5-ти кръг 2012/2013 година. Задача crossword.  
 Подход към решението на Ивайло Странджев

На първо място искам да отбележа че нямам почти никакъв опит със **C#** преди да участвам в този кръг и затова е напълно възможно кодът ми да не спазва най-добрите възможни практики.

Задачата от пети кръг се различаваше от всички останали за тази година с това, че тестването се извършваше през *web service*. Това определя някои особености в дизайна на решението ми и затова преди да опиша самите алгоритми, които използвам, искам да напиша няколко идеи, които се надявам да са полезни за останалите състезатели.

### Дизайн

Като правило *web service*-ите имат навик да се ненадеждни. Дори и на най-големите състезания не е рядкост *service*-ът да падне или пък да отговаря много бавно. Ето защо отделих немалко време да направя така, че да мога да тествам задачата си локално. Освен по-бързо тестване, това ми позволи да работя по задачата дори и *offline* – когато нямам връзка с интернет.

#### LocalService

В моето решение реализирах това чрез полиморфизъм – създадох един базов клас *ServiceBase* и два наследника *WebService* и *LocalService*. *WebService* е директно прокси към сайта на състезанието и не съдържа никаква реална логика, докато *LocalService* прочита теста от файл, намиращ се на локалната ми файлова система, по време на изпълнението на метода *Init*. Форматът, който използвах за тестовете, е много подобен на това, което финалният тестер създава с единствената разлика че *Q* се въвежда на същия ред непосредствено след броя редове и колони, а не след цялото поле. Създадох отделна функция *dumbTest* (понеже вече не мога да променя решението, което съм предал, се налага да ползвам имената на функциите от предаденото, някои от които умишлено са малко сгрешени ☺ ), която изпълнявайки *getField* и *nextWord* „прочита” тест от *web service*-а и го запазва във файл на локалната ми система.

Много е важно локалният *service* да хвърля грешки когато и *web service*-ът би го направил, за да може да се откриват грешки в решението. След като имплементирах локален тестер известно време изпълнявах всяка заявка и на локалния и на уеб *service*-а и валидирах, че те връщат едно и също, за да съм сигурен, че не съм допуснал грешка в имплементацията. Такава грешка определено щеше да е пагубна, понеже почти не съм тествал решението си с друго освен с моя локален тестер. Измерване показа че тест, който на *web service*-а се изпълнява за около 5000 милисекунди при мен завършва за около 40. Това ми позволи да изпълня близо 1000 теста преди да предам финалното си решение (което на *web service*-а би отнело близо час и половина).

#### StateWrapper

Друго нещо, което реализирах е клас, който нарекох *StateWrapper*. Той представлява „умно” прокси към *ServiceBase* инстанция, което помни текущото състояние. Така в този клас аз помня кои букви са написани в кои клетки на таблицата, колко време е изминало от началото на решието ми (за да съм сигурен, че няма да работи повече от 10 секунди), колко са текущите ми точки, както и кои думи са ми „известни”. Макар и да е възможно да се ползва метода *getField*, за да се следи текущото състояние на полето, всяко извикване на метод от *web service*-а е несравнимо по-бавно от изчисление, което се извършва локално. Ето защо аз пазя състоянието при себе си, като така си спестявам много време. При извикване на метода *skip*, на нашата програма се разкриват 5 думи, които също съхранявам в класа *StateWrapper* като „известни”.

Последната функция на StateWrapper е да ме предпази от грешка. Макар и когато тествам локално да искам да получавам грешка за всеки невалиден ход, при финалното ми решение това би довело до 0 на съответния тест. Ето защо написах *StateWrapper*, така че когато тествам локално да хвърля грешки, но при финалното ми решение вместо невалиден ход да вика *EndGame*. Макар и с 1000 успешно преминати теста да съм доста уверен, че решението ми никога не прави невалидни ходове реших, че е по-добре да се подсигуря.

Имплементацията на всичко описано по-горе ми отне почти половината от времето, което отделих за кръга.

#### 1.3 Генериране на тестове

На сайта на състезанието имаше качени по-малко от 10 теста, като нито един от тях нямаше повече от 50 реда или колони. Освен това броят заявки за примерните тестове беше много по-малък от позволените 50 000. Затова написах кратък скрипт на **Ruby** (файлът *generate.rb*), който приема като аргументи брой редове, брой колони и брой заявки и генерира тест със съответните параметри. За думите използвах само съществителните генерирани от *ruby gem*-a *random-word*. Бях фиксирал количеството камъни на 1/20 от всички клетки в полето, понеже осъзнавах че няма да имам време да имплементирам и различни решения спрямо концентрацията на камъни (макар и на теория то може значително да повлияе резултатите). Понеже в условието нямаше споменато абсолютно нищо за генерирането на тестове, трябваше да направя поне някакви допускания иначе задачата става с твърде много степени на свобода.

