Динамично програмиране

Александър Георгиев

Динамичното програмиране (оптимиране) е един от най-важните раздели на състезателната информатика, главно поради честото (ако не и преобладаващо) срещане на такива задачи по състезания. Почти няма международна олимпиада, на която поне единия ден да не се даде някакъв вид динамично програмиране; също така в големи тренировъчни програми от типа на USACO или TopCoder този вид задачи е особено често срещан. Това се дължи не само на интересните проблеми и приложението на тази техника в реални задачи, но също така и на сравнително лесното изготвяне на задачи и тестове за тях.

Да започнем с това какво точно представлява динамичното оптимиране – това е техника на програмиране, която значително съкращава нужното време за решаване на даден проблем, използвайки вече получени резултати на предходна стъпка от алгоритъма. В повечето случаи тя помага за свеждането на сложността на даден алгоритъм от експоненциална до полиномиална. Идеята е дадено нещо да не се изчислява повече от веднъж, ако сме сигурни, че резултатът от изчисленията ще е същият, а вместо това да се записва в таблица с вече изчислени стойности, която може да се използва при повторно попадане в същата ситуация.

Съществуват няколко различни вида задачи, които се решават с помощта на динамично програмиране. Те се разделят на подкатегории спрямо това, как точно се кодира ситуацията (state). Ситуация ще наричаме дадено положение по време на изпълнение на алгоритъма, което се определя еднозначно от набор аргументи (например позиция в даден масив, най-малък използван елемент и т.н.). Съществено важно е да се отбележи, че е *задължително* аргументите, които се използват за кодиране на ситуацията, да я определят еднозначно – тоест при каквито и да е други условия и еднакви такива аргументи, решението на ситуацията да е едно и също. Също така е необходимо всяка подзадача да е от вида на предходната; с други думи ако началната задача зависи от някакви аргументи, то подзадачите да зависят също от тях.

Техниките за динамично програмиране, които ще опишем в статията, включват динамично с един аргумент, с 2 и повече аргумента, по битова маска, с кръстосана битова маска, както и по шаблон. Също така ще разгледаме така наречената „оптимизация на вътрешния цикъл".

Преди да започнем със самите специфичности на различните категории динамично оптимиране е хубаво да кажем, че съществуват два основни начина на писане на такъв сорт задачи. Първият е същинско динамично – изпълнява се итеративно попълване на таблицата с изчислени стойности. Този метод има две предимства – отчасти итеративната реализация, макар и с малко, е по-бърза от рекурсивната; и второ – в някои задачи той може да ни позволи съкращаването на една размерност на таблицата със стойности (тоест оптимизация на паметта), което може да се окаже от решаваща важност (пример за това е задачата miners, IOI 2007, ден втори). Другият метод е така наречената мемоизация (memoization), чието основно предимство е това, че се пише по-лесно (рекурсивно) и в повечето случаи по-бързо (което е важно на състезания като TopCoder). Тя директно следва тривиалното решение на задачата (пълно изчерпване), като в хода на действие попълва таблица с вече изчислени стойности, и при повторно попадане в същата ситуация директно връща вече изчисления резултат. В някои случаи вторият начин може да е дори по-бърз като време за изпълнение, поради факта, че изчислява само стойностите, които са нужни за отговора на задачата, докато първият метод изчислява цялата таблица.

Нека започнем с най-тривиалния вид динамично – такова с кодиране на ситуацията с един елемент. За да го използваме ни е нужна задача, която зависи единствено от едно нещо (позиция в масив, най-голям/малък използван до сега елемент, брой бурканчета мед, които Мечо Пух е изял, каквото и да е). Например, нека е дадена числова редица с краен брой елементи, като от нас се иска да вземем нейна подредица, чиито елементи са ненамаляващи (задачата за най-дълга нарастваща подредица). Нека редицата бъде дадена в масив a[], а броят елементи е n. Тривиалното решение би било да изберем един от n-те елементи за първи, след което да изберем един от следващите елементи за втори и така нататък. Апроксимираната сложност на това решение е n! (n факториел), тоест експоненциална. Дори за кратки редици от около 20 елемента това би било твърде бавно и на супер мощен компютър. Какво се забелязва – ако можем да използваме даден член от редицата,

