ЗАВДАННЯ XXIV МІЖНАРОДНОЇ ОЛІМПІАДИ З ІНФОРМАТИКИ ТА РЕКОМЕНДАЦІЇ ЩОДО ЇХ РОЗВ'ЯЗАННЯ

Гуржій А.М., Бондаренко В.В.

ЗАВДАННЯ ПЕРШОГО ТУРУ

1. ОДОМЕТР З КАМЕНЯМИ

Леонардо винайшов оригіналвнвій одометр, що є возиком, який може вимірювати відстанв шляхом кидання каменів у тих місцях, де він повернувся. Підрахувавши кількість каменів, ми отримаємо кількість поворотів коліс возика, яка дозволить користувачу вирахувати відстань, яку проїхав одометр. Як справжні програмісти, ми додали одометру програмний контроль, розширивни його функціональність. Потрібно написати програму, що відповідає заданим нижче правилам.

Робоча поверхня. Одометр переміщується по уявній квадратній сітці розміром 256×256 клітинок. У кожній клітині не може знаходитись більше ніж 15 каменів. Кожна клітина задається парою координат (рядок, стовпчик), де кожна координата належить діапазону від 0 до 255 включно. Для клітини з координатами (i, j) сусідніми є клітини з координатами $(i-1, j)$, $(i+1, j)$, $(i, j-1)$ і $(i, j+1)$, якщо вони існують. Клітини, що розташовані у першому або останньому рядку або стовпчику, називаються границею. На початку одометр завжди знаходится у клітині з координатами (0, 0) (північно-західний кут) і орієнтований на північ.

Основні команди. Одометр можна запрограмувати за допомогою таких команд:

- **left** — повернутися на 90 градусів ліворуч (проти годиннокової стрілки) і залишитись у поточній клітині; наприклад, якщо одометр було орієнтовано на південь, то після виконання цієї команди його буде орієнтовано на схід;
- **right** — повернутися на 90 градусів праворуч (за годинниковою стрілкою) і залишитись у поточній клітині; наприклад, якщо одометр було орієнтовано на захід, то після виконання цієї команди його буде орієнтовано на північ;
- **move** — пересунутись вперед, тобто у тому напрямку, у якому орієнтовано одометр, у сусідню клітину. Якщо такої клітини немає (наприклад, вже досягнуто границі у напрямку орієнтації одометра), то команда нічого не робить;
- **get** — видалити один камень із поточної клітинки. Якщо у поточній клітинці каменів немає, то команда нічого не робить;
- **put** — добавити один камінь у поточну клітинку. Якщо у поточнй клітинці вже є 15 каменів, то команда нічого не робить. У одометра ніколи не закінчуються камені;
- **halt** — завершити виконання програми.

Одометр виконує команди у тому порядку, у якому їх задано в програмі. Програма повинна містити не більше однієї команди у рядку. Порожні рядки ігноруються. Символ # позначає початок коментаря; довільний текст після нього до кінця рядка ігноруються. Якщо одометр доходить до кінця програми, виконання програми завершується.

Приклад 1. Розглянемо таку програму для одометра. Після її виконання одометр опиняється орієнтованим на схід у клітинці (0, 2). Зверніть увагу, що першу команду move буде проігноровано, так як одометр знаходится у північно-західному куті й орієнтований на північ.

```
move # нічого не робить
right
# тепер одометр орієнтовано на схід
move
move
```

Мітки, границі і камені. Для того, щоб змінювати порядок виконання команд залежно від поточного стану, можна використовувати мітки. Мітка — це регістронезалежний рядок, що складається з не більше ніж з 128 таких символів: $a,\ldots, z, A,\ldots, Z$, 0,…, 9. Нові команди для роботи з мітками перелічено нижче. У всіх описах L означає довільну коректну мітку.

