Интерактивни задачи

В този документ ще въведем основния знания за техническата работа по интерактивни задачи – какво представляват те и как компютърът симулира ограниченията, които са нужни за създаването на интерактивна задача.

Какво е "интерактивна задача"?

До група С и в по-голямата част от нея, срещаните задачи ефективно очакват от състезателя да създаде функция, на която се подават входни данни, която извършва някаква работа с ограничено време и ресурси, и която връща някакъв изход.

Интерактивната задача (interactive problem) вмъква идеята за "взаимодействие" между състезателя и журито. Най-честият вид задача в тази форма е, общо казано, следната "Журито има таен обект, до който допуска ограничен достъп на състезателя. Въпреки ограниченият достъп състезателят трябва да разбере достатъчно за този скрит обект, за да е способен да отговори на някакъв въпрос."

Веднага ще дадем основен пример за такава задача: Алис и Боб играят игра – Алис си е намислила число x в някакъв ограничен интервал $[l, r]$. Боб знае в какъв интервал се намира числото на Алис, но не знае самото число (Боб знае стойностите на l и r, но не и на x). Целта е Боб да разбере какво е числото на Алис, но единственият начин да извлича информация за него е да пита въпроси от типа "Алис, числото ти по-малко или равно ли е от y?", където y е число, което Боб си избира.

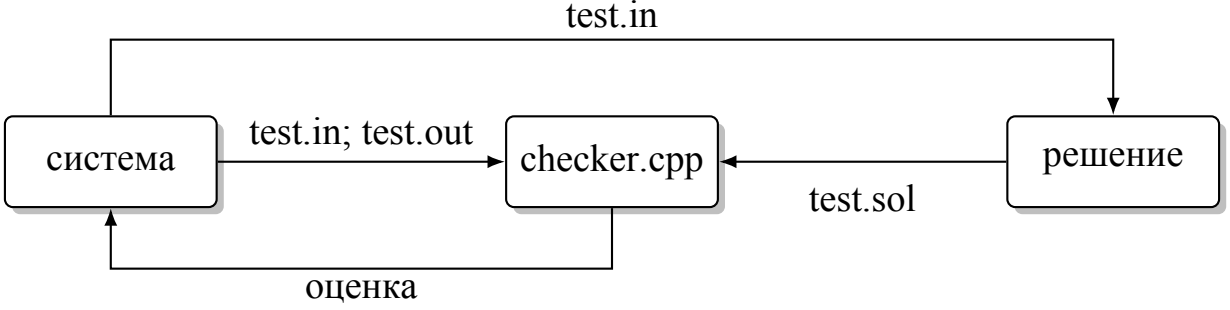
Дотук задачата е малко мудна, но не и невъзможна – Боб ще пита многократно въпроса си за стойностите $y = l, l + 1, l + 2, ... r$. Алис ще му отговаря "не, не, не, ...", докато в някакъв момент Боб не нацели числото и Алис не отговори "да".

Заради това сега ще покажем една още по-ограничена версия на играта, която все пак е решима – на Боб е позволено да пита $O(\log(r - l + 1))$ въпроса. Но и това не е особено по-трудно: както видяхме в миналия абзац, отговорите на Алис са "монотонни" и това позволява да изпълним двоично търсене върху тях, което ще пита точно нужният брой въпроси.

Нека се върнем към абстрактното описание на интерактивна задача и да видим как то съвпада с дадения пример: Журито в лицето на Алис имаше число, Боб имаше ограничен достъп до това число, защото Алис отговаряше само на определени въпроси относно него (и по-късно Боб имаше право на ограничен брой въпроси). Единственото лошо на този пример е, че целта ни беше да добием пълен достъп до числото – да го научим изцяло – а не да получим частичен достъп до него (например да разберем само (не)четно ли е), но това ще видим в по-късна задача.