например a[k], то независимо какви елементи сме избрали преди него (те със сигурност са по-малки), оптималната редица след него е винаги една и съща. Нека въведем допълнителен масив dyn[] с n елемента, който в началото е инициализиран с някаква невалидна стойност, например -1 (не можем да имаме редица с дължина -1). В него записваме оптималната намерена стойност, като не изчисляваме наново всички възможни продължения на редицата от позиция k, ако имаме записано число, различно от -1 в dyn[k]. Как ни помага това? За всяка от n-те позиции в редицата, ние пробваме да я продължим с всяко следващо число точно веднъж. Грубо броят операции, който ни е необходим, е n * n = n^2. Свеждането на сложността на задачата от експоненциална до полиномиална е огромно подобрение – вече редици с 20000 елемента ще се обработват за под две секунди. (Още една възможна оптимизация е да се ползва съкращаване на вътрешния цикъл, което ще разгледаме по-късно. То би довело до сложност O(n*logn), с която бихме могли да обработваме редици с над 2 милиона елементи за под две секунди.)

В допълнение може да се отбележи, че -1 в повечето задачи, където отговорът е естествено число, е удачна стойност за инициализация на таблицата със стойности, тъй като може да се ползва функцията memset(dyn, -1, sizeof(dyn)), която задава желаната стойност значително по-бързо от нормалния начин с обхождане на масива с цикли. Забележете, че числото, което функцията memset() приема като втори аргумент е char и запълва масива с char стойности, тоест ако имаме int dyn[128] и зададем memset(dyn, 3, sizeof(dyn)), елементите на масива dyn[] НЯМА да бъдат запълнени с 3. Изключение правят стойностите 0 и -1 поради особеностите на записване на числата в двоичен вид в компютрите. Също така бъдете внимателни с какво инициализирате масивите си – ако използвате валидна стойност, то програмата ви няма да дава грешни резултати, но ще изчислява някои ситуации отново и отново, и има голям шанс да прехвърли времевото ограничение! (пример: задача ChangingSounds, TopCoder SRM 366 – стойността -1 е валиден отговор и докато на малките примери решение с инициализиран с -1 масив работи достатъчно бързо, то на по-големи примери води до TL.)

Нека се върнем на задачата с най-дългата нарастваща подредица, но този път искаме да намерим броя на нарастващите подредици с точно k елемента (k <= n). Да кажем, че търсим подредица с точно 5 елемента, като

вече сме избрали 2 от тях и последният избран е a[j]. Задачата е идентична с началната – все едно трябва да изберем подредица с точно 3 елемента от редицата от a[j+1] до a[n], по-големи или равни на a[j]. Очевидно ако използваме динамичната таблица от предходната задача няма да постигнем нищо, тъй като няма да знаем колко дълга редица търсим от текущия елемент нататък. Нищо не пречи обаче да го кодираме в ситуацията – тоест да ползваме двумерна динамична таблица dyn[pos][rem], където pos указва на коя позиция в редицата сме, а rem – колко елемента още ни трябват. Тези 2 неща еднозначно определят ситуация, следователно така може да се реши задачата. Можем да доразвием идеята нататък – както двумерен, така масивът може да е три и повече мерен, в зависимост колко аргумента са нужни да определим дадена ситуация. Почти всички задачи в тренировъчната програма на USACO, които се решават с техниката динамично оптимиране, са с няколкомерни динамични таблици. Друга много типична задача, която илюстрира силата на тази техника, е задачата за раницата (knapsack problem) – дадени са обекти с определена цена и тегло, както и раница, която може да издържи максимално n килограма. Кои елементи да вземем, така че раницата да не се скъса, и в същото време стойността на предметите да е максимална?

