Розв'язання задач другого туру

27 березня 2012 р.

1 Завдання другого туру

1.1 «Розфарбування» (Роман Єдемський).

1.1.1 Розв'язання

Нехай деяке початкове розфарубвання **P** містить хоча б одну клітинку чорного кольору і гравцям в сумі належить зробити **K** ходів до завершення гри. Позначимо через $F(\mathbf{R})$ максимальну кількість білих клітинок в остаточному розфарбуванні, що може досягнути перший гравець при оптимальній грі обох, якщо початкове розфарбування — **R**. Оберемо клітинку чорного кольору у **P** і замалюємо її у білий колір, позначимо отримане розфарбування **P'**.

Твердження 1. *Для будь-якої фіксованої послідовності ходів кількість клітинок білого кольору в остаточному рахунку при початковому розфарбуванні **P'** не менша, аніж при початковому розфарбуванні **P**.*

Доведення. Справді, розглянемо будь-яку клітинку дошки. Якщо внаслідок послідовності ходів відбувається перефарбування цієї клітинки, то її остаточний колір не залежить від початкового розфарбування. Якщо ж унаслідок послідовності ходів не відбувається перефарбування клітинки, то її колір залежить лише від початкової позиції. Але за побудовою усі білі клітинки **P** є білими і у **P'**, тому кількість клітинок білого кольору в остаточному рахунку не зменшиться при заміні початкового розфарбування **P** на **P'**. □

Твердження 2. $F(\boldsymbol{P'}) \geqslant F(\boldsymbol{P})$.

Доведення. Розглянемо дві гри: гру **A** «фарбувашки» при початковому розфарбуванні **P** і гру **B** «фарбувашки» при початковому розфарбуванні **P'**. Нехай ми знаємо оптимальну стратегію для гри **A** і намагаємось побудувати стратегію, що дає не гірший результат для гри **B**. Зробимо хід у грі **B**, який є оптимальним першим ходом у грі **A**, позначимо його **u**. Нехай другий гравець, який слідує оптимальній стратегії в **B**, відповів нам деяким ходом **v** (якщо, звісно, $\mathbf{K} \neq 1$). Тепер припустимо, що у грі **A** було здійснено два ходи **u** та **v**. Нехай тепер оптимальним ходом є **x**. Здійснимо **x** у грі **B**. Якщо $\mathbf{K} \neq 3$, отримаємо у відповідь деякий хід **z**, і знову оберемо хід, виходячи з того, що у грі **A** здійснено вже **u**, **v**, **x**, **z**. Тобто для будь-якої послідовності ходів непарної довжини, здійсненої у грі **B**, ходом у відповідь ми обираємо оптимальний (за мірками гри **A**) хід у відповідь на цю ж саму послідовність ходів. Зауважимо, що оскільки перший гравець грав оптимально на кожному кроці (за мірками гри **A**), то отримана послідовність ходів призводить до не гіршого рахунку у грі **A**, аніж $F(\mathbf{P})$. В силу твердження 1 ця послідовність ходів у **B** призводить до не гіршого рахунку, аніж у грі **A**. Тобто, перший гравець може гарантувати собі рахунок не гірший, аніж $F(\mathbf{P})$, незалежно від ходів суперника, а отже $F(\mathbf{P'}) \geqslant F(\mathbf{P})$. □

Твердження 3. *Для обох гравців існує оптимальна стратегія, що включає лише перефарбування прямокутників розміром* $\min(\boldsymbol{N}, \boldsymbol{D}) \times \min(\boldsymbol{M}, \boldsymbol{D})$.

Доведення. Справді, нехай це не так, і на деякому кроці одному з гравців *необхідно* для досягнення оптимального результату перефарбувати прямокутник $\mathbf{Q}$ менших розмірів. Але з твердження 2 випливає, що використавши для перефарбування прямокутник з розміром $\min(\mathbf{N}, \mathbf{D}) \times \min(\mathbf{M}, \mathbf{D})$, який накриває $\mathbf{Q}$, гравець отримає результат не гірше, аніж при перефарбуванні прямокутника $\mathbf{Q}$. □

