Въведение

Нека първо сметнем колко заявки са ни нужни. Информацията в една пермутация е логаритъм от броя различни пермутации. Т.е. информацията която трябва да получим е $\log_2 N!$ бита, а максималната информация, която можем да извлечем на заявка е $\log_2 K!$ бита. Следва, че теоретичният лимит е:

$$\log_2 N! / \log_2 K! = \ln N! / \ln K! \approx N \ln N / (K \ln K) = N \log_K N / K$$

Ако искаме да сме изцяло точни, можем да използваме пълната апроксимация на Stirling, което е направено, когато се казва, че авторовото решение използва около 2 пъти повече заявки от теоретичния лимит. Използвайки формулата горе виждаме, че за $K = 1000$ авторовото решение използва около 60 сравнения.

От тук нататък ще разгледаме решения базирани на 3 различни базови идеи и как те могат да се развият за пълни или почти пълни точки.

Наивно решение със std::sort

Всъщност преди да започнем с трите основни подхода, нека първо разгледаме едно наивно решение. Всеки читав състезател трябва да успее да хване тези ~10 т. Решението е просто да използваме std::sort като подадем за функция за сравнение да се ползва наша такава, която слага двата елемента във вектор вика compare и така връща резултат.

Quick sort

Първият подход, който може да ни хрумне, е quick sort. Ако напишем просто стандартен quick sort, изкарваме ~10 т. – приблизително същото като с std::sort. Можем лесно да добавим по-ранно спиране на рекурсията като се достигнат под K елемента – тази оптимизация ще наричаме bottom (защото се случва на долните нива) и в случая ни изкарва ~11 т.

Multibranch quick sort

До тук обаче не използваме многостранното сравнение освен в листата на рекурсията. Един начин да ги ползваме е на всяка итерация на рекурсията да разделяме масива не на 2 части, а на K части – това ще наричаме multibranch. Това става като изберем $K - 1$ pivot-а и всеки елемент сравняваме с всички от тях. След това търсим къде е новия елемент или линейно (бавно, но минава в time limit-а) или с двоично търсене.

Това решение използва $O(N)$ сравнения на едно ниво от рекурсията, но прави $O(\log_K N)$ нива. С други думи използва $O(N \log_K N)$ сравнения, което е доста повече от теоретичната граница. Затова то изкарва само ~21 т.

Multiway quick sort

Друга относително проста идея е да разделяме масива на 2 части както нормално, но да обработваме по $K - 1$ елемента наведнъж. Т.е. фиксираме си един произволен pivot и всеки път го сравняваме с $K - 1$ елемента. Тези по-малки от него слагаме в левия подмасив, а по-големите от него – в десния.

Това решение очевидно се държи точно както нормален quick sort, но използва около K пъти по-малко сравнения, т.е. $O(N \ln N / K)$, което е доста по-добре от предното решение. Така това решение изкарва ~53 т.

Multiway quick sort с умен pivot

Можем да подобрим предното решение, като вместо да фиксираме pivot-а произволно, го изберем след първата заявка (т.е. средния елемент там). Това помага двете части, на които разделяме, да са по-близки по размер, което значително намалява броя нужни сравнения. Това изкарва ~65 т.

Multiway multibranch quick sort

Multibranch дели броя сравнения на $\ln K$, multiway – на K. Всъщност обаче е добра идея да използваме и двата подхода заедно, за да постигнем желаната сложност. Наивно можем да решим да имаме $K/2$ pivot-а и да обработваме по $K/2$ елемента наведнъж, но след малко мислене (или пробване) очевидно не е оптимално. По принцип можем да сметнем оптималния брой pivot-и, ако приемем, че разделянията са абсолютно равни и игнорираме частични използвания на сравнения и още 1-2 такива практически неща. На практика обаче е добре да пробваме ръчно за няколко минути. След като пробваме можем просто да използваме $S = 4$ или $S = 5$ (S е броя pivot-и), като само за малки K нагласим по-малки стойности за S.

Това решение изкарва решение изкарва ~74 т. с произволни pivot-и, ~80 т. с умни pivot-и (взети на равни интервали от първото сравнение) и ~87 т. с умни pivot-и повече неинтуитивно нагласяне на S спрямо K.

Multiway multibranch quick sort с DP

След като имаме това решение остава само да измислим по-добър начин да определяме S. На практика оптималната му стойност очевидно зависи и от N по няколко различни причини. Затова е добре да направим едно dp по N, което да смята колко е оптималната бройка pivot-и спрямо N. За целта смятам очаквания брой сравнения, ако при дадено N използваме дадено S за всяко разумно S и запазваме оптималното както и самата бройка, която следващите итерации за dp-то ще ползват.

Очаквания брой сравнения е равен на броя сравнения за текущото разделяне $(N - S)/(K - S)$ плюс броя сравнения за сортиране на солните нива. Можем да приемем, че отделните $S + 1$ разделени масива ще са с еднаква дължина т.е. това да е просто $(S + 1) \times dp[(N - S)/(S + 1)]$. Всъщност е добре вместо да закръгляме дължините им надолу (т.е. да си представяме, че тези елементи изчезват) да има някаква бройка закръглени надолу и някаква нагоре, така че сумата им да е $N - S$. Това решение вече има оптималната сложност по брой сравнения и изкарва ~95 т.

