Упражнение №3 по САА

Динамично оптимиране

Понякога се сблъскваме със задачи, за които няма известен ефективен алгоритъм. За много от тях можем да пуснем алчен алгоритъм, като се надяваме той да намери оптималния отговор. Съществува голям набор от задачи, обаче, където алчните алгоритми не винаги биха го намерили. В този случай единственото нещо, което можем да направим за да намерим прецизен отговор, е да пуснем изчерпващ алгоритъм. Но това, за съжаление, би било с експоненциална сложност и би изискало ужасно много време за дори сравнително малки инстанции на проблемите.

Запознайте се с техниката "Динамично Оптимиране"!

Вместо това, в някои по-специфични задачи можем да ползваме друга техника, която свежда сложността на решението ни от експоненциална до полиномиална. Тази техника се нарича *Динамично Оптимиране*. Нейната основна идея се базира на това да не питаме един и същи въпрос два (или повече) пъти, ако отговорът би бил един и същ при всяко от питанията.

Нека разгледаме следния пример:

Отишли сме на рожден ден, като предварително всички гости сме дали пари за общ (по-голям) подарък. Както често се случва, когато сме събирали парите, още не е било решено какво точно ще бъде взето за рожденика. Десетина минути след като сме се появили на партито, някой от гостите ни пита "Абе, какво взехме в крайна сметка на рожденика?". Ние също не знаем, затова питаме друг човек (който евентуално трети, и т.н.), докато най-накрая успеем да научим какъв е подаръка. Двадесет минути по-късно, някой друг гост също ни пита за подаръка. Този път не е нужно отново да питаме нашите познати - вече можем директно (веднага) да отговорим на въпроса, тъй като сме "научили" (запомнили) неговия отговор.

Този пример илюстрира директно каква е идеята за решение на дадена задача: ако сме достигнали състояние (state), до което сме стигали и преди, и нещо повече, знаем, че отговорът от този state ще е винаги един и същ, можем директно да върнем намерения резултат, вместо да го изчисляваме отново.

Обърнете внимание на нещо изключително важно - можем да приложим тази техника *само* ако отговорът не би се променил с времето (не се влияе от външни фактори). Защо това би се счупило, ако има външни фактори, лесно можем да покажем чрез друг (донакъде сходен) пример.

! Техниката "динамично оптимиране" е приложима само ако отговорът на дадена подзадача не зависи от външни (неконстантни) променливи!

Нашата добра приятелка Ели е с нас на партито, като друга наша приятелка, Крис, ни пита къде е тя. Ние също не знаем, затова тръгваме с Крис и намираме Ели в басейна. Два часа по-късно

Станчо се появява и също ни пита къде е Ели. Макар и ние вече да сме намерили веднъж отговора ("в басейна"), редица външни фактори може да са се намесили през това време върху неговата достоверност (станало е студено, на нея ѝ е омръзнало да стои там, отишла е да си сипе... натурален сок и др.). Най-правилното решение би било да отговорим отново "не знам" и отново да започнем търсенето.

State

Страхотно! Значи ще се опитваме да научим компютъра (по-точно програмата ни) да запомня някакви неща. За да можем да направим това, трябва да измислим начин, по който да представим въпросите в удобен за компютъра формат.

Всъщност това обикновено е най-трудната част при решаването на динамични задачи - как точно да кодираме въпроса? Очевидно "Къде е Ели?" ("Къде е батко!?") не е въпрос, който лесно може да бъде разбран от компютър. В най-най-честия случай той ще бъде зададен чрез числа - а какво точно означават числата ще зависи силно от задачата. Типичен стейт би бил едно единствено цяло число X, което ще бъде разбрано от програмата ни като "Какъв е отговорът за X?". Стойностите на аргументите в момента, в който ни е зададен въпроса, ние наричаме "състояние" (state). Интуитивно би било "Какъв е отговорът за 13?" да има един отговор, докато "Какъв е отговорът за 42?" да има друг. Съответно 13 и 42 са различни "стейтове" на въпросната задача.

Понякога въпросът може да е по-сложен, например "Колко е била средната заплата в държава X през година Y?". Забележете, че тук имаме два аргумента на въпроса - държава и година.

Това не е голям проблем (даже е много типично стейтът ни да се състои от повече от едно "измерение"). Тук два различни стейта биха били "България, през 1987 година", "Германия, през 2012 година". Нещо повече, стейтовете са различни дори ако всички аргументи с изключение на един съвпадат. Например "България, през 1987 година" е различно от "България, през 2012 година".