Понякога употребата на многомерна таблица не е удобна. Представете си, че имаме динамична задача, която зависи от 9 аргумента. Дори без числа и променливи в себе си, dyn[][][][][][][][][] изглежда грозно, да не говорим колко тромаво и сложно за писане. Нека вземем за пример задача C от Турнира за Купата на Декана на ФМИ, 2 декември, 2007г. Там динамичната таблица, която ни е нужна, е именно 9-размерна, като всяко измерение е число от 0 до 5 включително. Лесният начин, по който можем да се измъкнем от ужасното индексиране и предаване на параметри, е като ги кодираме в едно единствено число. Това можем да направим, като представим всяко измерение като цифра в бройна система с база 6. Така първото измерение ще е b[0] * 6^0, второто ще е b[1] * 6^1, и така нататък, като последното ще е b[8] * 6^8. Успяхме да представим ситуацията като едно единствено число в бройна система 6. Сега вече можем да използваме едномерна таблица с 10077696 клетки, която да ползваме за изчислени стойности.

Един от по-нестандартните видове динамични, е такова, чиято ситуация се пази под формата на битова маска. В най-честите случаи имаме сравнително

малък брой неща (почти винаги по-малко от 25), като искаме да ги съчетаем по такъв начин, използвайки всяко нещо само веднъж, че да получим оптимален резултат при някакви правила на оценяване. Очевидно всеки предмет може да бъде или използван или не. Един възможен начин да кодираме ситуацията би бил да направим многомерна таблица dyn[2][2][2]..[2] – всяка размерност отговаря за един елемент – 0, ако не е използван и 1, ако е използван. Това разбира се, би било много неудобно при повече елементи (например 20), но използвайки горната схема за кодиране, можем да направим едно двоично число, за което всеки бит ще отговаря за един елемент – 0, ако не е използван и 1, ако е. Тъй като много напомнят на шаблон с дупки (където вече са използвани предмети) можем да наричаме това представяне на ситуацията като едномерен шаблон или по-разпространеното – битова маска (шаблон се използва по-често при двумерна маска). Поради нативното представяне на числата в двоичен вид в компютърните архитектури сме много улеснени откъм манипулации с такъв вид числа. Нека маската е представена в променливата *mask*.

- Повдигането на 2 на дадена степен е елементарно – можем да ползваме операторът "<<" вместо предварително да изчисляваме стойностите. Например ако искаме да обявим масив с достатъчно място, за да съберем битовата маска за 20 елемента е достатъчно да направим int dyn[1<<20], което ни дава 2 на 20-та степен – точно това, което ни беше нужно.
- Ако искаме да проверим дали имаме 1 на позиция k в маската, можем да използваме: if (mask & (1 << k)).
- Ако искаме да направим бит-а на позиция k единица (тоест използваме k-ти предмет) можем да използваме просто mask = mask | (1 << k); (работи независимо какъв е бил той преди това).
- Ако искаме обратното – да го направим нула, независимо какъв е бил преди това, можем да ползваме mask = mask & (~(1 << k));
- Ако искаме да го променим (от нула в едно или от едно в нула) ползваме mask = mask ^ (1 << k);

По-подробна информация за битовите операции и хакове можете да намерите в тази чудесна статия.

С тези прости средства можем бързо да обработваме битовата маска за каквито цели ни е нужна. Да се върнем на идеята на динамичното по битова маска. Една примерна задача, която много добре илюстрира името този тип кодиране на ситуацията, е задача distance (A2, Есенен Турнир, Шумен, 2007). В нея накратко има 22 точки в равнината и искаме да ги групираме по двойки така, че сборът от разстоянията между точките във всяка двойка да е максимален. Нека вече сме избрали 6 от тях – и сме в същата задача, само че с 16 вместо началните 22 точки. Кои 6 точно сме избрали можем да отбележим като единички в едно двоично число с 22 бита и да пуснем рекурсивно решаване на задачата, като налагаме ограничението, че не можем да ползваме точки, на чиято позиция има 1 в битовата ни маска. Така на всеки ход от рекурсията ще слагаме по 2 нови единички в маската и ще се намираме в по-малка подзадача.