Multiway multibranch quick sort с Monte Carlo DP

Проблемът на предното решение е, че приема, че отделните подмасиви са перфектно разделени. Това не е толкова лоша апроксимация когато използваме умни pivot-и, но все пак не е оптимално. Вместо това можем произволно да генерираме K различни числа до N, да изберем pivot-ите, както бихме, когато реално сортираме, и после да видим какви размери масиви сме получили. Би било най-добре да повторим това няколко пъти, но поради time limit-а няма време. Това изкарва ~100 т.

Една оптимизация е да поддържаме един random sample докато вървим през dp-то като само произволно го ъпдейтваме с шанс $1/N$ да включим новия елемент като заменим стар. Това е много по-бързо, но малко по-лошо, защото sample-ите не са независими за различните N. Това може да се компенсира като поддържаме няколко успоредни sample-а. В крайна сметка е по-консистентно от другото и се използва за авторовото решение.

Merge sort

Друг подход, който може да ни хрумне, е merge sort. Ако напишем просто стандартен merge sort, изкарваме ~12 т., защото merge sort обикновено прави по-малко сравнения от quick sort. Естествено отново е добра идея да добавим bottom оптимизацията.

Multiway merge sort

Както с quick sort, така и тук можем да правим и multibranch, и multiway (или и двете) и се постигат подобни резултати. В приложените решения е имплементиран само multiway mergesort, който изкарва ~62 т.

Единственото различно от стандартен merge sort е merge-ването. Вземаме по $K/2$ елемента от двете половини и можем да сме сигурни, че те са следващите, които да добавим към общата редица, до последната редица от елементи само от еднаква половина. Например, ако елементите след сортиране са АББББААБАБАБББААБАА (А са елементи от първата половина, а Б от втората), знаем, че АББББААБАБАБББААБ със сигурност са следващите в общата редица, но за последните АА не знаем, защото може следващото неразгледано Б да е преди (или между) тях.

Multiway merge sort с подравняване

Една лека оптимизация за няколко точки е да си използваме листата на рекурсията по-ефикасно, т.е. да разделим на chunk-ове от по K, които директно да сортираме и после тях да merge-ваме. Най-удобно е това да се прави директно в рекурсията като просто подравняваме точката на разделяне към най-близкото кратно на K. Това ни носи ~65 т.

Topological sort

Последната разгледана идея е topological sort. Разглеждаме елементите като върхове, а резултатите от сравнения като ребра между поредните от тях. Ако някак магически сме питали нужните заявки, после лесно можем да открием отговора като направим топологично сортиране на този граф.

Предимството на този подход се вижда лесно: той използва и пази цялата получена информация от сравненията. Merge sort използва заявки само за merge-ване, но той вече знае реда на елементите от конкретна половина едни спрямо други, т.е. това не е нова информация. Quick sort пък след като раздели елементите в подмасиви забравя за техния ред едни спрямо други, т.е. хаби тази информация. От друга страна топологичното сортиране (по предназначение) спазва транзитивността на ребрата, т.е. знае, че ако $a < b$ и $b < c$, то $a < c$ без ръчно да добавяме и такова ребро в графа.

Остава да видим как обаче да питаме заявки. Можем да пробваме много произволни заявки и след това да сортираме, а ако не стане да повторим. Това обаче не е твърде добра идея, защото скоро след началото повечето заявки няма да са смислени.

Вместо това започваме топологичното сортиране и когато следващият елемент не може да се определи еднозначно (т.е. има над един елемент в опашката), започваме да добавяме всеки следващ елемент в списък за сравнение. Когато този списък стигне дължина K, пускаме заявка за сравнение и започваме отначало. Тук проблемът е, че топологичното сортиране е със сложност по време $O(N)$ и ще пуснем толкова топологични сортирания, колкото заявки ползваме. Това е твърде бавно за малки K и там ще използваме std::sort, колкото да имаме някакви точки. Това решение взема ~67 т.

Topological sort с rollback

За да се оправим с описания горе проблем, ще имплементираме stack, в който пазим всички промени направени по състоянието на програмата на всяка стъпка от топологичното сортиране. Също така в началото на всяка стъпка, когато е еднозначно как да продължим, записваме маркер, че това е еднозначно положение в stack-а. Когато открием многозначност и започнем да добавяме в списъка за сравнение, спираме да поставяме такъв маркер. Накрая след сравнението една по една връщаме всяка от направените промени, като ги pop-ваме от stack-а, докато не стигнем до маркер за еднозначно състояние. По този начин не повтаряме цялото топологично сортиране на всяка стъпка. Това все още е неефикасно (както по време така и по заявки) при $K = 2$ и затова там използваме std::sort.

Това решение изкарва ~97 т., което е почти максимума. Причината да не е много по добро от последния quick sort (а всъщност малко по-лошо) е, че докато запазва и използва цялата получена информация, не е оптимално в избирането на заявки за сравнение. Тъй като се базира на топологично сортиране, във всяка стъпка трябва да открие минималния следващ елемент, за да продължи правилно, т.е. това е един сложен selection sort, но докато правим самия selection, откриваме и доста друга информация, която по-късно ще бъде използвана.