Розв'язання задач першого туру

25 березня 2012 р.

1 Завдання першого туру

1.1 «Триоміно» (Ілля Порубльов).

1.1.1 Розв'язання

якщо таблиця заповнена правильно, але не фыгурками триомино, то такий тест зарахуэться?

З питань учасників

Розглянемо два різні правильні та ефективні способи, кожен з яких працює за час, пропорційний розміру вхідних даних. Спосіб (Б) очевидніший (більш стандартний), але обсяг коду виявляється меншим у способі (А).

Спосіб (А) Легко переконатися, що масив відповідає правильному покриттю фігурками триоміно тоді й тільки тоді, коли виконуються такі дві вимоги:

1. Значень «`0`» є рівно $(N * M) \bmod 3$ штук, а кожного зі значень від «`1`» до `(M*N)div 3` — рівно по три штуки.
2. Для кожного зі значень від «`1`» до `(M*N)div 3`, зайняті ним клітинки утворюють фігурку триоміно.

Твердження 1. *Три різні клітинки, «координати» (пари індексів) яких подані в парах* $(A.i, A.k)$, $(B.i, B.k)$ *та* $(C.i, C.k)$, *утворюють фігурку триоміно тоді й тільки тоді, коли* $\big(|A.i - B.i| + |A.k - B.k|\big) + \big(|A.i - C.i| + |A.k - C.k|\big) + \big(|B.i - C.i| + |B.k - C.k|\big) = 4$.

Доведення. Для кожної із двох можливих фігурок триоміно дійсно маємо $1 + 1 + 2 = 4$.

Оскільки A, B та C — різні клітинки, то кожна з узятих в дужки сум двох модулів (вони називаються *манхетенськими відстанями* між клітинками) ціла додатна. Число 4 можна розкласти в суму трьох цілих додатних лише як $1 + 1 + 2$ (або $1 + 2 + 1$, або $2 + 1 + 1$). З іншого боку, манхетенська відстань дорівнює 1 тоді й тільки тоді, коли клітинки мають спільну сторону. Зразу дві одиниці для пар (A, B), (A, C) та (B, C) означають, що якась із клітинок має спільні сторони з обома іншими (а ті дві між собою — не мають). А це і є всі можливі фігурки триоміно, й лише вони. □

Зрозуміло, перед застосуванням розглянутого критерію (для кожного зі значень від «`1`» до `(M*N)div 3`) треба переконатися, що виконується перша вимога (щодо кількостей), а також сформувати самі трійки «координат». Це можна зробити, наприклад, так. Створимо масиви `num : array[0..(200*200)div 3] of integer` та `C : array[1..(200*200)div 3] of array[1..3] of record i, j : integer end`. В елементах `num` зберігатимемо кількості

відповідних значень. Кожен елемент `C[v]` буде масивом, що міститиме перелік «координат» різних клітинок зі значенням `v`. Масиви `num` та `C` при бажанні можна формувати одразу при читанні вхідних даних (прочитали з файлу `v`, збільшили `num[v]`, якщо воно не перевищило 3, то дописали поточні «координати» до `C[v]`). Але тоді не забуваємо, що у вхідному файлі кілька тестових блоків, тому навіть якщо вже знаємо, що поточна відповідь «NO», дані блоку треба дочитати до кінця.

Спосіб (Б) Задача виділення фігурок у клітчатому полі (за допомогою, наприклад, пошуку в ширину або в глибину) досить стандартна, вказівки щодо її розв'язання можна знайти у багатьох джерелах. Вважаючи задачу виділення однієї фігурки відомою, дану задачу можна розв'язати так:

1. Завести допоміжний масив `[1..(200*200)div 3] of boolean`, смисл якого — «чи відбувався вже пошук фігурок, утворених даним числом».
2. Прочитати вхідні дані блоку в двовимірний масив.
3. Переглядати елементи двовимірного масиву рядок за рядком, і щоразу, знайшовши додатне число:
   (a) Якщо фігурку, утворену таким самим значенням, вже шукали — відповідь на поточний блок «NO», перевірку можна обривати.
   (b) Виділити фігурку, утворену поточним числом, замінюючи входження цього числа, наприклад, на протилежне (`mass[i][j]:=-mass[i][j]`), і рахуючи кількість клітинок у фігурці. Якщо кількість $\neq 3$ — відповідь на поточний блок «NO», перевірку можна припинити.
4. Якщо перевірка не була припинена, перевірити будь-яку одну з властивостей: або «кількість нулів у масиві від 0 до 2», або «для всіх значень від 1 до $(M*N)$div3 фігурка була виділена» — це дасть остаточну відповідь «YES» або «NO» для поточного блоку.