Твердження 4. *Нехай у початковому розфарбуванні було $\boldsymbol{W}$ білих клітин та перший гравець зробив хід, яким перефарбував $\boldsymbol{X}$ чорних клітин у білий колір. Тоді кількість білих клітин в остаточному розфарбуванні при оптимальній грі обох буде рівною $\boldsymbol{W} + \boldsymbol{X}$ при непарному $\boldsymbol{K}$ та $\boldsymbol{W} + \boldsymbol{X} - \min(\boldsymbol{N}, \boldsymbol{D}) \cdot \min(\boldsymbol{M}, \boldsymbol{D})$ при парному.*

Доведення. Нехай баланс розфарбування — це різниця між кількістю білих та кількістю чорних клітинок. Тоді перший гравець намагається максимізувати баланс, а другий — мінімізувати. Нехай у початковому розфарбуванні баланс дорівнює $\mathbf{T}$, тоді після першого ходу баланс дорівнюватиме $\mathbf{T} + 2 \cdot \mathbf{X}$. Унаслідок твердження 3 можна вважати, що обидва гравці на кожному своєму ході перефарбовують прямокутник розмірами $\min(\mathbf{N}, \mathbf{D}) \times \min(\mathbf{M}, \mathbf{D})$. Тому починаючи з другого ходу, незалежно від ходів суперника, у кожного з гравців є можливість змінити баланс на $2 \cdot \min(\mathbf{N}, \mathbf{D}) \cdot \min(\mathbf{M}, \mathbf{D})$ у вигідну для нього сторону.

Розглянемо кінцевий баланс: $\mathbf{T} + 2 \cdot X - \mathbf{A}_2 + \mathbf{A}_3 - \mathbf{A}_4 + \ldots + (-1)^{k+1} \cdot \mathbf{A}_k$, де $\mathbf{A}_i$ — це зміна балансу на i-му кроці. Позначимо $\min(\mathbf{N}, \mathbf{D}) \cdot \min(\mathbf{M}, \mathbf{D})$ за $\mathbf{U}$. Зрозуміло, що перший гравець може собі гарантувати щонайменше $\mathbf{Q} = \mathbf{T} + 2 \cdot \mathbf{X} - 2 \cdot \mathbf{U} + 2 \cdot \mathbf{U} - 2 \cdot \mathbf{U} + \ldots + 2 \cdot (-1)^{k+1} \cdot \mathbf{U}$, змінюючи на кожному своєму кроці баланс на $+2 \cdot \mathbf{U}$. З іншого боку, другий гравець може гарантувати собі результат, не більший від $\mathbf{Q}$, змінюючи на кожному своєму кроці баланс на $-2 \cdot \mathbf{U}$. Тому при оптимальній грі обох баланс у кінцевому розфарбуванні буде рівний $\mathbf{Q}$. З цього маємо, що при парному $\mathbf{K}$ баланс дорівнює $\mathbf{T} + 2 \cdot \mathbf{X} - 2 \cdot \mathbf{U}$, а при непарному — $\mathbf{T} + 2 \cdot \mathbf{X}$. Тобто при оптимальній грі при парному $\mathbf{K}$ маємо $\mathbf{W} + \mathbf{X} - \mathbf{U}$, а при непарному — $\mathbf{W} + \mathbf{X}$ білих клітин. □

Для розв'язання задачі залишається знайти прямокутник розмірами $\min(\mathbf{N}, \mathbf{D}) \times \min(\mathbf{M}, \mathbf{D})$ з максимальною кількістю чорних клітин.

1. Наївний розв'язок $O(\mathbf{N} \cdot \mathbf{M} \cdot \min(\mathbf{N}, \mathbf{D}) \cdot \min(\mathbf{M}, \mathbf{D}))$ набирає 50 балів.
2. Розв'язок $O(\mathbf{N} \cdot \mathbf{M} \cdot \min(\mathbf{N}, \mathbf{D}))$ з одномірними частковими сумами набирає 100 балів.
   Для кожного рядка поля i підрахуємо величину $\mathbf{S}[i][k]$, що дорівнює кількості клітин чорного кольору з початку рядка до його k-ї клітини включно. Нехай $\mathbf{S}[i][0] = 0$. Тоді щоб за $O(\mathbf{N})$ знайти кількість клітин чорного кольору на прямокутнику $[x, y] \times [c, d]$, необхідно підрахувати суму $\mathbf{S}[x][d] - \mathbf{S}[x][c-1] + \mathbf{S}[x+1][d] - \mathbf{S}[x][c-1] + \ldots + \mathbf{S}[y][d] - \mathbf{S}[y][c-1]$.
3. Розв'язок $O(\mathbf{N} \cdot \mathbf{M})$ із двомірними частковими сумами набирає 100 балів.
   Для кожної пари індексів поля (i, j) підрахуємо величину $\mathbf{S}[x][y]$, рівну кількості клітин чорного кольору на прямокутнику $[1, x] \times [1, y]$. Нехай $\mathbf{S}[0][y] = \mathbf{S}[x][0] = 0$. Тоді кількість клітин чорного кольору на прямокутнику $[x, y] \times [c, d]$ можна розрахувати так: $\mathbf{S}[y][d] - \mathbf{S}[x-1][d] - \mathbf{S}[y][c-1] + \mathbf{S}[x-1][c-1]$. Щоб підрахувати $\mathbf{S}[x][y]$ за $O(\mathbf{N} \cdot \mathbf{M})$ можна використати динамічне програмування: $\mathbf{S}[x][y] = \mathbf{S}[x-1][y] + \mathbf{S}[x][y-1] - \mathbf{S}[x-1][y-1] + c$, де $c = 1$ у випадку, коли (x, y) чорного кольору, та $c = 0$, якщо ні.