### Алгоритми

Решението ми използва няколко подхода и в зависимост от входните параметри избира кой от тях да приложи. Тук ще опитам да опиша общите идеи, които използвам в някои от тях:

* 1. Премахване на камъни – преди да започна да се опитвам да слагам думи, премествам всички камъни в горния край на полето. Така няколко реда от горния край са изцяло заети от камъни (и евентуално един – частично), но долната част е изцяло свободна. Този подход използвам само когато броя заявки е сравнително голям спрямо размера на полето (по долу ще опиша по-точно кога);
  2. „Алчно” поставяне на дума – итерирам по цялото поле и избирам мястото, където може да се сложи думата така, че да **добавя** минимално количество букви. Ако една буква вече е била на полето, то не се брои за добавена;
  3. „Алчно” поставяне на определено **наредено** множество от думи. Опитвам се да поставя думите „алчно” в определения ред. Възможно е този ред да не е същия като реда, в който думите ще се появят. По този начин, може и да реша да прескоча дадена дума, въпреки че за нея има възможна позиция, за да мога да поставя някоя от следващите думи;
  4. Добавяне на дума чрез премахване на единствен камък. Подходът е същия като при алчното поставяне на дума, но позволявам да има единствен камък, който ми пречи. Ако има такъв камък и има свободна позиция на полето, където да го преместя, първо премествам камъка и после слагам думата на избраната позиция.

#### 2.1 Избор на решение

В избора на конкретно решение използвам няколко критерия, които се определят от входните данни. Общите означения които ползвам са *n* – брой редове, *m* – брой колони, *q* – брой позволени заявки, *total\_stones* – общ брой камъни на полето.

1. Един тест е „голям”, ако *n* \* *m* >= 14400 или *n* \* *m* \* *q* > 100 \* 100 \* 6000;
2. Ако един тест не е голям, но *n* \* *m* >= 5 \* *q* смятам, че имам относително малко заявки;
3. При *n* <= 25, *m* <= 25 и *q* <= 700 смятам, че тестът е достатъчно малък за решение, изпробващо всички пермутации (вж. решение 2.1.2 по-долу);
4. Ако *n* \* *m* - *total\_stones* <= (*q* - *total\_stones*) \* 3 смятам, че си струва първо да отместя всички камъни.

В зависимост от горните критерии прилагам различни алгоритми.

#### 2.1.1 solve\_complete

Използвам това решение, ако според горните критерии теста не е голям, но има относително малко заявки (точка 2 по-горе).

Това решение се състои от два етапа. В първия викам последователно *next\_word* докато не срещна първата дума, която не може да се постави на полето. След първата дума която няма къде да се разположи, започвам да викам *skip* всеки път когато не ми е известно коя е следващата дума. Когато ми е известно коя е следващата дума проверявам дали има къде да се постави и ако няма такава позиция на полето отново викам *skip*. По този начин си гарантирам, че никога няма да сложа повече от една дума неуспешно и съответно няма да имам отрицателни точки.

И в двата етапа на решението, когато проверявам може ли да се постави дума, обхождам всички възможни начални клетки и се опитвам да поставя думата както хоризонтално, така и вертикално. Освен това, ако има няколко позиции където мога да поставя думата избирам тази, при която поставянето на думата ще изисква добавянето на най-малко нови букви на полето. Този подход съответства на идея **b)** от описаните по-горе.

#### 2.1.2 solve\_complete\_all\_ways

Използвам това решение, ако не са изпълнени критериите за горния подход *solve\_complete* и теста е достатъчно малък за решение, изпробващо всички пермутации (точка 3. от критериите).

Това решение отново работи с думите, които още не сме поставили, но знаем, че са следващите, които ще ни върне *service*-a. Ако няма такива думи, вика *skip*.

След като вече има думи, за които знаем, че предстоят, решението изпробва всички пермутации на предстоящите думи и опитва да разположи последователно думите във всяка пермутация. Отново за всяка дума обхождам всички възможни позиции, като на всяка позиция опитвам да разположа думата както вертикално, така и хоризонтално. Ако има няколко възможни позиции за дадена дума избирам тази, която добавя най-малко нови букви на полето. Това поставяне отговаря на описаната по-горе идея **c)**.

Реализирал съм пробването за една пермутация на думи във функцията ***try\_to\_place***. Тя приема като аргументи списък с известните думи и още един списък, съдържащ пермутация на индексите на думите от първия списък. Тази функция връща структура ***solution***, в която се съдържа колко букви сме успели да поставим общо за дадената пермутация (включително тези, които вече са присъствали на дъската), колко са новопоставените букви (т.е. такива, които са поставени върху празно поле на дъската) и самите позиции на думите.

От всички възможни пермутации избирам тази, за която общия брой букви е най-голям, а ако има няколко такива - избирам тази, за която новодобавените букви са най-малко. След като избера най-добрата пермутация според горните критерии, разполагам думите според дадения *solution*. Ако дадена дума не присъства в дадения *solution*, то викам ***skip***, за да я пропусна. Ако думата присъства, поставям думата на позицията посочена от дадения *solution*.

Ако на първата итерация от цикъла по всички пермутации на известните думи открия, че **нито една** от известните думи не може да се разположи на полето, превключвам решението си на ***solve\_complete\_bigger\_first***. Правя това понеже, по моя преценка, е твърде възможно да не мога да поставя повечето от оставащите думи, а обхождането на всички пермутации е 120 пъти по-бавно от проверяването на единствена от тях.