Един по-специален вариант на горния метод е динамично по кръстосана битова маска. Например ако имаме матрица NxM, и върху нея можем да прилагаме дадени операции, то кръстосаната битова маска може да ни пази дали сме приложили операция върху i-тия ред или върху j-тия стълб (таблицата ни ще е dyn[maskRows][maskCols]). Модификацията е интересна и въпреки, че е рядко срещана, тя показва, че с малко изменение на познат метод можем да получим решение на непозната задача. Примерна задача, където се ползва тази техника, е TurningMaze, SRM 385, Division I.

Една от нестандартните динамични задачи, която е представена както в тренировъчната програма на USACO, така и в подготовката на националите ни, е TwoFive, Day 1, IOI 2001. Там имаме квадратна матрица 5x5, която почваме да запълваме от горен ляв ъгъл и можем да слагаме следващ елемент, само ако над него и от ляво на него имаме вече друг (или съответно горна или лява стена). По този начин за дадена колона ни интересува не само дали вече има сложени елементи, но и колко са те на брой. Тъй като от ограниченията на задачата не може да имаме дупка между два елемента в дадена колона, то не е нужно да пазим отделна битова маска за всяка колона, а е достатъчно да пазим само колко вече сме сложили. Така динамичната ни таблица става от вида dyn[6][6][6][6][6] (естествено можем да ползваме метода за компресия на измеренията, но в случая не е нужно).

Като по-нагледен пример можем да дадем следната таблица:

A	D	J	.	.
B	E	K	.	.
C	G	.	.	.
F	I	.	.	.
H	.	.	.	.

Която би била определена в динамичното като dyn[5][4][2][0][0].
Тази техника се нарича „динамично по шаблон" (наистина, ситуацията се определя от шаблон по позициите) и се среща по състезания (както се вижда и от международен характер).

Основно свойство на динамичното оптимиране е, че жертваме памет в полза на време за изпълнение. Понякога обаче паметта може да не достига, затова трябва да правим някакви оптимизации по запазването на ситуацията (които все пак да запазват еднозначността). Един от вече споменатите методи в този контекст е използването на същинско динамично, вместо рекурсия с мемоизация. Това често ни позволява да ползваме една размерност по-малко, което в повечето случаи оптимизира нужната памет с хиляди пъти. Но има задачи, за които този подход не позволява намаляне на размерите на таблицата, или изобщо не може да бъде приложен.

Тук ще разгледаме 3 начина, за съкращаване на необходимата памет при специфични задачи. Нека се върнем на задачата distance, която разгледахме по-рано. Ограниченията на задачата налагат използване на 4-ри байтов тип (въпреки, че double или long long ще ни свършат работа откъм точност, то таблицата ще стане твърде голяма за това ограничение на паметта). Дори при използване на unsigned int (който стига за максималния отговор), нужната памет е твърде много. Какво можем да направим? Забелязва се, че на всеки ход добавяме по 2 точки (две единици в битовата маска), тоест броят вдигнати битове винаги е четен. Разсъждавайки можем да съобразим, че последният бит всъщност не ни е нужен – ако в останалите 21 бита имаме нечетен брой единици, то 22-рия е единица (за да се запази четността), и обратно. Това ни позволява да ползваме двойно по-малка таблица, което вече спокойно влиза в ограниченията на задачата.