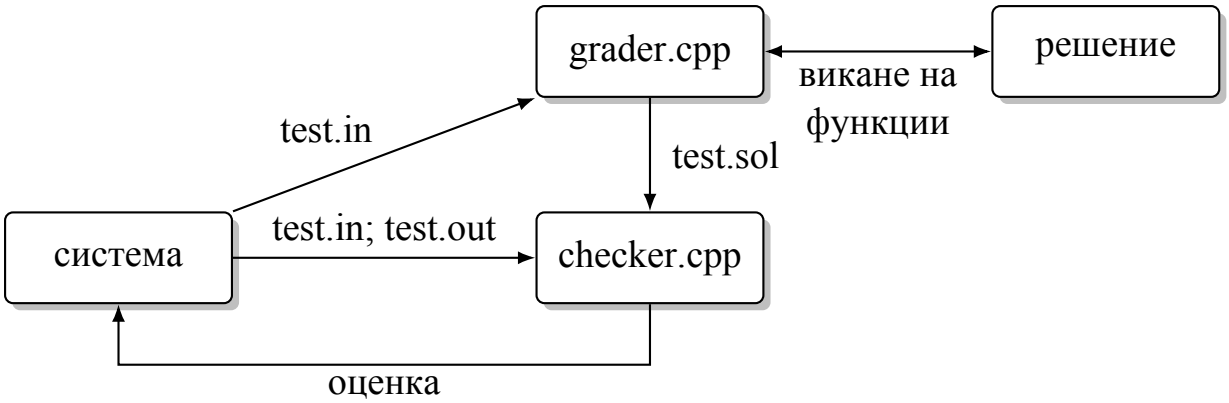
Как журито ни задава интерактивна задача?

В тази секция ще изясним техническите детайли относно работата по интерактивни задачи. Досега се очаква състезателите да имат сравнително ограничени умения за работа с компютъра, ако единствената им по-задълбочена работа с него е била да решават задачи от състезания.



Досега познатите задачи на състезателите са били обработвани от системата по следния начин: системата избира поредният тест, който трябва да се оцени и го подава на стандартния вход (`stdin`) на решението на участника. Решението на участника създава изход за този вход, който извежда на стандартния изход (`stdout`). От там то се подава на проверяваща програма (написана от автора на задачата), която оценява с колко точки трябва да се оцени този изход (например задачата иска да се изведат две числа, от които първото е за 30% от резултата, второто – за 70%, `checker.cpp` се грижи да направи това).

За интерактивните задачи поточната линия е малко по-сложна поради дребния детайл, че участникът в никакъв момент не трябва да получава по нерегламентиран начин достъп до скритията обект на журито. Този скрит обект реално се явява входните данни на текущия тест, който решаваме. Значи се оказва, че някак трябва да предоставим входните данни до решението, но не и изцяло – трябва по средата на комуникацията да има някакъв арбитър, който контролира достъпа на решението до входните данни. Тук добавяме в системата файл на име `grader.cpp`.



Системата подава входните данни на `grader.cpp`, който е отговорен за:

- прочитането на входните данни;
- дефинирането на ограничения достъп до тях: това се случва с функции (например `compare(x, y)` би била функция, която ни е предоставена, за да се допусне сравнение на две числа в скрит масив);
- извеждането на изход, който да бъде прочетен от `checker.cpp` и на база, на който да бъде оценена работата на участника.

Забележете, че единствената роля на решението на участника в тази диаграма е във викането на функции между него и `grader.cpp`. Заради това и не е нужно да се имплементира функция `int main()` – такава ще е имплементирана, така или иначе в `grader.cpp`, който играе ролята на входна точка за процеса по решаването на задачата. (В противен случай, ако имаше две дефиниции на `int main()`, компилаторът не би знаел коя от двете да изпълни първо).