Терминът "измерение" на динамичното идва от нуждата за лесен начин за описване на естествен език от колко аргумента зависи то. Например такова с един аргумент е "едномерно", такова с два аргумента е "двумерно" и т.н.

Забележете също така, че докато годината (Y) е цяло число, то държавата със сигурност не е. А както казахме, стейтът ни в огромна част от задачите е съставен само от числа. За целта можем да направим някаква "наредба" на държавите (примерно по азбучен ред) и така, когато кажем държава 3, ще знаем, че имаме предвид не държавата с име "3", ами третата (всъщност четвъртата) държава в списъка от държави, наредени по азбучен ред.

Често няма да се налага изобщо да "подреждаме" обектите, които искаме да индексираме. Обикновено ние ги получаваме като входни данни, като вместо да ги преподреждаме (примерно сортираме), ние просто ползваме реда, в който са ни дадени на входа.

В много по-редки случаи във въпросите ще включваме стрингове или числа с плаваща запетая, тъй като те са по-трудни за индексиране в компютърната памет (не могат да се

ползват като индекс в масив). Все пак съществуват и такива задачи, като с няколко от тях ще се сблъскаме малко по-късно.

Освен директното ползване на числа (като година, индекс в масив, т.н.) за аргументи на динамично, които ще разгледаме в тази тема, съществуват и други, по-екзотични начини за съставяне на аргументи (и, съответно, стейта). Чрез някои от тях описваме множества, чрез други - форма, чрез трети сливаме представянето на няколко аргумента в един и т.н. Те често са по-сложни както за измисляне, така и за писане, което прави задачите (малко или много) по-трудни.

Динамична таблица

В миналия параграф видяхме как да формулираме въпроси към компютъра. Също така е важно да видим как компютърът ще пази отговорите за тях. Нека се запитаме, как бихме пазили отговора, ако имахме само един въпрос? Логично - в променлива. Сега, как бихме пазили отговорите, ако имахме 1000 въпроса? Отново отговорът е логичен - в масив. Как бихме могли да пазим отговора, ако въпросът ни зависи от два аргумента? Ами отново просто - в двумерен масив. Масива, в който пазим вече изчислените отговори, наричаме "динамична таблица".

Нека отново разгледаме хипотетичното динамично, което би отговаряло на въпроса "Колко е била средната заплата в държава X през година Y?". Първо, трябва да видим какви са границите на въпросните аргументи. Това *много* зависи от самата задача - например светът съществува от стотици хиляди години, но в реална задача Y би имал стойности между (примерно) 1800 и 2012, тъй като едва ли бихме имали статистика за по-предни години. Така де факто имаме само 213 различни стойности, тоест само 213 клетки в едното измерение на динамичната таблица, вместо стотиците хиляди, които бихме могли да имаме в други задачи, в които отново единият ни аргумент е "година".

? *Редовна грешка при писането на динамично е да се обяви таблица с първо измерение за аргумента X, второ измерение за аргумента Y, но в кода да се индексира Y в първото измерение, а X във второто. Освен ако таблицата не е с еднакви размери, това най-вероятно би довело до излизане извън масива и crash на програмата ни, или, в по-гадния за дебъгване случай, печатане на грешни отговори, въпреки привидно правилен код.*

Различните стойности на X също могат сравнително лесно да бъдат определени. Например, повечето статистики сочат, че в света има 196 държави - тоест валидни стойности на X биха били между 0 и 195, включително. Какво става обаче, ако считаме само държави в европейския съюз? Тогава този брой би бил значително по-малък. Ами ако включим измислени държави (например Byteland, Нарния, Македония...)? Тогава този брой може да надхвърли 196. Отново - това число трябва да определим за конкретната задача и дадените ограничения. Да приемем, че ограниченията наистина са 196 държави и 213 години. Тогава двумерен масив (динамична таблица) с размери [196][213] би ни свършила работа да пазим отговорите на въпроса "Колко е била средната заплата в държава X през година Y". От какъв тип да е въпросният масив? Ами, зависи от въпроса, на който искаме да отговорим. В случая *средна заплата* би ни навело на мисълта, че отговорът ще е число с плаваща запетая, тоест double

dyn[196][213].

Макар и най-често отговорът да е някой от вградените типове (примерно int или double), не е невъзможно връщаният отговор да съдържа повече от една променлива (примерно string или struct), а понякога цяла структура данни (примерно vector или set).

! Изключително честа грешка при този тип задачи е да се декларира динамична таблица, която е по-малка, отколкото трябва. И докато декларирането на по-голям масив не е проблем, то по-малък такъв води до грешки в индексацията и почти винаги - до грешни отговори (ако не crash в самата програма). Лесно решение на проблема е просто да заделяте масиви с по няколко елемента по-големи от нужното във всяко измерение. Например горният масив би било по-добре да бъде заделен като [200][220], вместо [196][213].