Друга не толкова очевидна оптимизация (но реално вършеща същото като горната) ще илюстрираме с помощта на задачата MarblesInABag.
Накратко - имаме торба със сини и червени мъниста, като знаем първоначално

по колко от всеки вид имаме (до 4 хиляди). На всеки ход на играта 2-ма играчи теглят по едно мъничко и единият печели при определени правила. Самото решение с мемоизация е очевидно (единственото, което трябва да пазим, е броя мъниста от единия и от другия цвят). Не е нужно да пазим кой е на ход, защото можем да играем по 2 хода наведнъж и така винаги първият да е наред. За съжаление паметта не достига - нужната ни таблица double dyn[4000][4000] е малко под 128 мегабайта, което е почти двойно над ограничението от 64. Единият начин да решим този проблем е да направим по-сложното „същинско" динамично и така свеждаме паметта до линейна по N, тоест 2 * double[4000]. Но няма ли по-лесен начин за решение на дадения проблем (няма ли друг живот!)? Има разбира се! Достатъчно е да забележим, че тъй като на всеки ход се вземат по 2 мъниста, то след всеки ход четността на мънистата се запазва (ако в началото са били нечетни, след всеки ход остават нечетни и обратно). Какво ни помага това? Ами можем да „елиминираме" един бит от динамичната таблица, който пази четността на, да кажем, сините мъниста. Тоест таблицата ще е от вида dyn[4000][2000]. Реално пазим броя на червените и броя на половината сини мъниста. Целочисленото деление на 2 намалява второто число на половина, но за съжаление губи информацията за последния бит на сините мъниста (евентуално, ако умножим по 2 резултата ще получим с 1 по-малко от първоначалното). За даден ход да възстановим броя на наличните сини мъниста е достатъчно да умножим подадената стойност по 2 и да се погрижим сбора на сини + червени да е от същата четност като първоначалния сбор. Например: в началото е имало 3372 червени и 1337 сини мъниста. На текущата стъпка сме подали 2762 червени и 413 сини. Умножаваме сините по две и имаме 2762 червени, 826 сини. Гледаме четността на първоначалния сбор: (3372 + 1337) % 2 == 1. Текущата четност от своя страна е (2762 + 826) % 2 == 0. Значи ни липсва едно синьо мъничко, тоест текущите сини мъниста не са 826 ами 827. Ако пък четността съвпадаше, щяхме да запазим броя на сините мъниста 826. Като цяло оптимизация на последния бит може да се реализира, когато таблицата ни е разредена (sparse), тоест по лесен начин можем да докажем, че не ползваме поне половината от полетата. Ако знаем и кои полета не ползваме и знаем защо не ги ползваме, можем да мислим за оптимизация на някое от измеренията.

Като трети пример ще покажем задачи, в които не просто можем да оптимизираме (намалим) едното измерение на динамичния масив, а направо можем да го премахнем. Такъв пример е задачата ПРЕЛИВАНЕ (ден втори, НОИ 2003) – имаме три съда с определена вместимост, като е дадено началното количество вода и правим преливания от един съд в друг по определени правила. Очевидно можем да дефинираме ситуация като количество вода във всеки съд – dyn[200][200][200]. Но какво би станало ако ограничението за вместимост на съдовете не беше 200, а 2000? Динамична таблица dyn[2000][2000][2000] би била прекалено голяма за заделяне в паметта. От друга страна в задачата е казано, че при преливането не се губи вода, тоест сумата от трите съда е константна: s1 + s2 + s3 = n. Да предположим, че не знаем колко вода имаме в третия съд (x = s3). Получаваме равенството s1 + s2 + x = n, което можем да преобразуваме до x = n – s1 – s2. Тоест, ако пазим стойностите на само 2 от трите съда можем да намерим количеството вода в третия. Това ни разрешава да ползваме и по-малка таблица dyn[2000][2000], която вече спокойно влиза в ограниченията. Като трети пример можем да дадем задача A1 apple (НОИ 2007, трети кръг). Там двама души извършват едновременно ходове по правоъгълно поле, като могат да ходят единствено надолу и надясно. Очевидно на всеки ход позицията им се увеличава или по колона, или по ред, като общия брой ходове е текущият ред + текущата колона (ако сме започвали броенето от (0, 0)). При ограничение на редове и колоните до 70 не можем да пазим позицията и на двамата като dyn[row1][col1][row2][col2], тъй като нужната памет става твърде много. За сметка на това, както отбелязахме вече, ако знаем позицията на единия човек, то знаем и колко хода вече сме направили. Така можем да запазим само текущия ред на втория човек и по броя ходове (изчислен от първия) да намерим колоната, в която се намира. Така динамичната таблица е от вида dyn[row1][col1][row2] и е напълно достатъчна за целите на задачата.