Най-интересна от състезателна гледна точка е комуникацията чрез функции между решението на състезателя и `grader.cpp` – обикновено в условието на една задача ще описани няколко функции – някои, от които вие ще трябва да дефинирате, и някои, които са готови от журито за Ваша полза. Например в горния ни пример с Алис и Боб, ако Боб трябва да напише програма, която да даде на Алис, примерен "интерфейс" би бил – Боб ще дефинира функцията `int solve(int l, int r);`, която Алис ще извика, когато е готова да започне комуникацията. В тази функция Боб ще опише своя алгоритъм и когато има нужда да зададе въпрос на Алис би викал функция `bool question(int y);`.

Още един детайл, който озадачава много при първа работа с интерактивна задача е защо условието държи решението на състезателя да не извежда нищо на стандартния вход. Стандартният поток от данни за вход, изход и грешки (`stdin, stdout, stderr`) е запазен за кому-

никацията между системата, `grader.cpp` и `checker.cpp`, конкретно системата ще сложи на стандартния вход входните данни (примерно скритото число за този тест), а след края на взаимодействието `grader.cpp` ще изведе изхода, който ще се прочете от `checker.cpp`, който пък ще оцени решението и ще даде финалната оценка.

Забележете, че ако в някакъв момент състезателя има достъп до този канал, той може да се възползва, извличайки информация, която не трябва да знае или подавайки информация, която фактически не е достигната.

Реални случай са примерно следните:

- Състезател вика `fseek` в решението си, чрез което прочита стандартния вход втори път и разбира скритата пермутация на задача без да вика каквито и да е заявки.
- Състезател извежда текста `OK` от функцията си и веднага я спира. `checker.cpp` прочита това и решава, че за всеки тест състезателят е заслужил пълен брой точки – съответно решението се оценява със 100 точки.

Колкото и вълнуващо да звучи да научите за такива "хакерски" подходи за решаване на задачите, такива дупки вече са закърпени в системата – една опция е входните данни да се хешират, друга е `checker.cpp` да очаква парола, която само `grader.cpp`, а трета е да се добави в системата `manager.cpp`, който стриктно менижира за всяка програма до какво има достъп.

Как да тестваме интерактивна задача?

За да може състезателя удобно да тества решението си се предоставя минимизирана версия на горната система от файлове в така наречената `contestant/` папка. Най-значимо в нея се намира `Lgrader.cpp` или просто `grader.cpp` и допълнителен библиотечен файл, описващ всички функции, които журито или състезателя трябва да имплементират. За нашата примерна задача с Алис и Боб такъв файл би бил `game.h` и ще съдържа следните редове:

```
int solve(int l, int r);
bool question(int x);
```

За да пуснем програмата си и да я тестваме в реална ситуация, *ще трябва да компилираме две програми едновременно* – нашето решение, например `game.cpp` и програмата на журито `grader.cpp`. Най-просто това става чрез командата в терминала

```
$ g++ game.cpp grader.cpp -o game
```

Имайте напредвид, че за да пуснете тази команда в терминала трябва да имате инсталиран компилатора GCC на съответната ви операционна система (Windows или нещо Linux-базирано). Ако досега сте компилирали само с "натискането на бутон" в някое IDE (например Code::Blocks), реално сте изпълнявали такава команда.

В резултат от командата ще получите изпълнимия файл `game`, който представлява Вашето решение. При пускане, той ще очаква вход – скритият обект на конкретната задача – ще разиграе комуникацията и накрая ще върне някакъв изход. Взависимост как е написан `grader.cpp` може да се окаже, че локално програмата ви има достъп до входните данни – това на системата определено няма да е така. Това се получава, защото когато компилираме два файла едновременно те споделят ресурси и съответно памет помежду си – една функция, дефинирана в единия файл (например `solve` в `game.cpp`), може да се ползва от другия файл (например `grader.cpp`); същото се отнася и за променливи.

Заедно с лекцията е предоставен архивиран файл с програми `game.h`, `game.cpp`, `grader.cpp`. Поиграйте си с тях, за да придобиете усет как точно работят интерактивните задачи.

Някои примерни решения на интерактивни задачи

В остатъка от този документ ще се занимаваме с няколко интерактивни задачи, които пасват на дефинираното определение до тук. Забележете, че не обещаваме изчерпателно изреждане на всички идеи, които могат да се срещнат в такъв тип задачи.