- **L:** (тобто L, за яким іде двокрапка ':') — визначає розташування мітки L у програмі. Усі оглошення мітки мають бути унікальними. Оголошення мітки не впливає на одометр;
- **jump L** — продовжити виконання, безумовно перейшовши на рядок з міткою L;
- **border L** — продовжити виконання шляхом переходу на рядок з міткою L, якщо одометр знаходиться на границі, причому він орієнтований у бік границі поля, тобто інструкція move нічого не зробить. У протилежному випадку виконання програми продовжується у звичайному порядку і ця команда нічого не робить;
- **pebble L** — продовжити виконання шляхом переходу на рядок з міткою L, якщо у поточній клітині є хоча б один камінь. У протилежному випадку виконання програми продовжується у звичайному порядку і ця команда нічого не робить.

Приклад 2. У результаті виконання такої програми одометр визначає розташування першого (самого західного) каменя у рядку 0 і зупиняється там. Якщо в рядку 0 немає каменів, то одометр зупиняється на границі в кінці рядка. У цій програмі використовуються дві мітки: leonardo і davinci.

```
right
leonardo:
pebble davinci # знайдено камінь
border davinci # кінець рядка
```

move
jump leonardo
davinci:
halt

Спочатку одометр повертається праворуч. Цикл починається з оголошення мітки leonardo: і закінчується командою jump leonardo. У цьому циклі одометр перевіряє присутність каменя у поточній клітинці або те, що він досягнув границі. Якщо це не так, то одометр виконує команду move і пересувається з клітини (0, *j*) у клітину (0, *j*+1). Тут можна обійтись без команди halt, так як все одно програма закінчить виконання, коли дійде до останнього рядка.

Завдання. Напишіть і відправте програму на мові одометра, який описано вище. У кожній підзадачі, які визначено нижче, потрібно реалізувати певну поведінку одометра, водночас мають бути дотримані два таких обмеження.

- ***Розмір програми*** — програма має бути достатньо короткою. Розмір програми — це кількість команд у ній. Оголошення міток, коментарі і пусті рядки не враховуються під час обчислення розміру програми.
- ***Кількість операцій*** — ця програма має бути достатньо швид-кою. Кількість операцій — це кількість виконаних кроків: кожна окрема команда рахується як один крок, не залежно від того, чи мала ефект ця команда, чи ні. Оголошення міток, коментарі і пусті рядки не рахуються кроком.

У прикладі 1 розмір програми дорівнює 4 і кількість операцій дорівнює 4. У прикладі 2 розмір програми дорівнює 6. У випадку запуску програми з одним каменем у клітині (0, 10) кількість операцій буде дорівнювати 43: right, 10 ітерацій циклу, кожна ітерація займає 4 кроки (pebble davinci; border davinci; move; jump leonardo), і нарешті, виконуються команди pebble davinci і halt.

Підзадача 1 [9 балів]. На початку у клітинці (0, 0) знаходиться x каменів, і y каменів в клітині (0, 1), у той час як всі решта клітин пусті. Зверніть увагу, що у любій клітині може бути не більше 15 каменів. Потрібно написати програму, після виконання якої одометр будет знаходитися у клітинці (0, 0), якщо $x \le y$, і в клітинці (0, 1) — у протилежному випадку. Нас не хвилює напрямок орієнтації одометра; нас також не хвилює кількість каменів і їх розташування.

Обмеження: розмір програми ≤100, кількість операцій ≤1000.

Підзадача 2 [12 балів]. Така ж задача, яка описана вище, але після завершення програми клітинка (0, 0) має містити рівно x каменів і клітинка (0, 1) має містити рівно y каменів.

Обмеження: розмір програми ≤200, кількість операцій ≤2000.

Підзадача 3 [19 балів]. У рядку з індексом 0 знаходиться рівно два камені. Один — у клітинкі (*0, x*), інший — у клітинці (*0, y*), x і y відрізняютвся, $(x+y)$ — парне. Потрібно написати програму, у резулвтаті виконання якої одометр опинитвся у клітині (0, $(x+y)/2$), тобто рівно посередині між двома клітинами, які містили камені. Кінцеве розташування коменів не має значення.