1.2 «Буфер обміну» (Данило Мисак).

1.2.1 Розв'язання

Барт діє в такий спосіб: копіює початкову фразу, вставляє її a_1 разів, копіює утворені $a_1 + 1$ фразу, вставляє їх a_2 разів, копіює утворені $(a_1 + 1)(a_2 + 1)$ фраз, вставляє їх a_3 разів,

$\ldots$, копіює утворені $(a_1+1)(a_2+1)\ldots(a_{k-1}+1)$ фраз, вставляє їх a_k разів, унаслідок чого дістає $(a_1+1)(a_2+1)\ldots(a_k+1)=n$ фраз. Для зручності позначимо a_i+1 через p_i. Неважко підрахувати, що Барт зробив усього $p_1+p_2+\ldots+p_k$ операцій і отримав число $n=p_1p_2\ldots p_k$. Отже, початкова задача зводиться до такої: розкласти число n на множники $p_1,p_2,\ldots,p_k$ так, щоб сума $p_1+p_2+\ldots+p_k$ була мінімальною. Залишається помітити, що якщо хоча б одне із чисел $p_1,p_2,\ldots,p_k$ у відповідному розкладі не буде простим, то його можна розкласти хоча б на два множники, більші за 1, зменшивши при цьому суму: якщо, наприклад, $p_1=d_1d_2$ і $d_1>1,d_2>1$, то $d_1+d_2<p_1$, а тому $d_1+d_2+p_2+p_3+\ldots+p_k<p_1+p_2+\ldots+p_k$. Таким чином, сума $p_1+p_2+\ldots+p_k$ є найменшою тоді, коли $n=p_1p_2\ldots p_k$ — розклад числа n на прості множники (при цьому деякі з чисел $p_1,p_2,\ldots,p_k$ можуть збігатися). Значить, щоб отримати відповідь, число n треба розкласти на прості множники, а потім знайти їхню суму.

Щоб розкласти задане число n на прості множники, можна перебрати в порядку збільшення всі потенційні дільники цього числа — числа від 2 до $n-i$, щойно n націло поділиться на якесь із цих чисел k, зменшувати значення n у k разів (і продовжувати перебір із k, а не з $k+1$, на випадок, якщо n ділиться на k у деякому степені). Утім, щоб суттєво покращити час роботи в найгіршому випадку, можна здійснювати перебір не від 2 до n, а від 2 до $\sqrt{n}$. Якщо по закінченні перебору, після всіх ділень, n не дорівнюватиме 1, це означатиме, що поточне значення n — просте число: інакше хоча б один із його дільників не перевищував би $\sqrt{n}$. Під час реалізації не слід забувати також, що початкове значення n може дорівнювати 1, у цьому випадку правильна відповідь — 0.

Знехтувавши часом, потрібним на ділення, можна оцінити складність поданого алгоритму як $O(\sqrt{n})$.