Контрола младша група 2022 – present

Този път скритият обект ще е пермутация на числата от $1, 2, \ldots, n$, която ще трябва да научим, а протоколът за комуникация ще е функцията `question(x, k)`, която ни дава k-тото по големина число сред първите x числа в пермутацията. Забележете, че според условието и здравия разум трубва да е изпълнено $1 \leq k \leq x \leq n$, но такава проверка не е добавена в прикачения `grader.cpp` (поне по време на писането на този документ). Добро упражнение за читателя е да прецени къде трябва да я добави и да го направи сам.

Естеството на заявките, които можем да подаваме предполага да почнем да търсим числата от ляво надясно в пермутацията – първо да определим първото, второто, третото и така нататък. Допълнително, доста удобно е, че заявката `question(1, 1)` директно ни дава първото число. За да доразвием идеята нека разгледаме следната постановка: знаем стойностите на числата на позиции $p_1, p_2, \ldots, p_{x-1}$ и търсим да определим стойността на p_x.

Полезно е да разгледаме стойностите, които заемат извикванията на функцията `question` с първи параметър x или $x-1$ и всевъзможните стойности за втори параметър k – все пак те трябва някак да "кодират" долепянето на новия елемент към редицата ни. Например ако `question(x-1, 1)` и `question(x, 1)` имат равни стойности, това означава че с или без елемента p_x, най-голямата стойност в редицата е една и съща, а понеже p е пермутация това директно ни казва, че няма как p_x да е най-голямата стойност в $p_1, p_2, \ldots, p_x$. Същата логика всъщност можем да продължим за $k = 1, 2, \ldots, (x-1)$ и на всичко отгоре проверката `question(x-1, k) == question(x, k)` е монотонна с нарастването на k. Следователно можем да използваме двоично търсене, за да намерим стойността на p_x. Единствената уловка е, че може да се окаже, че за $1 \leq k \leq x-1$ тази проверка винаги е вярна, тоест p_x е най-малкият елемент в префикса от $x-1$ елемента, тоест стойността на p_x е `question(x,x)`.

IATI младша група 2018 – namuhs

Тук скритият обект е напълно произволен масив от цели числа, а целта ни не е някак да възстановим целият масив, а само да открием единственият подмасив с максимална сума на числата в него и да върнем двете му краища l и r. Комуникацията с журито е особена ограничена и особена – можем единствено да си избираме два подмасива на целия масив и да ги сравняваме (не примерно да извличаме информация за сумата на числата в един от тях). Допълнително ограничение е *сложността* на изпълнение на една заявка от програмата на журито, която е $O((r_1-l_1+1)+(r_2-l_2+1))$ – обикновено сложността на заявките ни към журито е минимална и почти незначителна, но тук наивната имплементация на функцията ще изисква допълнителна грижа от нас, за да имплементираме каквото и да е решение.

Привидно много неща се случват в задачата, така че ще работим по нея малките към големите случаи – понеже нямаме достъп до съдържанието на масива, единственият параметър, който диктува стратегията ни за решаването на задачата е дължината му. При $n = 1$ решението е тривиално. При $n = 2$ може би се струва да се замислим, но отново без директен достъп до стойностите в масива, нямаме по-добра опция от да разиграем всички възможни отговори, които са три на брой. За да представим по-ясно идеята ни ще прескочим $n = 3$ и директно ще погледнем в случая $n = 4$.