Обмеження: розмір програми ≤100, кількість операцій ≤200000.

Підзадача 4 [до 32 балів]. На дошці знаходиться не більше 15 каменів, ніякі два з них не знаходяться в одній клітинці. Потрібно написати програму, яка збирає всі камені у північно-західному куті. якщо бути більш точним, то, якщо спочатку на дошці було x каменів, то після завершення програми в клітині (0, 0) має знаходитись рівно x каменів, а в решті клітин каменів має не залишитись.

Кількість балів за цю підзадачу залежить від кількості операцій, що виконала і відіслала на перевірку програма. Якщо L — це максимальна кількість операцій для різних вхідних даних, що відповідають цій підгрупі, то ви отримаєте таку кількість балів:

- 32 бали, якщо $L \le 200000$;
- $32-32\ \log_{10}(L/200000)$ балів, якщо $200000 < L < 2000000$;
- 0 балів, якщо $L \ge 2000000$.

Обмеженя: розмір програми ≤200.

Підзадача 5 [до 28 балів]. У кожній клітинці може бути довільна кількість каменів (зрозуміло, від 0 до 15). Потрібно написати програму, яка знаходить мінімум, тобто після її завершення одометр знаходиться в такій клітинці *(i, j)*, що довільна інша клітинка містить не менше каменів, ніж клітинка *(i, j)*. Після виконання програми кількість каменів у кожній клітинці має бути такою ж, як і до запуску програми.

Кількість балів за цю підзадачу залежитв від розміру програми P.

- 28 балів, якщо $P \le 444$;
- $28-28\ \log_{10}(P/444)$ балів, якщо $444 < P < 4440$;
- 0 балів, якщо $P \ge 4440$.

Обмеження: кількість операцій ≤44400000.

Деталі реалізації. Надішліть для превірки один файл по кожній підзадачі, складений за правилами, що описані вище. Розмір кожного файлу не повинен перевищувати 5 мегабайт. Для кожної підзадачі вашу програму для одометра буде протестовано на декількох наборах вхідних даних, і ви отримаєте деяку інформацію про ресурси, що використала ваша програма. У тому випадку, якщо ваш код не є синтаксично коректним, ви отримаєте інформацію про вид синтаксичної помилки.

Рекомендації щодо розв'язання

Як ілюстрацію, наведемо можливий розв'язок підзадачі 5, у якому код використовується повторно. Зауважте, що знайти мінімум не важко, забираючи по одному камінцю з кожної з клітин. Однак, всі забрані камінці мають бути повернуті назад, що ускладнює розв'язок.

Підзадача 5. Генератор розв'язку на Python

```
#Для всіх можливих мінімальних значень
#(за винятком 15), шукаємо клітину,
#що містить стільки камінців. Застосовуємо
# деякі оптимізації для зменшення кількості
#інструкцій.
#Перший крок шукає нулі.
print "jump 0_scan_all"
for i in xrange (0,15):
#Ця секція намагається перейти на наступний
#рядок після одного проходу. Якщо вона досягає
#границі , можна шукати наступного кандидата
#на максимум.
print "%d_test_next_row : " % i
print " right "
print " border %d_scan_al l " % ( i+1)
print "move"
print " right "
print "%d_test_next_row_l1 : " % i
print " border %d_test_next_row_l1end" % i
print "move"
print "jump %d_test_next_row_l1" % i
print "%d_test_next_row_l1end : " % i
```

```
print " right "
# починаємо перевірку наступного рядка сітки.
print "%d_scan_all: " % i
print " right "
print "%d_test_scan_row : " % i
for j in xrange(i):
print "get"
print " pebble %d_test_scan_row_continue" % i
print "jump end_%d" % i
print "%d_test_scan_row_continue : " % i
for j in xrange(i):
print "put"
#Коли доходимо до границі, намагаємось
#перейти на наступний рядок та перейти
#назад у перший стовпчик.
print " border %d_test_next_row" % i
print "move"
print "jump %d_test_scan_row" %i
#Коли ми знаходимо мінімум, можемо повторно
#використати код, що розкладає камінці назад
#у клітини in the cell.
for i in xrange(14,0, –1):
print "end_%d:" % i
print "put"
print "end_0 : "
#Якщо всі клітини містять 15 камінців,
#підходить довільна позиція.
print " 15 _scan_all : "
```

