**Анализ**

*Тагове: ограничена памет, кодиране на информация, сортиране, сортиране с броене, пирамидално сортиране, сортиране със сливане*

Първо е важно състезателите да се ориентират какво се иска точно. Добрите решения изискват реализирането на сортиращи алгоритми, които е удобно да се разбият на много малки стъпки и да е възможно с много малко информация да се разбере в какъв етап е алгоритъмът. Тук се включват и всякакви дребни детайли, като броячи на цикли, индекси на масиви, памет в стека на рекурсията и т.н. Всичко трябва явно да се реализира и да се отдели съответния брой битове памет. Задачата в началото беше предвидена за IATI B, но се оказа, че когато се преследва най-добро представяне от гледна точка брой гетвания на паметта, доста се усложняват алгоритмите и трябва да се прилагат много трикове и идеи, за да се достигнат оптимални имплементация. Затова оставихме задачата за НОИ-3 АВ.

Начални идеи и трикове

Ще започнем с една част от предварителните идеи, които са съществени за повечето решения. Първото важно нещо, което трябва да се съобрази е, че клетките в паметта на Правеца са 32-битови, а стойностите и броят числа се побират в 16 бита най-много. Това е централно, за да може да се пишат решения, които не ползват допълнителната памет. Причината да се махат голяма част от точките при допълнителната памет, не е че предната идея е някакво сериозно наблюдение, а просто значително се усложнява реализацията, когато не ползваме допълнителните клетки. Освен това всяка допълнителна клетка реално има 32 свободни бита. Докато началните клетки, заради заетите битове от стойностите на масива реално имат само 16 свободни бита. Можем да кажем, че старшите 16 бита ще използваме като служебни в алгоритмите и там ще записваме нужната информация за текущото състояние, етап, броячи, индекси и т.н. на алгоритмите. За по-лесно ще наричаме младшите 16 бита на една клетка долен ред на клетката, а старшите 16 бита – горен ред.

Друг трик, който може да направим, е свързан с по-удобната реализация на самите алгоритми и по същество с него получаваме еквивалента на клетките от допълнителната памет само че в нашите и така почти директно можем алгоритъм, който ползва допълнителна памет, да превърнем в такъв, който не ползва. Идеята е да направим частично сортиране в началото. Лесно можем да сортираме и да сложим на местата първите 2, 3 и т.н. най-големи (или най-малки) елемента. Ако са достатъчно малко на брой тези елементи, можем с едно минаване през масива (разбира се разделено на стъпки, при които разглеждане само по един елемент на масива) да запишем в долните редове на крайните клетки най-големите стойности. Ако пък искаме да сортираме повече елементи в началото, то ще трябва да ги намираме един по един, т.е. с няколко минавания през масива. Това пък от своя страна налага повече етапи на алгоритъма и затова ползваме главно първия начин. След като сме направили този трик, вече долните редове на последните *x* клетки са каквито трябва да са в края при сортирането. Така долните редове не трябва да се променят повече и можем спокойно да ползваме горните им редове за служебна информация. За да получим пълния еквивалент на допълнителната памет (както и за да намалим броя гетвания) ще направим следното. Нека разгледаме клетки с индекси ***N***-1 и ***N***-2. В горния ред на клетка ***N***-2 ще сложим стойността от долния ред на клетката ***N***-1. Така вече клетка ***N***-1 може да се нулира и да получим 32 свободни бита! Това можем да направим по-общо като групираме по двойки сортираните *x* клетки и го приложим за всяка двойка. Единствено накрая трябва да не забравим да върнем обратно елементите на масива.

Едно важно нещо, което е хубаво да се осмисли, е, че в част от случаите просто бихме искали да прибавим или извадим число от някоя клетка. Обикновено, тук ни се налага да направим един гет за тази клетка, за да видим каква е стойността и да знаем колко ще е новата стойност за сетване.