Технічне зауваження Вхідні дані досить великі за розміром, тому слід вибрати ефективний спосіб читання. Read мови Паскаль, scanf мови C та оператор `>>` класу `ifstream` мови C++ достатньо ефективні. А поєднання оператора `>>` об'єкта `cin` з `freopen`-ом — ні.

1.2 «Мутація» (Роман Єдемський).

1.2.1 Розв'язання

Після формалізації отримаємо таку постановку задачі:

Дано два рядки $\mathbf{A}$ та $\mathbf{B}$ довжиною до 40 000. Необхідно знайти усі такі символи рядка $\mathbf{B}$, що після видалення одного з них залишок $\mathbf{B}$ був би підпослідовністю рядка $\mathbf{A}$.

Є відома задача про перевірку того, що один рядок є підпослідовністю іншого. Тобто нехай ми перевіряємо, що $\mathbf{B}$ є підпослідовністю $\mathbf{A}$. Розв'язати таку задачу можна за допомогою жадібного алгоритму: перебираємо символи рядка $\mathbf{A}$, починаючи з початку, і якщо зустрічаємо символ, рівний першій літері рядка $\mathbf{B}$, то видаляємо цю літеру з нього. Якщо по закінченні виконання цього алгоритму рядок $\mathbf{B}$ залишився порожнім, то початковий рядок $\mathbf{B}$ був підпослідовністю рядка $\mathbf{A}$. Складність такого алгоритму є лінійною відносно суми довжин двох рядків.

Інтерпретуємо цей жадібний алгоритм як алгоритм динамічного програмування: для кожного префікса рядка $\mathbf{A}[1 \dots i]$ підрахуємо $\mathbf{P}[i]$ — найбільшу довжину префікса рядка $\mathbf{B}$, що $\mathbf{B}[1 \dots \mathbf{P}[i]]$ є підпослідовністю префікса $\mathbf{A}[1 \dots i]$. Нехай $\mathbf{P}[0] = 0$, тоді для $i > 0$: $\mathbf{P}[i] = \mathbf{P}[i-1]$, якщо $\mathbf{P}[i-1] = |\mathbf{B}|$ або $\mathbf{A}[i] \neq \mathbf{B}[\mathbf{P}[i-1]+1]$ та $\mathbf{P}[i] = \mathbf{P}[i-1]+1$ у випадку $\mathbf{A}[i] = \mathbf{B}[\mathbf{P}[i-1]+1]$ та $\mathbf{P}[i-1] < |\mathbf{B}|$. Якщо $\mathbf{P}[|\mathbf{A}|] = |\mathbf{B}|$ то $\mathbf{B}$ є підпослідовністю $\mathbf{A}$.

Адаптуємо цю ідею на випадок, коли нам треба видалити одну літеру з рядка $\mathbf{B}$, а саме: підрахуємо $\mathbf{P}[i]$ та аналогічну їй величину $\mathbf{S}[i]$ — найбільшу довжину суфікса рядка $\mathbf{B}$, при якому він є підпослідовністю суфікса $\mathbf{A}[i \dots |\mathbf{A}|]$. Припустимо, що ми видалили деякий символ x з рядка $\mathbf{B}$ і отримали рядок $\mathbf{C}(x) = \mathbf{B}[1 \dots x-1] + \mathbf{B}[x+1 \dots |\mathbf{B}|]$. Нескладно зрозуміти, що $\mathbf{C}(x)$ є підпослідовністю $\mathbf{A}$ тоді і тільки тоді, коли знайдеться префікс $\mathbf{A}[1 \dots i]$, що $\mathbf{B}[1 \dots x-1]$ є підпослідовністю $\mathbf{A}[1 \dots i]$, а $\mathbf{B}[x+1 \dots |\mathbf{B}|]$ є підпослідовністю $\mathbf{A}[i+1 \dots |\mathbf{A}|]$. Тобто для деякого індексу i виконується $\mathbf{P}[i] \geqslant x-1$ та $\mathbf{S}[i+1] \geqslant |\mathbf{B}| - x$, а з властивостей монотонності величин $\mathbf{P}$, $\mathbf{S}$ можна стверджувати, що існує також індекс $0 \leqslant j \leqslant |\mathbf{A}|$ такий, що $\mathbf{P}[j] = x-1$ та $\mathbf{S}[j+1] \geqslant |\mathbf{B}| - x$. Вважаємо, що $\mathbf{S}[|\mathbf{A}|+1] = 0$.