Едно от най-важните неща на всеки компютърен алгоритъм е неговата ефективност – колко време ще отнеме изпълнението му за дадени ограничения на входните данни. За разлика от някои други задачи, сложността на алгоритми, базирани на динамично оптимиране, е сравнително лесна за изчисление. Като цяло най-лесната апроксимация на броя необходими операции в най-лошия случай се смята като броят на клетките в динамичната таблица се умножи по

броя продължения на дадена ситуация. Наистина – всяка ситуация се изчислява точно по веднъж, като за нейното пресмятане са нужни k на брой операции - възможните продължения на алгоритъма. Например нека се върнем на задачата за най-дългата нарастваща подредица – там динамичната таблица е с размерност дължината на редицата (n), като от всяка позиция трябва да проверим всеки от следващите елементи за евентуално продължение (грубо сметнато – отново n). Така общата сложност е O(n*n). Или пък задачата за търговския пътник, която при достатъчно малък брой градове може да се реши с динамично оптимиране, базирано на битова маска за посетени градове и още една размерност, указваща текущия град. Така динамичната таблица е dyn[n][2^n] и от всяка позиция можем да изберем да отидем до някой от останалите n-1 града. Това грубо е n*2^n*n (n*2^n за всяка клетка от таблицата, *n за всяко продължение), което ни дава и самата сложност на алгоритъма – O(n^2*2^n).

Напоследък динамичното оптимиране стана толкова често срещано по състезания, че нормалните задачи (дори с част от странните си видове, които показахме по-горе) станаха стандартни и лесни за повечето състезатели. От друга страна те не са претенциозни откъм сложни структури данни или дълги алгоритми след като се измисли как точно да се кодира ситуацията. Затова напоследък се наблюдава тенденцията да се дава съчетание на този вид задачи с използването на по-сложна структура от данни или алгоритъм. Тези задачи биват наричани „динамично с оптимизация на вътрешния цикъл", тъй като цикълът, който въртим в дадена позиция от динамичното бива заменен с използването на по-бърз похват (двоично търсене) или по-сложна структура данни (RMQ, индексни дървета). Така ние съкращаваме броя необходими операции с някаква променлива n и на нейно място добавяме log(n). Тук можем да припомним задачата за най-дългата ненамаляваща подредица и нейното свеждане до алгоритъм със сложност O(n*log(n)) вместо началната експоненциална или O(n*n) с помощта на динамична таблица.

Нека направим следната статистика: преди 2006-та година няма почти нито една задача с оптимизация на вътрешния цикъл по български състезания. Оттогава насам: Задача 1 и Задача 6 от конкурса по програмиране на Мусала Софт и PC Magazine, 2006-2007г.; Задача А2 – Автобусна Линия от пролетния

турнир по информатика, Ямбол, 2006г.; Задача А6 – Сортиране – НОИ 2006; Задача А5 – Матрьошки – НОИ 2007г. Две конкурсни задачи и 3 задачи от контролни за национален отбор, две от които от националната олимпиада.

Както казахме и по-рано, има два начина за писане на оптимизация на вътрешния цикъл. Тъй като начинът с алгоритъм (двоично търсене) е описан в книгата на Преслав Наков и Панайот Добриков – Програмиране = ++Алгоритми, а и като цяло той може да се ползва в сравнително по-тесен кръг от задачи, тук ще се спрем на метода със структура от данни (индексни дървета). Като пример отново ще разгледаме задачата за най-дълга ненамаляваща подредица поради простотата й. Забележка – в случая трябва да използваме същинско динамично, а не рекурсия с мемоизация, поради несъвместимостта й с оптимизацията.