До розв'язання задачі можна підійти з допомогою динамічного програмування. У даному випадку стандартне його застосування буде таким: послідовно для всіх пар чисел $k,m, 1\le k\le n, 1\le m\le k$, ми підраховуємо найменшу кількість операцій $f(k,m)$ із буфером обміну, які приводять до стану (k,m), де k — кількість фраз у листі, а m — кількість фраз, які на даний момент зберігаються в буфері обміну. При цьому $f(1,1)=1$, а коли $k>1$, то $f(k,m)$ дорівнює: $f(k-m,m)+1$, якщо $k-m\ge m$; $\min\{f(k,i)+1 \mid 1\le i\le k-1\}$, якщо $m=k$; ∞ в інших випадках («∞» позначає, що відповідного стану неможливо досягти в принципі; на практиці можна використовувати досить велике число, яке достеменно перевищує правильну відповідь). Коли ми підрахуємо всі значення, відповіддю буде число $\min\{f(n,i) \mid 1\le i\le n-1\}$.

Складність такого алгоритму можна оцінити як $O(n^2)$; кількість пам'яті, яку використовує алгоритм, теж дорівнює $O(n^2)$. Але якщо зауважити, що $f(k,m)\ne\infty$ лише в тому випадку, якщо m є дільником k, то для кожного фіксованого k можна знаходити всі дільники за $O(\sqrt{k})$ і рахувати $f(k,m)$ лише для відповідних значень m. Це дасть оцінку $O(n\sqrt{n})$ як на час виконання алгоритму, так і на використовувану пам'ять (якщо зберігати результати обчислень для кожного k не індексованим по m масивом, а списком).

Цікавим є той факт, що коли реалізовувати ті самі рекурентні співвідношення з допомогою простої рекурсії, можна зекономити порівняно з простішим підходом динамічного програмування не лише оперативну пам'ять, але й час виконання програми — принаймні, для n достатньо малих, щоб на рекурсію не забракло пам'яті у стеку.

За допомогою динамічного програмування можна створити і такий алгоритм, який працюватиме за $O(\sqrt{n})$. Для цього достатньо зауважити, що якщо виписати кількості фраз, які опинялися в буфері обміну на оптимальному «шляху» до n фраз, матимемо послідовність, у якій кожне наступне число ділиться на попереднє. Якщо ввести функцію $g(k)$, яка дорівнює мінімальній кількості операцій, потрібних для отримання k фраз, зможемо записати співвідношення: $g(1)=0, g(k)=\min\{g(i)+k/i \mid i \text{ ділить } k, 1\le i<k\}, k>1$. Порахувавши $g(k)$ послідовно для всіх дільників n, матимемо відповідь — $g(n)$. Це займе

$O(\sqrt{n} + d^2(n)) = O(\sqrt{n})$ часу і стільки ж пам'яті (через $d(n)$ позначено кількість дільників числа n).

1.3 «Перехрестя» (Андрій Коротков).

1.3.1 Розв'язання

Відстань d будемо вважати допустимою, якщо існує розташування перехрестя, при якому максимальна відстань від міст до нього не перевищує d. Допустима відстань має такі властивості.

1. Якщо d_1 — допустима відстань, і $d_2 > d_1$, то d_2 — також допустима відстань.
2. Якщо d_1 — не допустима відстань, і $d_2 < d_1$, то d_2 — також не допустима відстань.

Ці властивості дозволяють скористатися бінарним пошуком для знаходження оптимальної відстані як мінімальної допустимої відстані. Попередньо відсортуємо міста за x-координатою.

Наведемо алгоритм перевірки відстані d на допустимість.

Нехай x-координата вертикальної магістралі рівна x_0. Тоді міста з x-координатою із проміжку $[x_0 - d, x_0 + d]$ будуть задовольняти умови допустимості. Решта міст будуть знаходитися на відстані, більшій за d, від вертикальної магістралі. Отже, відстань від цих міст до горизонтальної магістралі не повинна перевищувати d. Це можливо тоді і лише тоді, коли різниця максимальної та мінімальної y-координат цих міст не перевищує $2d$.

Методом двох вказівників переберемо найлівіше і найправіше міста, які будуть віддалені від вертикальної смуги не більш ніж на d. На кожному кроці це буде певний проміжок $i..j$. Отже, відстань від міст $1..i-1$ та $j+1..k$ до горизонтальної магістралі не повинна перевищувати d. Для ефективної перевірки можливості відповідного розташування горизонтальної магістралі попередньо обчислимо мінімальну і максимальну y-координати міст на проміжках вигляду $1..i$ та $i..k$.

Складність алгоритму — $O(K \log N)$.

1.4 «Опукла оболонка» (Ярослав Твердохліб).