Звідси маємо такий алгоритм:

1. Побудуємо величини $\mathbf{P}$ та $\mathbf{S}$.
2. Перебираємо індекс j $(0 \leqslant j \leqslant |\mathbf{A}|)$:
   якщо $\mathbf{P}[j] + \mathbf{S}[j+1] \geqslant |\mathbf{B}| - 1$ та $\mathbf{P}[j] + 1 \leqslant |\mathbf{A}|$:
   індекс $\mathbf{P}[j] + 1$ додаємо до набору шуканих індексів.

1.3 «База даних» (Андрій Коротков).

1.3.1 Розв'язання

Введемо позначення: $\Gamma = (V, R)$ — граф, що задає ієрархію на підприємстві, V — множина вершин (співробітників), R — множина ребер (начальник, підлеглий). $last_0[t][x]$ — найбільший час виконання команди tx. $last[t][x]$ — найбільший момент часу, коли для вершини x була виконана операція t. $ans[x]$ — чи матиме права x після виконання заданої послідовності операцій.

$$last[t][x] = \begin{cases} last_0[t][x], & \text{якщо } t \neq 4; \\ \min(last_0[t][x], \min_{(y,x) \in R}(last[t][y])), & \text{якщо } t = 4. \end{cases}$$

Топологічно відсортуємо вершини графа. Обчислимо відповідь для вершини графу, починаючи з топологічно старших. В такому разі, при розгляді довільної вершини x всі її начальники будуть розглянуті попередньо. Умови володіння правами для x будуть виглядати наступним чином:

1) $last[1][x] > last[2][x]$;

2) $ans[y]$ AND $max(last[3][y], last[4][y]) > last[5][y]$, $(y, x) \in R$.

Складність алгоритму $O(N + M + K)$.

1.4 «Автомагістраль» (Ілля Порубльов).

1.4.1 Розв'язання

Спосіб з *гарантовано* допустимими часом роботи і об'ємом пам'яті використовує «meet in the middle», «2 вказівника» та модифікації злиття і має асимптотичну складність $O(2^{N/2})$ часу, $O(2^{N/2})$ пам'яті. Значно ефективнішого правильного розв'язку, мабуть, не існує, бо задача NP-повна (як узагальнення NP-повної задачі «є сукупність елементів, вибрати деякі з них так, щоб сума вибраних була якомога ближчою знизу до S»).

Враховуючи спосіб оцінювання даної оліміпади, непоганих результатів можна досягти також і перебірними (які не гарантують, що програма завжди вкладатиметься у обмеження по часу) та/або евристичними (які не гарантують правильності) методами — можна перепробувати кілька переборів та/або евристик і вибрати з них варіант, який приносить максимальні бали.

Оптимальний розв'язок — підтримка сукупності не домінованих пар Побудуємо для фрагментів магістралі по дві сукупності пар (t, p), тобто (час; вартість) — одна сукупність для шляхів (від початку), що закінчуються на безплатній дорозі даного фрагмента, інша — на платній. На попередніх фрагментах дороги можна міняти.

Якщо для $(a.t, a.p)$ та $(b.t, b.p)$ виконується $(a.t \leqslant b.t)$ and $(a.p \leqslant b.p)$, будемо називати пару $(b.t, b.p)$ *домінованою* — її можна викинути, бо пара $(a.t, a.p)$ гарантовано не гірша.

Твердження 2. *Сукупність не домінованих пар можна впорядкувати* $a_1.p < a_2.p < \ldots < a_L.p$*; при цьому одночасно виконуватиметься* $a_1.t > a_2.t > \ldots > a_L.t$.

Доведення. Впорядкувати $a_1.p \leqslant \ldots \leqslant a_L.p$ можна будь-яку сукупність. Різних пар $(a.t, a.p)$ та $(b.t, b.p)$ з $a.p = b.p$ не буде, бо при $a.t < b.t$ пара b буде домінованою, при $a.t > b.t$ — пара a, а при $a.t = b.t$ нема смислу зберігати дві копії однакової пари. Далі, при $a.p < b.p$ не може бути $a.t \leqslant b.t$, інакше пара a була б домінованою. □

Будуватимемо послідовності не домінованих пар, впорядковані $p\nearrow$, $t\searrow$. Для 1-го фрагмента ці послідовності тривіальні — єдина пара $\langle(a_1, 0)\rangle$ для безплатної дороги і єдина $\langle(b_1, c_1)\rangle$ для платної. Надалі, є впорядковані послідовності не домінованих пар для $(s-1)$-го фрагмента (позначимо їх *lastfree* та *lastpaid*), будемо будувати послідовності не домінованих пар для s-го (позначимо *nextfree* та *nextpaid*).