Вече сме готови да пристъпим към описанието на алгоритмите. За жалост, тази задача няма едно решение за 100 точки, защото имаме реализация на сортиране с броене чрез най-много 3 гетвания за една стъпка, а от по-общите алгоритми имаме най-добри реализации на пирамидално сортиране и сортиране със сливане, които ползват най-много 4 гетвания. Когато човек пише решения на подобен тип задача, където оптималната версия е доста пипкава и с много детайли, е най-добре първо да напише по-лесни версии (в случая с допълнителна памет, след което малко по-малко да оптимизираме броя гетвания) и да не се хваща веднага с цялостното решение.

Решение на втора подзадача – 10 точки. Параметри: *c*=0,7; *num*=10

Тази подзадача е предвидена за добри реализации на всякакви квадратични сортирания. За да се вземе пълният брой точки, трябва да се направят добри реализации без допълнителна памет. Достатъчно е просто да използваме горния ред на две клетки, за да реализираме например сортиране чрез метода на мехурчето (трябват ни 16 бита, за брояча, който обхожда масива и 1 бит информация, за да знаем дали се е случила размяна на текущата итерация). Понеже *num*=10, няма какво особено да оптимизираме от към броя гетвания. Затова, за да разменим две елемента на масива, можем да ползваме горния ред на някоя свободна клетка като временна променлива. Накрая трябва да не забравим да нулираме горните редове на клетките, където пазим служебната информация. Очакваната сложност е .

Решение на трета и четвърта подзадача – 55 точки. Параметри: *c*=0,3; *num*=3

На пръв поглед решението тук може да изглежда по-сложно от това на пета подзадача, заради малкия *num*. Но тук имаме едно улеснение – стойностите на числата в масива са от 0 до ***N***-1. Това всъщност предразполага да мислим за решение на базата на сортиране с броене. То се оказва доста по-приятно и лесно за реализация от някой от общите алгоритми. Сравнително лесна версия и директна реализация на сортиране с броене е приложена във файла pravetz\_countSort\_17.13p. В това решение се използват две клетки от допълнителната памет за състояния и най-много 4 гетвания. Нека първо анализираме етапите на сортирането чрез броене. Имаме първи етап, в който обхождаме масива и построяваме count масива. За този етап ни трябват 16 бита – за текущия индекс в масива. Другият детайл тук е къде се намира count масива – най-лесно е в горните редове на първите ***N*** клетки. Вторият етап е по-сложен, защото обхождаме count масива и слагаме съответната стойност на мястото в масива, до което сме стигнали. Тук ни трябва 32 бита информация, едните 16 са за текущата стойност, а другите – за текущия индекс, до където сме сложили сортираните числа. Това всъщност са двете клетки от допълнителна памет. Логичен въпрос е не може ли да ползваме само една клетка за служебната информация. Проблемът е всъщност, че трябва да я ползваме и при двата етапа. Така че реално максимално ни трябват 32+1 бит информация за в кой етап сме и затова без да направим нещо по-умно ни трябват две клетки за служебната информация. Вече преминаваме към пълната версия.

Когато пробваме да махнем допълнителната памет, идва и един голям проблем с това, че при най-лошия случай за запазване на count масива трябва да използваме всичките горни редове на първите ***N*** клетки. За да имаме валидно решение искаме да имаме малко глътка въздух. Затова ще използваме хитрата идея, че може да пренебрегнем нулите. Първо, знаем, че те винаги ще са най-отпред при сортирането (ако ги има). Освен това тяхната бройка ще е ***N*** – останалото. Така с това изхитряване ще може да отпадне и нуждата от допълнителен бит, за да знаем в кой етап на алгоритъма сме. В това решение, винаги ще използваме нулевата клетка за цялата информация, тъй като ни трябват най-много 32 бита. Двата етапа изглеждат по следния начин:

* първи етап – ще използваме само битовете в долния ред на нулевата клетка, за да пазим индекса, до който сме стигнали, при обхождането на масива (стойността винаги ще е по-малка от 216); ще попълваме count масива само за стойностите по-големи от 0; за да можем да сме сложили нужните нули, то когато видим стойността на елемент от масива, ще нулираме долния ред преди да продължим; така цялата информация ще остане в count масива, но това е достатъчно
* втори етап – ще използваме горния ред на нулевата клетка за текущия индекс, а долния ред за текущата стойност; тук едната хитрост, която ще използваме е, че ще слагаме сортираните стойности в обратен ред – от голямата към малката и съответно ще обхождаме масива в обратен ред; така можем да се постараем винаги текущия индекс в масива да е ≥ 1 и по този начин стойността на нулевата клетка в този етап (за разлика от първия) винаги ще е ≥ 216, като реално това ще е различаването на двата етапа 😊; заради този трик се налага малко неприятни случаи накрая – малко преди да приключим сортирането трябва понякога да стоим на индекс 1, докато намерим каква ще е стойността за индекс 0, а в други случаи, когато сложим стойността на индекс 1 знаем, че същата стойност ще е и на индекс 0 и приключваме

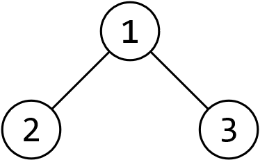
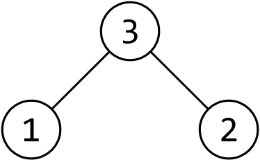
В предложената авторова реализация pravetz\_countSort\_55p вместо горен и долен ред се ползва частно и остатък при деление с ***N***. Реално така ползваме по-малко памет при по-малко на брой елементи, което е бонус, но разбира се не дава допълнителни точки. Сложността тук е .

Решение на пета подзадача – 88,75 точки. Параметри: *c*=0,3; *num*=4

Тук вече ще направим общ алгоритъм за сортиране. Приложените идеи стигат максимално до най-малко 4 гетвания и затова не могат да хванат пълни точки за трета и четвърта подзадача. Също ограничението за ***N*** тук е по-малко, за да няма проблеми с тайм лимита, защото решенията имат по-голяма константа и започват да работят доста по-бавно. Това е и причината да я има трета подзадача, за да е ясно, че такова решение ще хване по-голямата част от точките на задачата. Голяма част от общите сортиращи алгоритми имат проблеми в тази задача, защото искат доста допълнителна памет (или поне над 4 клетки на веднъж). Един от известните алгоритми с малка памет е бързото сортиране. Оказва се, че то се нуждае от логаритмична памет, защото в действителност рекурсията пази кои пивоти са избрани. Понеже имаме логаритъм нива, от там, за да го реализираме ни трябва логаритмична свободна памет. Така се налага да използваме един от началните трикове, за да я осигурим и като цяло ще имаме нужда от малко повече гетвания, както и константата ще се покачи доста, затова не е и писана такава реализация. При написано сортиране чрез броене, човек може да се замисли дали не може да се обобщи до поразрядно сортиране (radix sort). Но това, че трябва няколко пъти да правим сортиране с броене ни качва поне един гет, а и има и доста други проблеми. Оказва се, че най-подходящият алгоритъм тук е пирамидалното сортиране.

Реално пирамидалното сортиране по принцип не ползва допълнителна памет – при първия етап максималната пирамида се построява директно в масива, а след това при втория етап (самото сортиране) пирамидата се смалява, а сортираният масив нараства и всичко пак е на едно място. За нашата задача единствено ще ни трябват битове, в които да пазим етапите на алгоритъма, както и брояч на цикъла и позиция в пирамидата. Затова прилагаме сравнително директната имплементация в pravetz\_heapSort\_55p, която използва трика в първия етап да се сортират трите най-големи елемента и техните горни редове да бъдат за служебната информация. Това решение използва най-много 7 гета главно защото, за да се достъпи служебната информация се нуждаем от три гета, което както говорихме по-рано може да се оптимизира и до два. Използвано е само стандартната процедура *heapify\_down*, като там реално пада голямата трудност за последващото оптимизиране. Понеже не можем да я реализираме класически рекурсивно, то и нея трябва да разбием на стъпки. Заради това имаме като служебна информация и позиция в хийпа, която съответства на едно извикване на *heapify\_down*. Получаваме в най-лошия случай 7 гета, когато искаме да махнем максимума от пирамидата, защото тогава разменяме нулевата клетка с последната в пирамидата и правим една итерация на *heapify\_down* (тя коства 3 гета) и така за тази стъпка имаме 3+1+3 гета.