1.4.1 Розв'язання

По-перше, помітимо, що всі операції видалення декількох точок можна розбити на елементарні операції видалення однієї точки. Оскільки кожна точка додається до верхнього ланцюга не більше одного разу, то й у сумі операцій видалення буде не більше ніж N.

Нехай ми вже побудували опуклу оболонку множини точок в певний момент часу. Розглянемо відрізок, що сполучає найлівішу і найправішу точки з множини. Частину опуклої оболонки, яка лежить над цим відрізком разом з двома його кінцями, назвемо верхнім ланцюгом опуклої оболонки. Іншу частину оболонки та два кінці обраного відрізка назвемо нижнім ланцюгом опуклої оболонки. Таким чином, найлівіша і найправіша точки належать обом ланцюгам одночасно. Верхній і нижній ланцюги разом з обраним відрізком утворюють опуклі багатокутники. Площа всієї опуклої оболонки дорівнює сумі площ цих двох багатокутників. Отже, достатньо навчитись виконувати всі необхідні операції з верхнім ланцюгом, а для нижнього все буде аналогічно.

Розв'язок на 50 балів Заведемо стек, в якому будуть зберігатись точки верхнього ланцюга. Додавати нову точку будемо так само, як і в алгоритмі Грехема. А саме, видалятимемо останню точку зі стека до тих пір, поки поворот між двома останніми точками стека (назвемо їх A і B) і новою точкою (D) не стане правим (тобто векторний добуток $AB \times AD < 0$). Після цього додамо нову точку до стека. Для того, щоб окрім самого ланцюга

знаходити ще й його площу, потрібно разом з кожною точкою зберігати площу опуклого багатокутника, утвореного всіма точками, які в стеку знаходяться перед нею (включаючи її). Тоді при додаванні нової точки достатньо до вже збереженої площі ланцюга додати площу трикутника, утвореного першою і останньою точками зі стека, а також новою точкою.

Ми вже вміємо додавати точки до верхнього ланцюга і рахувати після цього його площу. Але описаний вище алгоритм не дозволяє видаляти точки з множини. А саме, при додаванні нової точки ми видаляємо декілька останніх точок зі стека. Тепер при операції видалення ми маємо знову повернути ці точки до стека. Для цього заведемо ще один стек, куди ми будемо перекладати точки при видаленні їх з першого стека. Якщо потрібно повернути декілька видалених точок в перший стек, то ми можемо взяти їх із другого. Потрібно лише для кожного запиту додатково запам'ятати розмір другого стека для того, щоб ми знали, скільки точок потрібно повернути до верхнього ланцюга. Цей алгоритм може обробляти всі типи операцій і має часову складність $O(NM)$, де M — це кількість рядків вхідного файлу, у яких просять видалити декілька останніх точок. Набирає такий розв'язок 50 балів.

Розв'язок на 100 балів Для отримання повного розв'язку потрібно зрозуміти, яке місце є найповільнішим у цьому алгоритмі. Розглянемо наступну ситуацію: нехай в нас є побудований верхній ланцюг якоїсь множини точок (зберігається у стеку). Після цього ми додаємо нову точку, яка лежить набагато вище всіх інших, і тому зі стека видаляються усі точки, крім першої. Тепер ми видаляємо цю точку, і всі ті точки знов перекладаються з одного стека в інший. Отже, за 2 операції алгоритм зробив $O(N)$ дій. Якщо їх повторювати багато разів, то складність буде $O(N^2)$.

Замінимо в нашому розв'язку стек звичайним масивом, для якого будемо додатково пам'ятати кількість точок, яка належить верхньому ланцюгу (розмір стека). При додаванні нової точки знайдемо бінарним пошуком кількість точок, яку потрібно видалити зі стека перед тим, як покласти цю нову точку до нього. Замість перекладання усіх цих точок у другий стек ми покладемо туди лише розмір верхнього ланцюга, а також всю інформацію про останню з точок, що видаляються (тобто її номер у масиві, її координати, а також площа многокутника, пов'язаного з нею). Після цього ми замінимо цю точку на нову, перерахуємо для неї площу і змінимо розмір ланцюга. При цьому всі інші точки, які мали бути видалені, залишаються в масиві без змін (проте вони знаходяться за межами ланцюга). При видаленні останньої точки нам достатньо змінити лише розмір ланцюга і замінити одну точку на ту, що зберігається у другому стеку. Складність такого розв'язку — $O(N \log N)$.