Каква е идеята на тази оптимизация? Да кажем, че в момента се опитваме да изчислим оптималното продължение от позиция k. Изчислили сме оптималните подредици, започващи от позиции k+1 до n (тоест след текущата). „Вътрешният цикъл" търси продължение, като гледа само такива числа, които са по-големи или равни на текущото. Това търсене е линейно и прави сложността на алгоритъма общо квадратна (с динамично запомняне на резултати). Би било хубаво ако не се налагаше да обхождаме всички продължения, а имахме лесен начин да намерим тази позиция в интервала [k+1, n], която едновременно съдържа по-голямо или равно число от текущото и ни носи оптимален резултат. Всъщност при предните стъпки на алгоритъма сме събрали информацията, която ни е нужна, за да е възможно това, но при обикновеното динамично не я използваме цялостно. Примерно вече сме обходили всички числа, и знаем кои от тях са по-големи от текущото, но по никакъв начин това не ни помага. Използвайки структурата от данни индексни дървета или в случая нейната разновидност - RMQ - можем да осъществим бързия отговор на въпроса „кое е най-доброто продължение, по-голямо или равно на числото от текущата позиция?". Ако не сте запознати с тези структури Ви препоръчвам горните две статии, тъй като първо са на български, второ са доста разбираемо обяснени от бивши национали по информатика.

Нека имаме редицата:
3, 8, 1, 7, 5, 6, 2, 9, 8, 11, 42, 13, 17, 101, 4, 13, 15, 666, 1337, 1024;

и по случайност се намираме на позиция 10 (нула индексирано), тоест в числото 42. Досега намерените отговори на подредицата са оцветени в синьо:
3, 8, 1, 7, 5, 6, 2, 9, 8, 11, 42, 13, 17, 101, 4, 13, 15, 666, 1337, 1024;
И имат съответно резултати: 5, 4, 3, 5, 4, 3, 2, 1, 1. Примерно един оптимален отговор, започващ веднага след 42 от числото 13 би бил подредицата 13, 13, 15, 666, 1337, която има 5 члена (както е и отбелязано при резултатите). Тъй като текущото число е 42, нас ни интересува продължението с максимална дължина, чиито първи член е по-голям или равен на 42. RMQ (Range Minimum/Maximum Query) има точно тази функционалност – показва минималния или максималния елемент в даден интервал. В случая ни интересува максималния елемент от всички полета, по-големи или равни на 42. Структурата данни ще претърси стойностите 3, 2, 1, 1, отговарящи съответно на продълженията от числата 101, 666, 1337 и 1024, които отговарят на условията:
3, 8, 1, 7, 5, 6, 2, 9, 8, 11, 42, 13, 17, 101, 4, 13, 15, 666, 1337, 1024;
От таблицата на резултатите това са: *5, 4, 3, 5, 4, 3, 2, 1, 1*.
Очевидно максималният до сега намерен резултат, започващ от число, по-голямо или равно на 42 е 3, започващ от 101. Тогава за позиция 10, тоест числото 42, можем да запишем, че оптималният резултат е 4, със следващо число 101. След обновяване на RMQ-то сме в сходна задача, със следната редица:
3, 8, 1, 7, 5, 6, 2, 9, 8, 11, 42, 13, 17, 101, 4, 13, 15, 666, 1337, 1024;
И следната таблица с резултати: 4, 5, 4, 3, 5, 4, 3, 2, 1, 1. Както лесно се вижда, единственото невалидно продължение е числото 4 на позиция 14, тъй като е по-малко от 11 (текущото). Аналогично алгоритъмът намира за 11 максимален отговор 6 (с продължение числото 13 на позиция 11) и продължава нататък.
С други думи – обновяваме резултата в интервала [текущо_число, безкрайност) (където безкрайността е стойност, по-голяма от максималното ни входно число). Операциите в индексни дървета са с логаритмична сложност, така че на мястото на линейното обхождане със сложност O(n) имаме O(log(n)). Ако пък максималното число е много голямо, но броят на входните числа не е – може да „компресираме координатите", тоест на всяко число да съпоставим друго, в повечето случаи по-малко, което запазва наредбата (тъй като разликата между числата не ни интересува, а само това, дали е по-малко или не).