#### 2.1.3 solve\_fast

Прилагам това решение, ако не са изпълнени критериите за горните решения - преценил съм че тестът е **голям** (точка 1 от критериите) и си струва първо да премахна всички камъни (точка 4 от критериите).

На първата стъпка от решението премествам всички камъни от полето в първите редове (идея **a)**), като поставям камъните от ляво надясно и от горе надолу на тези редове. Това е реализирано във функцията ***remove\_stones.***

След като преместя камъните, създавам масив, в който помня колко последователни свободни позиции има на всеки ред, броейки от ляво надясно. Съответно веднага след преместването на камъните тази стойност ще е 0 за първите няколко реда, евентуално по-малка от *m* за последния ред, който не е изцяло запълнен с камъни, и *m* за всички останали редове.

След това поставям думи последователно като отново, ако не ми е известно коя ще е следващата дума, викам ***skip***. Впоследствие търся ред, на който има поне толкова свободни позиции, колкото е дължината на думата. Ако намеря такъв ред, разполагам думата на него в най-лявата свободна позиция. Ако няма достатъчно свободни позиции на никой ред, отново викам ***skip.*** Ако с горния алгоритъм не успея да поставя повече от 10 последователни думи, преминавам към похват подобен на ***solve\_complete***. Забележете, че в първия си етап ***solve\_fast*** разполага думите само хоризонтално. Затова ако не мога да разположа повече от 10 думи последователно, започвам да итерирам по всички позиции на полето и за всяка позиция се опитвам да поставя следващата дума както хоризонтално, така и вертикално. Втория етап на решението е много по-бавен, но позволява поставянето на повече думи. След като го добавих, резултатите на този тип тестове се подобриха с 10-20%. Трябва също да се отбележи, че по време на втория етап, ако в даден момент решението ми е отнело повече от 9.5 секунди, то извиквам ***EndGame***, за да няма риск да надвиша *time limit*-а.

2.1.4 solve\_fast\_few\_queries

Използвам това решение, ако тестът не отговаря на критериите на никое от решенията по-горе - преценил съм че тестът е голям (критерий 1), но не си струва първо да премахна камъните (критерий 4). Това решение е подобно на ***solve\_fast***, но без да премахвам камъните в началото. По тази причина не мога да си гарантирам, че свободните позиции на всеки ред са единствена непрекъсната последователност и затова пазя списък с хоризонталните последователности от свободни клетки. В този списък пазя началните позиции на всички хоризонтални последователности от свободни клетки с дължина от 1 до *m*. Когато разполагам дума, търся последователност с дължина възможно най-близка до дължината на думата.

Като изключим търсенето на сводна позиция и факта, че не премествам камъните в началото, това решение е абсолютно същото като ***solve\_fast.*** Отново ако не успея да разположа повече от 10 думи поред, преминавам към решение подобно на описаното в ***solve\_complete***.

#### 2.1.5 solve\_complete\_bigger\_first

Използвам това решение, ако тестът не отговаря на критериите за никое от решенията по-горе и съм преценил, че не си струва първо да отместя всички камъни (критерий 4). Тук отново, подобно на ***solve\_complete\_all\_ways***, на всяка итерация ми е известна поне една дума. В противен случай викам ***skip***. При това решение сортирам множеството от известни думи по дължина в намаляващ ред и се опитвам да поставя само тази подредба на думите. Така, вместо с всичките 120 пермутации, които пробвам да поставя в ***solve\_complete\_all\_ways***, работя само с една. Макар и да е възможно така да пропусна най-доброто решение, печеля много в бързодействие.

Ако на дадена стъпка установя, че не съм могъл да разположа нито една от известните ми думи, изпълнявам процедурата описана в идея **d)**: тук опитвам да поставя следващата дума, която ми е известна, като първо преместя единствен камък. В действителност търся позиция където не мога да поставя думата заради единствен камък и се опитвам да преместя камъка на някоя друга свободна позиция. Това е реализирано във функцията ***try\_to\_fit\_removing\_stone.*** Ако благодарение на тази процедура успея да наместя поне една от известните думи, отново преминавам към нормалния подход в това решение.

#### 2.1.6 solve\_complete\_bigger\_first\_remove\_stones

Използвам това решение, ако не са изпълнени критериите за никое от по-горните решения и сметна, че си струва първо да отместя камъните (критерий 4). Отново изпълнявам функцията ***remove\_stones****,* а след това продължавам с решението ***solve\_complete\_bigger\_first.***

#### Обобщение

Като цяло това са шестте решения, които използвам. По време на състезанието написах още няколко подхода, които с течение на времето отхвърлих, понеже се представяха по-зле от по-нови решения.

За да формулирам критериите, описани по-горе, изпълних повече от 1000 теста, като се опитах да ги създам с разнообразни параметри, така че да покрия възможно най-много случаи. Решението ми се състои от близо 1400 реда код, без да включваме генерирането на тестове.