2. ПАРАШУТНІ КІЛЬЦЯ

Ранню і достатньо складну версію того, що ми сьогодні називаємо парашутом, описано у работі Леонардо Codex Atlanticus (ca. 1485). Парашут Леонардо складався із тканини, що була натягнута на дерев'яний каркас у формі піраміди.

Зв'язані кільця. Парашутист Адріан Ніколас випробував проект Леонардо через 500 років. Для цього сучасна полегшена конструкція з'єднала парашут Леонардо з тілом людини. Ми хочемо використовувати з'єднанні кільця, у яких ще передбачені крюки для закріплення кілець до тканини одягу. Кожне кільце зроблене з гнучкого і міцного матеріалу. Кільця легко з'єднуються разом, так як кожне кільце може бути відкрите і знову закрите. Особлива конфігурація з'єднаних кілець — це *ланцюг*. Ланцюг — це послідовність одного або більше кілець, у якій кожне кільце з'єднане тільки з двома сусідніми кільцями, крім першого й останнього, які з'єднані тільки з одним іншим кільцем, як показано нижче. Зверніть увагу, що одне кільце — також є ланцюгом (рис. 1).

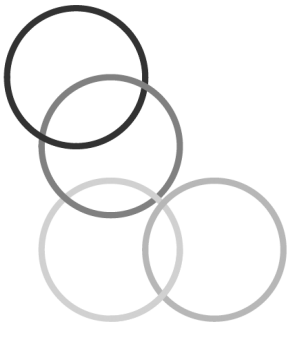

Рис. 1

Приклад. На рисунку 2 наведено 7 кілець, що пронумеровані від 0 до 6. Маємо 2 критичних кільця. Одне критичне кільце — це кільце номер 2, так як після його видалення решта кілець утворюють ланцюги [1], [0, 5, 3, 4] і [6]. Друге критичне кільце з номером 3, так як після його видалення решта кілець утворюють ланцюги [1, 2, 0, 5], [4] і [6]. Якщо ми видалемо довільне інше кільце, ми не отримаємо множину незв'язаних ланцюгів. Наприклад, після видалення кільця номер 5, хоча ми і маємо ланцюг [6], але зв'язані кільця 0, 1, 2, 3 і 4 ланцюг не утворюють.

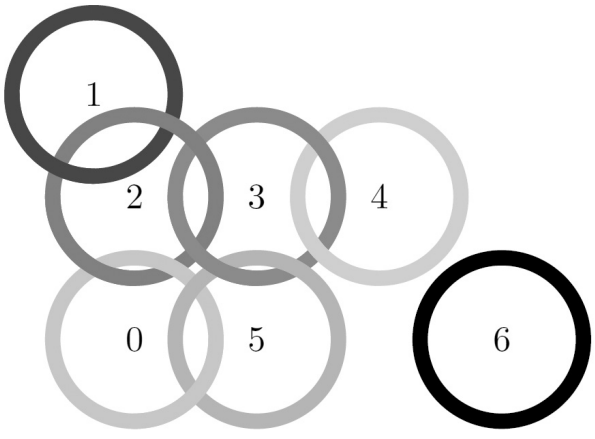

Рис. 2

Завдання. Ваша задача — обчислювати кількість критичних кілець у конфігурації, що отримано в результаті взаємодії з вашою програмою.

