element = int(s[j])
            res = res + sequence_element * int(s[string_element]) -
sequence_element * (1-string_element)
            res_str += '+'*string_element + '-'*(1-string_element) +
str(sequence_element)
        if res == m:
            return res_str
    return False
```

Правда, реалізувати “повернення” в такому переборі буде набагато складніше, так як комбінації, що перебираються, гірше структуровані.

А ось варіант рекурсивного розв’язку перебором із поверненням задачі з ферзями (для довільного розміру шахової дошки):

```
# 1.5sec, 22740 combinations
counter = 0
n = 8
print 'v7, recursion bactracking'

def output_desk(p):
    if p == None:
        return
    for i in range(n):
        row = ''
        for j in range(n):
            flag = False
            for ii in range(n):
                if p[ii][0] == i and p[ii][1] == j:
                    flag = True
            row += '1'*int(flag) + '0'*(1-int(flag))
        print row

def is_ok(p):
    global counter
    counter += 1
    for i in range(n):
        for j in range(n):
            if i != j and \
                (p[i][0] == p[j][0] or p[i][1] == p[j][1] \
                or p[i][0]+p[i][1] == p[j][0]+p[j][1] \
                or n-p[i][0]+p[i][1] == n-p[j][0]+p[j][1]):
                return False
    return True

def find_position():
    def try_queen(pos):
        if len(pos) == n:
```

```
            if is_ok(pos):
                return pos
            else:
                return False
        for i in range(len(pos)):
            for j in range(i+1, len(pos)):
                if pos[i][1] == pos[j][1]:
                    return False
        for current_queen_pos in range(0,n):
            pos_new = pos[:]
            pos_new.append((len(pos), current_queen_pos))
            res = try_queen(pos_new)
            if res:
                return res

    start_position = []
    return try_queen(start_position)

p = find_position()
print counter
print p
output_desk(p)
```

Евристичний алгоритм (евристика) -- це алгоритм розв'язку задачі, який не має точного обґрунтування, але в більшості практичних випадків забезпечує прийнятну відповідь (за якістю та часом роботи).

Це не значить, що рішення, отримане евристичним шляхом завжди є приблизним. Під неточністю евристики розуміють, що навіть, якщо знайдені таким чином відповіді є точними, немає гарантій, що вони так само будуть точними для будь-яких інших вхідних даних.

Евристичні методи передбачають застосування творчого підходу та досвіду для пошуку рішення. В загальному випадку вони застосовуються не лише в програмуванні, але для вирішення будь-яких задач. Наприклад, проектування складних систем, розв’язування логічних головоломок, будь-яка творча діяльність.

В програмуванні це реалізується в спробі відтворити дії людини з розв’язку задачі і формалізувати їх у вигляді алгоритму. Якщо розглядати задачу як лабіринт (розповсюджена метафора з психології), то пошук рішення є пошуком шляху в цьому лабіринті, а евристика -- це набір прийомів, вироблених вами, які дозволяють відкинути частину шляхів, зменшуючи таким чином простір можливих рішень.

Таким чином, будь-які ваші алгоритми, розроблені на інтуїтивному рівні, та будь-які оптимізації алгоритмів, які впроваджуються для збільшення швидкості їх роботи, можна вважати евристичними.