Съвет: не пишете оптимизация на вътрешния цикъл, освен ако не е НЕОБХОДИМА. Винаги е по-лесно да се дебъгне просто динамично, отколкото индексно дърво!

Съществува и друг вид оптимизация на вътрешния цикъл, която не разчита на сложна структура от данни, а по-скоро на правилно наблюдение (съответно елиминиране на много излишна работа). Така понякога е възможно оптимизацията на даден алгоритъм не от n към log(n) (като множител) ами директно съкращаване на n. Пример за такава задача е споменатата по-рано Задача 1 от конкурса по програмиране на фирма Мусала Софт и PC Magazine. В нейния анализ можете да намерите обяснение какви трябва да са изпълнените условия за да е възможно това и какво точно наблюдение трябва да направите.

От къде може да се подготвяте за задачи, базирани на динамично оптимиране? На практика отвсякъде – едва ли има много тренировъчни програми, които да не покриват този толкова важен дял от състезателното програмиране. Двете основни места, където има достатъчно задачи, са тренировъчната програма на USACO и TopCoder practice rooms. И двата сайта предлагат анализ на задачите (USACO след като ги решите, TopCoder и преди това в match editorials). Ако не сте много навътре с този тип задачи (а и с информатиката като цяло) е препоръчително да порешавате TopCoder High School мачовете, които освен, че предоставят не твърде сложни задачи, дават възможност и за чудесна тренировка на техниката динамично оптимиране (от около 50 мача има над 40 динамични задачи). От къде можете да четете повече на тази тема? На български – от книгата на Преслав Наков и Панайот Добриков – Програмиране = ++алгоритми. На английски – има сравнително добри статии с примери, както из тренировъчната програма на USACO, така и в туториъла за динамично програмиране на сайта на TopCoder.

Препратки:
http://www.xkcd.com/399/
http://ioi2007.hsin.hr/tasks/day2/miners.pdf
http://www.topcoder.com/stat?c=problem_statement&pm=7973
http://infoman.musala.com/contests/kupa_na_dekana/2007/tasks_acm.doc

http://www.topcoder.com/tc?module=Static&d1=tutorials&d2=bitManipulation
http://infoman.musala.com/contests/fall/2007/tasks_A.doc
http://www.topcoder.com/stat?c=problem_statement&pm=8471
http://www.csie.ntu.edu.tw/~b93103/study/ioi/ioi2001/day1/twofive/twofive.rtf
http://www.topcoder.com/stat?c=problem_statement&pm=9754&rd=13485
http://infoman.musala.com/contests/noi/2003/round3_tasks.html
http://infoman.musala.com/contests/noi/2007/round3/A123_bul.doc
http://konkurs.musala.com/content/c06/p1/shoney.pdf
http://konkurs.musala.com/content/c06/p6/contest.pdf
http://infoman.musala.com/contests/spring/2007/tasks_a.doc
http://infoman.musala.com/contests/noi/2006/round3/A/day2/a456.pdf
http://infoman.musala.com/contests/noi/2008/round3/AB.rar
http://judge.openfmi.net/wiki/index.php/Индексни_Дървета
http://judge.openfmi.net/wiki/index.php/RMQ
http://konkurs.musala.com/content/c06/p1/analyze.pdf
http://ace.delos.com/usacogate/
http://www.topcoder.com/tc
http://www.topcoder.com/tc?module=Static&d1=tutorials&d2=dynProg