Нелоша практика е (когато имаме достатъчно памет) да се ползват размери на измеренията, които са степен на двойката, която е по-голяма или равна на нужния размер. Например вместо [196][213] бихме ползвали [256][256].

! Жизнено важно е ако ползваме подхода с един масив да го инициализираме със стойност, която е наистина невалидна. Какво би станало, ако *-1* беше валиден резултат? Интересното е, че функцията би продължила да връща верните отговори. Но бихме "изгубили" свойството на динамичното за клетки, чиито резултат е *-1* - попадайки в такава клетка ние бихме я изчислявали отново и отново, независимо, че вече сме намерили нейния резултат. Това би могло да доведе до time limit, който е изключително труден за намиране (тестовете, на които се чупи, са сравнително специфични).

Тук възниква важен въпрос: как знаем кои отговори са вече изчислени и кои не? Най-лесният начин за справяне с това е да ползваме втори масив (от тип bool), който пази тази информация. Това, обаче, не винаги е удобно, тъй като трябва да го поддържаме аткуален през цялото време (тоест при всяко бъркане в динамичната таблица, трябва да бъркаме и в този масив). Това води до малко повече код и нужна памет, което, макар и не особено голям проблем, много състезатели предпочитат да избегнат (когато това е възможно). Нека например динамичната таблица е от тип int, а връщаните стойности са естествени числа. Тъй като знаем, че числото *-1* се поддържа от типа (цяло число), а също така е невалиден отговор (не е естествено число), можем предварително да запълним целия масив с минус единици, и когато се питаме дали вече сме изчислили даден резултат:

1. проверяваме каква е стойността в масива за въпросния стейт
2. ако стойността е различна от *-1*, значи въпросният резултат вече е изчислен и връщаме директно тази стойност (тя е резултата от изчисленията)
3. ако стойността е *-1*, значи въпросният резултат още не е сметнат и продължаваме с изчисленията

Първа примерна задача

Преди да продължим нататък, ще дадем един сравнително лесен пример, върху който да показваме разни работи. Най-най-стандартният такъв е задачата за намиране на числата на Фибоначи. Той е толкова удобен за демонстрация на застъпващи се подзадачи, че се ползва *почти винаги* като първи пример за динамично оптимиране. На нас, обаче, ни е омръзнал, затова ще го пропуснем.

Добре де, това не е единствената причина да не го ползваме. Да намерим N-тия член на числата на Фибоначи има много очевидно итеративно решение (следващо директно от дефиницията), за което изобщо не ни трябват сложнотии като рекурсия и динамично. Съответно много от вас биха се запитали "Защо да си правим живота сложен, като може да е лесен?". Това, всъщност, е много добра практика при състезателното програмиране. Колкото по-просто е едно решение (което, разбира се, е достатъчно ефективно), толкова по-добре.

Вместо това ще разгледаме следната задача, която е почти също толкова проста, но чието математическо решение не е толкова тривиално.

Дадено ви е поле с $1 \leq N \leq 30$ реда и $1 \leq M \leq 30$ колони. Започвайки от горния ляв ъгъл, по колко начина можете да стигнете до долния десен такъв, движейки се само надолу или надясно?

Например при поле с три реда и три колони има шест такива пътя: { (надясно, надясно, надолу, надолу), (надясно, надолу, надясно, надолу), (надясно, надолу, надолу, надясно), (надолу, надясно, надясно, надолу), (надолу, надясно, надолу, надясно), (надолу, надолу, надясно, надясно) }.

По-нататък в темата ще разгледаме няколко възможни решения на този проблем.

```
#include <cstdio>
long long recurse(int remRows, int remCols) {
	// Ако нямаме избор (отишли сме в краен ред или колона)
	if (remRows == 0 || remCols == 0)
		return 1;
	long long ans=0;
	ans += recurse(remRows - 1, remCols); // Движим се надолу
	ans += recurse(remRows, remCols - 1); // Движим се надясно
	return ans;
}
int main(void) {
	int n=18, m = 18;
	fprintf(stdout, "%lld\n", recurse(n - 1, m - 1));
	return 0;
}
```

Наистина, учудващо е как толкова кратък и прост код може да бъде *толкова* неефективен. Още при сравнително малки аргументи (към N = M = 18) тя става бавна, а към N = M = 20 вече *много* бавна. Защо? Нали дъската тогава има едва 400 клетки? Това означава, че трябва да изчислим *само 361 стойности*. Защо се бави толкова?