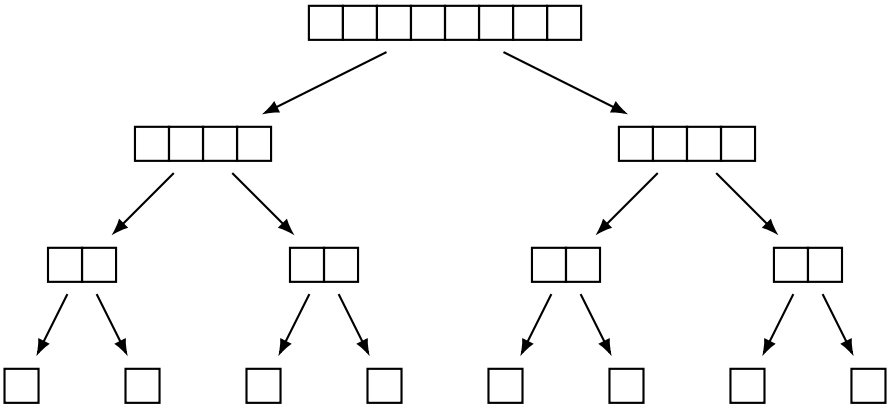
За $n = 4$ броя подмасиви е 10, което е, да кажем, сравнително много. При $n = 4$ обаче задачата е много близка до $n = 2$ всъщност – просто имаме две инстанции на $n = 2$ задача

долепени една до друга. Така отговорът в големия случай е един от следните:

- или отговорът за лявата половина на масива;
- или отговорът за дясната половина на масива;
- или някой напълно нов подмасив, който обаче преминава през "средата", която разделяй левия от десния подмасив.

Така всъщност в третия случай трябва да проверим дали суфиксът с най-голяма сума на лявата половина, долепен до префикса с най-голяма сума на дясната половина произвежда по-добър отговор от тези в първите два случая.

Спряхме да говорим за конкретната стойност на n, защото тя и не е нужна в решението ни вече – разработихме начин една голяма инстанция на задачата да се разбие на две по-малки и отговорите от тях да бъдат слети. Това, което получихме е именно решение от типа "разделяй и владей".



Защо това решение е ефективно? Стандартно при "разделяй и владей" решенията можем да си мислим за дървото на рекурсивните извиквания. На всяко ниво в него, обединението на подмасивите, за които извикванията на функцията в това ниво отговарят ще дава целия помасив. Но тези нива са приблизително $O(\log n)$ на брой. Следователно решението ни ще извърши сумарно $O(n \log n)$ работа с общо $O(n)$ извиквания на `compare` функцията.

Забележка

Миналите две задачи представиха две решения, съответно с двоично търсене и разделяй и владей. Значителна (но не цяла) част от интерактивните задачи допускат решения с някоя от тези техники, така че е добра идея да ги имаме впредвид, когато работим по интерактивна задача.

IATI младша група 2020 – biggest

Журито има скрита пермутация и ни подава параметър k. Нашата цел е да намерим позициите, на които са записани k-те най-големи числа в тази пермутация. За комуникация имаме право единствено да даваме два индекса в редицата и да получим като отговор за кой от двата записаното върху него число е по-голямо. Първите $n-1$ сравнения извършени в комуникацията са "безплатни" и не се причисляват в сметките, определящи финалния резултат на участника.

Ще използваме тази задача, за да представим една по-обща теория, значима за някои интерактивни задачи – модел на изчисление, основан на сравнения. В най-общи думи този модел на изчисление обхваща алгоритми, които единствено обработват паметта си сравнявайки я помежду си (сравнение на две числа в пермутацията). С така поставени ограничения се оказва, че можем да извадим някои интересни общи твърдения, не за един конкретен алгоритъм, а за всякакви алгоритми, работещи в този модел. Въпреки че някои от твърденията са доказани в последния линкнат файл, ще повторим някои от тях.

Твърдение 1.1: Намиране на максимален елемент

За да се намери максималният елемент в редица, ползвайки единствено сравнения между елементите, трябва да се извършат поне $n-1$ сравнения. С точно $n-1$ сравнения може да се състави алгоритъм, който намира максималния елемент.

Очевидно е, че $n-1$ сравнения могат да създадат нужния алгоритъм – обхождаме масива от ляво надясно и сравняваме най-новия елемент с най-големия досега открит.