На початку є деяка кількість попарно не зв'язаних кілець. Потім кільця зв'язуються разом. У довільний момент у вас можуть запитати кількість критичних кілець у поточній конфігурації. А саме, ви маєте реалізувати три процедури:

- **Init(N)** — ця процедура викликається рівно один раз на початку, щоб повідомити, що у початковій конфігурації є N попарно не зв'язвних кілець, що пронумеровано від 0 до $N-1$ включно;
- **Link(A, B)** — зв'язуютвся два кільця з номерами A і B. Гарантується, що A і B є різними, і ще не зв'язані безпосереднво; крім того, немає додаткових умов для A і B, зокрема немає умов, що виникають з фізичних обмежень. Зрозуміло, що Link(A, B) і Link(B, A) є еквівалентними;
- **CountCritical()** — повертає кількість критичних кілець для поточної конфігурації зв'язаних кілець.

Приклад. Розглянемо наш рисунок з $N=7$ кілець і припустимо, що вони спочатку не зв'язані. Ми покажемо можливу послідовність виклику процедур, де після останнього виклику ми отримуємо ситуацію, зображену на рисунку 2.

Виклики	Повернуті значення
Init(7)	
CountCritical()	7
Link(I, 2)	
CountCritical()	7
Link(0, 5)	
CountCritical()	7
Link(2, 0)	
CountCriticalQ	7
Link(3, 2)	
CountCritical()	4
Link(3, 5)	
CountCritical()	3
Link(4, 3)	
CountCriticalQ	2

Підзадача 1 [20 балів]

- $N \le 5000$.
- Функція CountCritical викликається тільки один раз, після всіх інших викликів. Функція Link викликається не більше 5 000 разів.

Шдзадача 2 [17 балів]

- $N \le 1000000$.
- Функція CountCritical викликається тільки один раз, після всіх інших викликів. Функція Link викликається не більше 1 000 000 разів.

Підзадача 3 [18 балів]

- $N \le 20000$.
- Функція CountCritical викликається не більше 100 разів. Функція Link викликається не більше 10 000 разів.

Підзадача 4 [14 балів]

- $N \le 100000$.

- Функції CountCritical і Link викликаються в сумі не більше 100 000 разів.

Підзадача 5 [31 бал]

- $N \leq 1000000$.
- Функції CountCritical і Link викликаються в сумі не більше 1 000 000 разів.

Рекомендації щодо розв'язання

Зрозуміло, що утворена кільцями структура є графом, де кільця відповідають вершинам, а зв'язки між кільцями — ребрам. Неоптимальний, але достатній для розв'язання всіх підзадач, крім останньої, розв'язок базується на таких спостереженнях:

- якщо є вершина V ступеня ≥ 4, ніяка інша вершина не може бути критичною, оскільки її видалення все ще залишає V ступенем ≥ 3; отже якщо є більше однієї вершини ступеня ≥ 4, то критичних вершин немає;
- якщо є вершина V ступеня 3, кожна критична вершина ще V або один з її сусудів;
- якщо є цикл, то всі критичні вершини лежать на ньому;
- якщо граф є лінійним (тобто набір окремих шляхів), всі його вершини є критичними.

Алгоритм перевірки можна легко розширити на динамічний випадок з останньої підзадачі, єдиною нетривіальною задачею буде підтримання інформації про цикли, що можна робити за допомогою відповідних структур даних (система непересічних множин та інші).

3. РАКОПИСЕЦЬ

Деякі люди говорять, що Леонардо був великим шанувальником Іогана Гутенберга, німецького коваля, який винайшов рухливий (набірний) друк, і що він віддав належне, сконструювавши машину, названу ним *ракописець — U gambero scrivano* — дуже простий набірний пристрій. Він чимось схожий на сучасну просту друкарську машинку і всього 2 команди: одна, щоб надрукувати наступний символ, і друга, щоб відмінити декілька останніх команд. Чудовою властивістю ракописця є виключна потужність команди відміни, яка сама собою розглядається як команда і також може бути відміненою.