<u>Задача:</u>

Добавете код в началото и края на програмата за да проверите за колко време работи тя. Сравнете времената за работа при N = M = 18 и N = M = 20.

Задача:
Добавете код, който да изчисли, колко пъти се изчислява стойността за remRows = 5, remCols = 5 при дъска с размери N = M = 18. Пресметнете броя на извикванията и на remRows = 3, remCols = 3, както и за remRows = 2, remCols = 2.

От бектрек към динамично

След като сме намерили начин да задаваме въпроса (определили сме стейта) и начин да пазим отговорите за него (заделили сме динамична таблица), остава да модифицираме бектрека, който написахме, за да го превърнем в динамично (и да спре да бъде толкова бавен =)). Всъщност промените, които трябва да направим, са съвсем минимални. При всяко "питане" - тоест при влизане в рекурсията - трябва да проверим дали вече не знаем отговора, и при всеки "отговор" - тоест излизане от рекурсията - трябва да запаметим току-що намерения отговор в таблицата. Така двете места, в които трябва да пипнем кода ни, са съвсем в началото и съвсем в края на рекурсията.

Макар и в повечето случаи да можем да направим проверката дали вече сме изчислили даден отговор съвсем в началото на рекурсията, е много по-сигурно (нашият код да не крашне) ако първо се справим с частните случаи, и едва *после* направим проверката. Защо е така?

! Ако някои от частните случаи отнемат доста време (тоест не са константна проверка) трябва да се замислите дали не можете да ги преместите след проверката в таблицата. Така ще гарантирате, че тези изчисления ще се извършат по най-много веднъж за всяка клетка на таблицата.

Ами частните случаи обикновено са единични if-ове, тоест константни проверки, което не е по-бавно от проверка в динамичната таблица (и, съответно, няма да навреди на скоростта на програмата ни). Нещо повече, частните случаи обикновено се справят с индекси, които са отдолу (обикновено по-малки от нула) или отгоре (по-големи от *N*) на ограниченията. А когато използваме аргументите да индексираме в масив с *N* елемента, то и двата случая биха довели до излизане извън него. Извършвайки хендълването на частните случаи преди индексирането в масива с изчислените стойности би се справило с тези нелегални индексирания.

Memoization

Тази техника на оптимизиране на бектрек се нарича мемоизация. Съществува и друг (итеративен) начин за строене на динамична таблица, който ще разгледаме малко по-късно. В редица случаи той би могъл да бъде по-бърз и по-ефективен откъм памет, но изобщо не е толкова интуитивен и логичен (за човек), колкото рекурсията с мемоизация.

? Много хора (както в България, така и в международни форуми) наричат метода "меморизация", което всъщност звучи логично, тъй като думата "memorize" на английски означава "уча наизуст". Все пак правилният термин е "memoization", който произлиза от латински и има специфична употреба в информатиката.

Решение

Сега да приложим наученото в дадената примерна задача!

Първо - има ли шанс тя да се решава с динамично оптимиране? Казахме, че структурата е от желания тип - има много припокриващи се подзадачи, които са от типа на началната задача. Имаме две измерения, ограничени до 30, което чудесно би се събрало в паметта в двумерна таблица с размери, примерно, [32][32]. Искаме да отговорим на въпрос "по колко начина", което, както казахме, е един от най-силните индикатори за динамично.

Супер, значи до тук няколко от хинтовете бяха покрити. Сега да пробваме да мемоизираме горенаписаното изчерпващо решение.

Трябва да забележим, че броят начини да *завършим* пътя до крайната клетка *не зависи* от това как сме стигнали до текущата такава. Следователно ще се опитваме да запомним отговорите на въпроса: "По колко начина можем да стигнем до края, ако можем да мърдаме надолу още *remRows* пъти и надясно още *remCols* пъти?".

Обявяваме динамичната таблица, инициализираме я преди да извикаме рекурсията, добавяме проверка дали вече сме намерили отговора на даден въпрос веднага след частните случаи, и последно запомняме новите отговори веднага преди да ги върнем от рекурсията. Това изглежда по следния начин:

```
#include <cstdio>
#include <cstring>

const int MAX = 32;
long long dyn[MAX][MAX];

long long recurse(int remRows, int remCols) {
    // Ако нямаме избор (отишли сме в краен ред или колона)
    if (remRows == 0 || remCols == 0)
        return 1;

    // Проверяваме дали отговорът не е вече запомнен
    if (dyn[remRows][remCols] != -1)
        return dyn[remRows][remCols];

    long long ans = 0;
    ans += recurse(remRows - 1, remCols); // Движим се надолу
    ans += recurse(remRows, remCols - 1); // Движим се надясно

    // Запаметяваме отговора в таблицата и го връщаме
    return dyn[remRows][remCols] = ans;
}

int main(void) {
```

```
    int n = 30, m = 30;
    // Инициализираме динамичната таблица
    memset(dyn, -1, sizeof(dyn));
    fprintf(stdout, "%lld\n", recurse(n - 1, m - 1));
    return 0;
}
```

Втора примерна задача

За затвърждаване ще направим почти същото, само че в малко повече детайли върху една малко по-сложна задача.