Нека допуснем, обаче, че има алгоритъм A, който ползва под $n-1$ сравнения. Нека за даден вход I_1 между всеки два елемента, които са сравнени от A да нарисуваме ребро между тях. Знаем, че в този граф ще има n върха, но $n-2$ ребра, следователно задължително графа ще е несвързан.

Ще създадем нов вход за алгоритъма I_2, в който всички елементи от редицата, които не са в компонента на свързаност на максималния елемент (нека я кръстим C) в графа, ще бъдат увеличени с достатъчно голяма стойност, така че един от тях да стане по-голям от максималния елемент на I_1. Така сравненията между всяка двойка числа, които са в C остава същата, всяка двойка числа, които са извън C остава същата, но очевидно вече има нов максимален елемент.

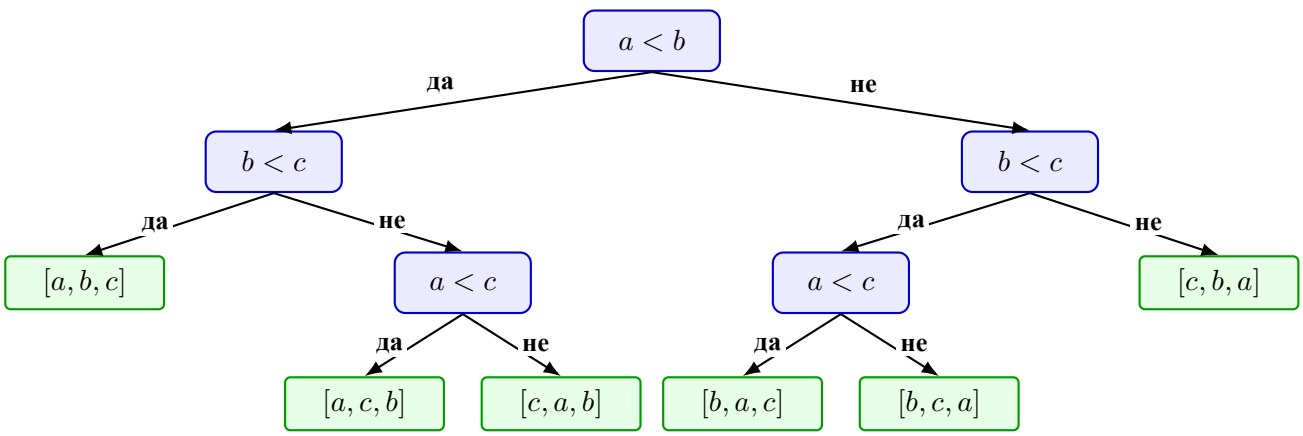
Алгоритъмът A обаче няма как да знае, че има нов максимален елемент, защото той никога няма да има причина да направи сравнение между елементи от I_2, които не са били сравнени при вход I_1. Все пак A е детерминистичен алгоритъм в модел на изчисление, основан на сравнения – всички "решения"са основани на предишните сравнения, а по дефиниция всички сравнения, които са правени върху I_1 ще останат същите върху I_2. Следователно има вход I_2, за който алгоритъмът ни греши, тоест нямаме алгоритъм, който за $< (n-1)$ сравнения решава проблема.

Забележете, че числото $n-1$ тук съвпада с броя заявки, които журито ни предоставя "безплатно". Преди да решим задачата обаче ще направим още едно (ненужно) доказателство, което ще ни даде малко "птичи поглед" над ситуацията, която ни предоставя проблема.

Твърдение 1.2: Долна граница на сравненията при сортиране

За да се сортира редица от различни числа с дължина n, трябва да се извършат *поне* $\Omega(n \log n)$ сравнения.

Забележете, че тази долна граница на броя сравнения е в сила, само когато достъпът до данните се осъществява единствено чрез сравнения – ако не е така, можем да направим сортиране чрез броене и да сортираме данните чрез 0 сравнения.