Аналогічно злиттю, будемо рухатися індексом i по *lastfree*, індексом j по *lastpaid*, щоразу порівнювати поточні пари *lastfree*$[i]$ та *lastpaid*$[j]$, і вибирати ту з них, де менше p. Тільки, на відміну від злиття:

1. Кандидатом на додавання у результат є не просто вибрана пара, а пара, побудована за такими правилами:
   (a) При побудові *nextfree* поле p копіюється з вибраної пари, а при побудові *nextpaid* до нього ще додається вартість проїзду по платній дорозі поточного фрагмента c_s.
   (b) Поле t збільшується: на час проїзду по самому фрагменту (a_s для *nextfree* або b_s для *nextpaid*); якщо відбулася зміна дороги (платна→безплатна або навпаки), то додатково ще на q_s.
2. В разі рівності *lastfree*$[i].p =$ *lastpaid*$[j].p$, вибираємо пару з меншим (з урахуванням всіх поправок п. 1b) часом.
3. Дописуємо побудовану пару до результату тільки коли або результат ще порожній, або t побудованої пари строго менше за t останньої наразі пари результату.

Вибір меншого p та п. 2 гарантують, що домінована пара не потрапить у результат раніше пари, яка її домінує. А п. 3 — що не потрапить пізніше.

Зберігання лише не домінованих пар, *як правило*, суттєво зменшує розміри сукупностей. Але можливі вхідні дані (наприклад: всі $q_i = 1$, всі $b_i = 3$, решту визначити як $a_i = c_i = 7 \cdot 2^i$), для яких розміри кожної з сукупностей не домінованих пар для i-го фрагменту будуть 2^{i-1}, що при $i \approx N \approx 42$ забагато. Тому, *крім* вже розглянутого етапу, треба користуватися також ідеєю meet in the middle. При розробці тестів планувалося, щоб ефективна побудова не домінованих пар (без meet in the middle) набирала 60–70% балів.

Оптимальний розв'язок — meet in the middle (з «2-ма вказівниками») Сукупності не домінованих пар можна будувати не для всіх фрагментів зразу, а двічі по $N/2$. Візьмемо $N_1 := N$ div 2, $N_2 := N - N_1$, і побудуємо сукупності не домінованих пар для N_1 фрагментів, починаючи від початку («ліва половина») та N_2 фрагментів, починаючи від кінця («пра́ва»). Тоді розміри кожної з послідовностей не перевищуватимуть $2^{20} \approx 10^6$. Остато́чні відповіді задачі будемо шукати, поєднуючи ці послідовності для «половинок». Причому, для поєднань дуже зручно, що послідовності задані в порядку $p\nearrow$, $t\searrow$.

Твердження 3. *«Ліва половинка» з характеристиками* $(left[i].t, left[i].p)$ *може бути частиною оптимального маршруту (вигляду «*min *час при вартості* $\leqslant S$*») лише при поєднанні з «правою половинкою», яка має значення* $right[j].p$*, найближче знизу до* $S - left[i].p$*.*

Доведення. «Праву половинку» з більшим значенням p брати не можна, бо шлях буде не допустимим (вартість $> S$). А взявши «половинку» зі значенням $right[k].p < right[j].p$, отримаємо (завдяки $p\nearrow$, $t\searrow$) $right[k].t > right[j].t$, тоді як час треба мінімізувати. □

Отже, при переборі i у порядку спадання, кожне відповідне за твердженням 3 значення j буде або тим самим, що для попереднього i, або більшим ($left[i].p$ зменшилося $\Rightarrow S - left[i].p$ збільшилося). Тобто, застосовна ідея «два вказівники»: організувавши цикл пошуку j всередині циклу перебору i, можна продовжувати пошук j з того місця, де спинилися при попередньому i, й усе проходження двома вказівниками працюватиме за $O(|left| + |right|)$.