Едното място е да оптимизираме достъпванията за служебната памет, както казахме по-рано. Освен това в третия етап достъпваме от време на време нулевата клетка, затова можем в нея да пазим и етапа. Така реално отпадат 2 гета и стигаме до 5 общо, като приложена имплементация може да видите в pravetz\_heapSort\_73p. В тази реализация по хитър начин се използва само един бит информация, за да различим първия етап от другите два (същинските), а трети от втори различаваме с това, че всеки път слагаме допълнителен флаг в един бит, когато почваме *heapify\_down* в третия етап.

Сега ще опишем крайната версия – тя е във файла pravetz\_heapSort\_77.5p. Единственото място за оптимизация вече е *heapify\_down*, като трябва някак да направим стандартната процедура без нито веднъж да видим стойностите на върха и неговите деца едновременно. За тази цел може просто да разбием и това на подетапи – в първи подетап сравняваме бащата с лявото дете и ги разменяме ако има нужда, във втори подетап бащата с дясното дете и ги разменяме, ако има нужда, и след това евентуално продължаваме надолу (ако не е бил максимумът в бащата). Съответно трябва да кодираме нужната информация в бащата, за да знаем на кой подетап сме и какво да правим накрая, но това са само няколко бита, а върховете на пирамидата имат празни битове в горния ред (освен нулевата клетка). Има само един случай, в който този начин ще даде грешен резултат и той е показан на фигурата вдясно (началното и крайното състояние). Проблемът е, че при реалният *heapify\_down* единствено ще са се разменили 1 и 3, така че за да го оправим при нас трябва да разменим стойностите на двете деца. Но това пак не може да го направим директно. Можем да го направим чак следващия път, когато отидем в дясното дете, от където ще се извика следващия *heapify\_down*. Но тогава как знаем с кой връх да го разменим? Много просто, понеже само в този случай е проблемът, то знаем, че винаги сме в дясно дете и трябва да го разменим с лявото, а неговият индекс е с 1 по-малко! Това вече е достатъчно, за да направим *heapify\_down* с максимално 2 гета, както и целият алгоритъм вече с максимално 4 гета (реално са почти 3 гета, защото 1 гет е само за 1 бит информация – в кой етап сме).

Сега ще опишем оптимално решение (с максимално 4 гета), което реализира сортиране със сливане (pravetz\_rumen\_mergeSort\_88.75p). Тъй като ще ни трябва информация до къде сме стигнали в сортирането, ще използваме клетките с индекси 0 и 1 за тази цел, а числата от масива на позиции 0 и 1 ще изпратим на втория ред на клетките 2 и 3. Ще разделим алгоритъма на няколко фази, като в клетка 0 ще записваме на коя фаза се намираме:

1. [cases = 1] Сортиране на масива от клетка 4 до клетка ***N***-1.

В горния ред на клетка 0 освен фазата (cases), ще запазим още числата step и known, като step ни показва, че дължината на интервалите, които се опитваме да слеем е 2step. Known е 0 или 1 и ще видим за какво ни трябва по-нататък. Тъй като cases е най-много 3, за него са ни достатъчни 2 бита, а step е най-много 16, така че за него са ни достатъчни 5 бита, а за known ни е необходим точно 1 бит, така че виждаме, че има достатъчно място, където да ги запазим в горния ред на клетка 0.

В клетка 1 ще запазваме индексите на позициите, които се опитваме да сравним при текущото извикване на програмата (pos1 и pos2). Ако някое от тях е равно на ***N***, то значи съответният интервал вече е приключил и от сливането остава да допишем числата само от другия интервал.