За да процедираме с доказателството ни ще разглеждаме така наречените "дървета на решенията" (decision tree). В случая всяко листо на дървото представлява един възможен изход (възможна сортираща пермутация) и всеки връх, който не е листо представлява взимането на едно решение (оценката на едно сравнение между две числа). Ще допуснем, че е очевидно, че едно балансирано двоично дърво (каквото в случая е нашето дърво на решенията) има височина $\log m$ за m – броя на листата му. В случая обаче $m = n!$ и височината на дървото е $\log n!$. Заради свойството на логаритмите, че $\log ab = \log a + \log b$ ще получим, че $\log n! = \log n + \log (n-1) + \cdots + \log 2 + \log 1 > (n/2) \log (n/2) = \Omega(n \log n)$.

Това всъщност представлява някакъв много частичен случай на нашата оригинална задача, където $k = n$. Добре образован състезател примерно би искал в такъв случай да се ориентира към решения със сложност $O(k \log n)$ за извличането на първите k най-големи числа. Но за да се стигне до там първо ще трябва да въведем нужната структура на иначе случайната ни пермутация.

Тук ще се върнем на проблема с първите $(n-1)$ сравнения, който правим безплатно. Предложенията сканиращ алгоритъм за намирането на максимума в първото твърдение тотално няма да ни е полезен, но пък не твърдим, че това е единственият алгоритъм, който намира максималния елемент за $(n-1)$ сравнения. Друг алторитъм би бил следния: за всяка двойка от типа $(2k, 2k+1)$ извършваме едно сравнение и намираме някакъв "локален максимум". След това, отново сдвоени намираме максимумите на блокчета от четири поредни позиции, после осем и така нататък.

По този начин ще създадем един "туринр" между числата, в който ще се изиграят общо $(n-1)$ игри (може да се убедите или като си разиграете един пример или като разпишете доказателството чрез силна индукция). На пръв поглед това естествено не върши много повече работа – пак ни намери максималния елемент, което е добре, но добавената сила на този метод е, че имаме много повече информация за сравненията между останалите числа.

След като сме намерили най-голямото число, сега ще искаме да намерим второто най-голямо число – кое е то? В така създадения турнир, второто най-голямо число ще е едно от числата, които директно е загубило от първото (ако не беше загубило от него, от кой друг да загуби), а такива числа има най-много $\log n$ на брой. Следователно, след като извлечем най-голямото число ще искаме да направим втори "мини-турнир" измежду числата, които са загубили директно от него и да ползваме резултати от този мини-турнир, за да допълним целия турнир.

Именно имплементацията на идеята от последния абзац различава високите резултати на задачата – ако започнем директно да си строим сегментно дърво, за което всеки връх е или най-големият елемент в съответния интервал, или някаква фиктивна стойност, ако всички елементи в интервала са вече извлечени, ще получим 96 точки.

Това решение все пак има някаква "статичност" (дори да сме махнали 100 числа, формата на сегментното дърво е същата като в началото, а това няма нужда да е така), от която бихме искали да се освободим, за да изкараме 100 точки. Ще извлечем най-полезните идеи на решението и ще работим само с тях.

На първо място, искаме да извличаме предварително информация за елементите, дори когато те не са сред "силните" победители в турнира. Нещо, което можем да правим, за да постигнем това е винаги да избираме двата елемента с най-малко победи и да правим сравнение помежду им. По този начин отново ще постигнем структура на двоично дърво, но вече сравненията, които сме изпълнили зависят по-малко от наредбата на числата в теста и повече от стойностите им. Ако докато изпълняваме едно сравнение поддържаме за всяко число списък кои други числа е надвило ще получим и сравнително бърз начин да "закърпим" дупките, които

получаваме, когато извадим най-голямото досега налично число.

Нещо подобно на идеите от тази задача може да видите в секция “3.2 Finding the second-largest of n elements” в статията за изчислимия модел.

Допълнителни задачи

- Индивидуално състезание НШИ 2026 cave
- Младен Манев 2019 A flowers
- Контрола младша група 2023 cells
- IATI 2022 B sorting

Автор: Иван Лупов