Всього проходів 2-ма вказівниками треба зробити $2 \times 2 \times 2 = 8$ штук: min вартість для часу $\leqslant T$ чи min час для вартості $\leqslant S$; якою (безплатною чи платною) доро́гою закінчується ліва половина; якою права половина. Якщо одна з доріг безплатна, а інша платна, треба безпосередньо в даному проході врахувати $q[N_1]$.

Бажано *не* переписувати код 8 разів, а оформити як підпрограму і багатократно ви́кликати. Наприклад, підпрограм може бути дві: min вартість для часу $\leqslant T$ та min час для вартості $\leqslant S$; яка дорога з якою перевіряються, варто задати параметрами. Аналогічно, варто зробити однією процедурою (з різними параметрами) побудову сукупностей не домінованих пар $nextfree$ та $nextpaid$.

Інші способи формування не домінованих переліків Замість модифікацій злиття, можна формувати кожен перелік не домінованих пар так: включити до переліку пари, відповідні абсолютно всім 2^{N_i} шляхам; відсортувати за неспаданням p, а при рівних p за неспаданням t. Після цього, за один прохід, легко вибрати лише не доміновані пари, порівнюючи кожну поточну пару з останньою вибраною.

Такий підхід простіший для написання, але менш ефективний, бо сортування переліка довжиною $2^{N/2}$ потребує $O(2^{N/2} \log(2^{N/2})) = O(2^{N/2} \cdot N)$ дій, а рекомендований алгоритм формує правильно впорядкований перелік за $O(2^{N/2})$.

Ще один спосіб побудови не домінованих пар — виконання додаткових дій з `map`-ом (STL мови C++); див., зокрема, вказівки до задачі «Choosing a camera» студентської олімпіади SEERC–2011 (`codeforces.ru/blog/entry/2880`). Цей спосіб теж потребує $O(2^{N/2} \cdot N)$ дій.

При розробці тестів планувалося, щоб дані способи (при використанні meet in the middle) набирали 75–80% балів.

Розв'язання динамічним програмуванням при S, T **до** 10^5–10^6 Якщо S і T досить малі, щоб можна було підтримувати кілька масивів від 0 до $\max(S, T)$, задачу можна вирішити, розв'язавши серії підзадач *«Яку мінімальну суму* $S(i, r, t)$ *доведеться сплатити, щоб дістатися від початку до кінця* i*-го фрагменту, витративши не більш ніж* t *центів, причому* $r = 0$ *означає, що останній* i*-й фрагмент їхали по безплатній дорозі,* $r = 1$ *— по платній»*, та аналогічну серію $T(i, r, s)$.

При $S, T \sim 10^{16}$, цей спосіб практично неможливий. Але він гарантовано проходить 40% тестів, де значення до 10^5. В інших тестах можна усі значення відповідної природи поділити (з заокругленням до цілого), щоб поділені S та T були в межах до, наприклад, $4 \cdot 10^6/N$ (приблизно максимальне значення, при якому програма ще поміщається у обмеження по часу і пам'яті). Сума заокруглених може відрізнятися від заокруглення суми, тому такий спосіб не завжди даватиме правильну відповідь. Але є ймовірність, що якась

частина відповідей інших тестів теж буде правильною. Причому, якщо значення ділити саме з заокругленням (до найближчого), ця ймовірність вища, ніж коли ділити цілочисельно (без остачі).

Гілки та межі, інші перебірні методи Бажаючі можуть знайти інформацію про ці методи за ключовими словами «метод гілок та меж», він же «метод розгалужень та обмежень», «метод ветвей и границ», «branch and bound», а також «backtracking». Можна пробувати й інші перебірні або перебірно-евристичні методи («метод отжига», генетичні алгоритми, і ще багато інших).

При розробці тестів ставилася мета, щоб очевидні варіанти гілок та меж набирали 60–70% балів. Але поведінка гілок та меж сильно й малопрогнозовано залежить від дрібних змін, тому можливі значні відхилення від оголошених відсотків. Ще раз нагадаємо, що при використаному лояльному способі перевірки можна було пробувати міняти оціночні функції та інші деталі перебору й вибирати варіант, що набере найбільше балів.

Більш ефективними виявилися переборні підходи, які починаються з сортування за певною евристичною властивістю, яка визначає, припущення щодо якого фрагменту варто робити на зовнішньому рівні рекурсії, щодо якого (не обов'язково сусіднього) — на наступному, і т. д. Інше евристичне порівняння визначає, в якому порядку робити рекурсивні виклики: спочатку розглянути, що даний фрагмент їдуть по платній дорозі, а потім, що по безплатній, чи навпаки).