Ако step е четно, числата ще са ни записани в долния ред (както е в началото), а при сливането на текущите интервали ще записваме числата в новата подредба съответно на втория ред. Ако пък step е нечетно, ще ги взимаме от втория ред и ще ги записваме на първия.

Да отбележим, че ако знаем колко е дължината на интервалите, които сливаме и позициите, на които са числата, които сега сравняваме то можем лесно да намерим позицията, на която трябва да запишем по-малкото от двете числа, така че няма нужда и това да пазим. Нека тази позиция я означим с pos.

Ако някое от pos1 или pos2 е ***N*** (БОО да смятаме, че е pos1), то остава да прочетем числото на позиция pos2, числото на позиция pos и съответно да запишем на правилния ред в клетката pos числото *a*pos2. Имаме get съответно за клетки 0, 1, pos2 и pos, така че с 4 get се справяме с този случай. Увеличаваме стойността на pos2 с 1, като внимаваме дали не е било край на текущия интервал. Записваме новите step, pos1, pos2 в клетките 0 и 1 и приключваме.

Aко и в двата текущи интервала има числа, то трябва да подходим по-внимателно, защото ако прочетем и двете числа, заедно с клетките 0, 1 и pos бихме получили 5 get. Тук идва и употребата на known и свободното пространство в долния ред на клетка 0. Сравняването на числата на позиции pos1 и pos2 и записването на по-малкото ще извършим на две извиквания на нашата функция. На първата, ще прочетем числото на позиция pos2 и ще го запишем в долния ред на клетка 0, като отбелязваме, че сме го направили чрез known ← 1. Във втората, когато пък known е вече 1, ще прочетем числото на позиция  pos1 и ще вземем числото от позиция pos2 от долния ред на клетка 0. Отново прочитаме числото на позиция pos и записваме по-малкото от текущите две в съответния ред, след което записваме новите step, pos1, pos2 и known в клетки 0 и 1.

1. [cases = 2] и [cases = 3] Вече клетките 4… ***N***-1 са сортирани

В тези фази, това което трябва да направим е да запишем сортираните числа в клетки 4… ***N***-1 в долния ред, а в горният да запишем 0. Всъщност използваме cases = 2 ако числата са ни в долния ред и cases = 3, ако са съответно в горния.

Това можем да направим по много начини, като в предложената реализация в долния ред на клетка 0 запазваме до коя позиция сме стигнали и на всяко викане на функцията изчистваме една клетка.

1. [cases = 0] Справяне с първите 4 числа от масива. Вече сортирали голямата част от масива, поставянето на правилните места на тези 4 числа не би трябвало да е особено сложно. В предложената имплементация се сортират чрез bubble sort, като се използват горните редове на клетки 0 и 1, за да се знае кое от тези 4 числа се подрежда в момента и до коя позиция е стигнал bubble sort-а, като преди това “компресираните” числа в клетки 2 и 3  се връщат в долните редове на клетки 0 и 1. Веднъж приключило сортирането и за 4тото число, изчистваме горния ред на клетки 0 и 1 и сме готови 😊

Второто решение има по-малка константа и затова хваща всички предвидени точки, докато решението с пирамидално сортиране има по-голяма константа и трябва да се направи малко по-добра реализация (с използване на повече побитови операции), за да хване същите точки (без да тайм лимитва на последните групи на 4-та подзадача). Сложността на тези решения е .

Остава отворен въпроса има ли общ сортиращ алгоритъм, който може да се реализира с най-много 3 гета. Също е интересно формално да се докаже, че това е теоретичния минимум за тази задача. Лесно се вижда, че ни трябват поне 2 гета – достъпването на клетка със състояние и достъпване на друга клетка, за да видим каква стойност има и евентуално да я променим.

*Анализ: Румен Михов и Илиян Йорданов  
Автор: Илиян Йорданов*