Завдання. Вам необхідно реалізувати програмну модель ракописця: вона починає роботу з пустого тексту, обробляє послідовність команд, що передаються їй користувачем, і запити відносно певних позицій у поточному стані тексту, як описано нижче.

- **Init()** — викликається один раз на початку виконання, без аргументів. Може використовуватись для ініціалізації структур даних. Ця операція ніколи не відміняється.
- **TypeLetter(L)** — додає в кінець тексту один символ L — маленьку літеру з діапазона $a,\ldots,z$.
- **UndoCommands(U)** — відміняє останні U команд, де U — додатнє ціле число.
- **GetLetter(P)** — повертає символ — літеру, що знаходиться в позиції P поточного тексту, де P — невід'ємне ціле число. Перша літера тексту має індекс 0. Цей запит не є командою і тому ігнорується командою відміни.

Після початкового виклику Init() інші процедури можуть виклика-тись нуль або більше разів у довільному порядку. Гарантується, що U не буде перевищувати кількість раніше отриманих команд і що P буде менше ніж поточна довжина тексту (кількість літер у поточному тексті).

Виклик UndoCommands(U) відміняє попередні U команд у зворотньому порядку. Якщо відміняється команда TypeLetter(L), то з кінця тексту видаляється літера L. Якщо відміняється команда UndoCommands(X), то для цього значення X вона заново застосовує попередні X команд у їх оригінальному порядку.

Приклад. Нижче наведено послідовність викликів разом зі станама тексту після кожного з викликів.

Виклик	Результат	Поточний текст
Init()		
TypeLetter(a)		a
TypeLetter(b)		ab
GetLetter(l)	b	ab
TypeLetter(d)		abd
UndoCommands(2)		a
UndoCommands (1)		abd
GetLetter(2)	d	abd
TypeLetter(e)		abde
UndoCommands (1)		abd
UndoCommands (5)		ab
TypeLetter(c)		abc
GetLetter(2)	c	abc
UndoCommands(2)		abd
GetLetter(2)	d	abd

Підзадача 1 [5 балів]

Загальна кількість команд і запитів знаходиться у діапазоні від 1 до 100 (включно) і немає викликів UndoCommands.

Підзадача 2 [7 балів]

Загальна кількість команд і запитів знаходиться у діапазоні від 1 до 100 (включно) і немає відмін команд UndoCommands.

Підзадача 3 [22 балів]

Загальна кількість команд і запитів знаходиться у діапазоні від 1 до 5 000 (включно).

Підзадача 4 [26 балів]

Загальна кількість команд і запитів знаходиться у діапазоні від 1 до 1 000 000 (включно). Всі виклики GetLetter відбуваються після всіх викликів TypeLetter і UndoCommands.

Підзадача 5 [40 балів]

Загальна кількість команд і запитів знаходиться у діапазоні від 1 до 1 000 000 (включно).

Рекомендації щодо розв'язання

Розумним способом отримати ефективний розв'язок буде використати префіксне дерево як структуру даних для збереження еволюції системи. Префіксне дерево буде містити всю історію до поточного моменту, а момент часу буде задаватись показчиком на вузол у дереві.

Обробка однієї команди у цьому випадку буде займати час $O(I)$:

- для друкування літери потрібно посунутись вниз по дереву (створивши новий вузол, якщо потрібно);
- для відміни K команд потрібно пересунутись на K станів назад.

У всіх підзадачах крім останньої, після обробки всіх команд, остаточний вміст рядка можна перенести з префіксного дерева до масиву і використовувати його для обробки запитів за час $O(I)$, що дає загальну оцінку $O(N)$ для часу та пам'яті.

Остання підзадача вимагає більш складного підходу для пошуку літер у тексті. Для цього необхідно мати можливість визначити k-го нащадка поточного вузла. Є багато структур даних, що забезпечать загальний час $O(N\log N)$. Наприклад, кожен вузол на рівні D може містити вказівник на свого 2^k-го нащадка, де k-є позицією самої правої 1 у двійковому представленні D.