Ели е наследила от баба си (Нора) винарска изба. В нея има $N \leq 200$ бутилки вино, наредени в редица. За простота ще ги номерираме с числата от *0* до *N-1*, включително. Техните начални цени са неотрицателни цели числа, които са ни дадени в масива *P[]*. Цената на *i*-тата бутилка е дадена в P_i. Колкото повече отлежават бутилките, толкова по-скъпи стават те. Ако бутилка *k* е отлежала X години, нейната цена става $X*P_k$.

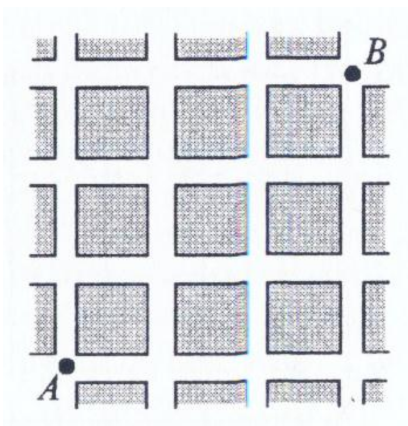
В завещанието си бабата на Ели е поискала всяка година внучка ѝ да продава по една от тях, като избира или най-лявата или най-дясната останала. Каква е максималната сума пари, която Ели може да спечели, ако продава бутилките в най-добрия за нея ред? Считаме, че бутилките са отлежавали 1 година, когато бива продадена първата от тях.

Например ако имаме 4 бутилки с цени $\{P_0 = 1, P_1 = 4, P_2 = 2, P_3 = 3\}$, оптималното решение би било да продаде бутилките в реда {0, 3, 2, 1} за печалба 1*1 + 2*3 + 3*2 + 4*4 = 29.

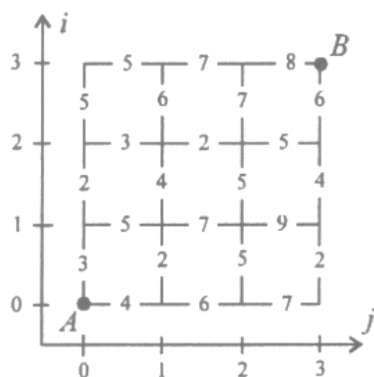
Задача 3 – Движение на североизток

В един град има правилна правоъгълна система от улици както е показано на фиг. 1. Пешеходец трябва да започне движението си от т. A и да отиде до т. B, като спазва правилото, че от всяко кръстовище може да тръгне на север или на изток. За преминаването на всяка отсечка от улица, заключена между две кръстовища, се заплаща определена такса. Схематично това може да се представи както е изобразено на фигура 2, където са зададени конкретни числени стойности на пътните такси.

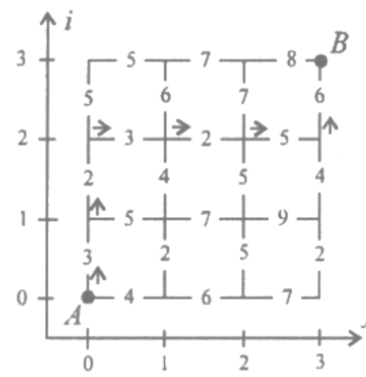
В задачата се търси каква е минималната сума, която трябва да се плати за да се премине маршрута?



Фиг. 1. Движение от A до B в един град



Фиг. 2. Пример с пътни такси за всяка улица



Фиг. 3. Възможен маршрут с цена 21

Източници:

1. Александър Георгиев, Динамично оптимиране, част I, http://informatika.bg/lectures?topic=DP1
2. Динамично оптимиране, Open FMI http://judge.openfmi.net:9080/mediawiki/index.php/%D0%94%D0%B8%D0%BD%D0%B0%D0%BC%D0%B8%D1%87%D0%BD%D0%BE_%D0%BE%D0%BF%D1%82%D0%B8%D0%BC%D0%B8%D1%80%D0%B0%D0%BD%D0%B5
3. Емил Келеведжиев, Динамично оптимиране, Анубис 2001 www.tasheva.info/files/Dinamichno-optimirane.rar