Задача 3. Максимальная сумма непрерывного участка

Дана числовая последовательность из N элементов ($1 \le N \le 10\,000$). Требуется выбрать из нее любое количество чисел, идущих подряд, без пропусков. Какую максимальную сумму можно при этом получить?

Рассмотрим простой пример, поясняющий условие. Пусть задана такая последовательность:

8	-2	12

Если бы не требование брать числа без пропусков, мы выбрали бы только положительные элементы и получили бы сумму 20. Но, поскольку пропуски делать нельзя, мы включаем в сумму всю последовательность и получаем результат 18. А вот если бы вместо –2 было бы –20, наибольшая сумма была бы равна 12 и получалась бы из одного последнего элемента.

Решение 0, или Почти лирическое отступление

Я очень люблю эту задачу и часто даю ее в качестве входного теста, когда начинаю работать с новой группой учеников, уже знакомых с основами программирования. Предложенные решения помогают мне многое понять о реальных программистских знаниях и навыках людей, с которыми предстоит работать.

Чаще всего предлагают нечто похожее на приведенные ниже решения 1 и 2, иногда — довольно запутанные (и, как правило, ошибочные) программы, основанные на анализе многочисленных частных случаев взаимного расположения положительных и отрицательных элементов.

Но один раз мне предложили совершенно неожиданное решение, которое в какой-то степени подтолкнуло меня к написанию этой работы. Автор решения предлагал перебрать все подмножества заданной последовательности, для каждого из них определить, представляет ли оно непрерывный участок, если да — подсчитать сумму элементов и выбрать из этих сумм наибольшую.

Если вы уже прочитали разбор задачи о построении возрастающего ряда, то понимаете, что предложенное решение имеет сложность не ниже $O\left(2^N\right)$ и не дает шансов на получение реального результата уже при N порядка нескольких десятков.

Интересно, что это решение, в отличие от многих других, было предложено мне в виде общего описания, но не было доведено до программы. Думаю, это неудивительно: если квалификация программиста достаточна для организации перебора подмножеств, он обычно способен придумать для данной задачи более эффективный (хотя далеко не всегда самый эффективный) алгоритм.

Решение 1

Переберем все возможные непрерывные участки исходной последовательности. Сделать это несложно: будем по очереди делать все элементы начальными, а для каждого начала перебирать все возможные концы, то есть элементы, идущие в исходной последовательности после начала. Для каждой пары "начало—конец" подсчитаем сумму элементов и выберем из этих сумм наибольшую.

Возможно, данный алгоритм проще запрограммировать, чем описать словами, поэтому приведем основной фрагмент соответствующей программы:

```
maxSum := a[1];
for start := 1 to N do begin
   for finish := start to N do begin
      sum := 0;
      for i := start to finish do begin
        sum := sum + a[i]                          {***}
      end;
      if sum > maxSum then maxSum := sum
   end
end;
```

Оценим эффективность этой программы. Очевидно, что самое "узкое" место здесь — вычисление суммы. Строчка, отмеченная звездочками, находится внутри трех вложенных циклов. Первый цикл (`start`) выполняется N раз. Второй цикл (`finish`) выполняется `N+1-start` раз для каждого `start`, третий — (`i`) — `finish-start+1` раз для каждой комбинации `start` и `finish`. С ростом N все эти величины растут линейно, значит, количество выполнений строки {***} растет как $O(N^3)$, а алгоритм имеет кубическую сложность.

Подсчитаем количество сложений более строго. Когда `finish=start`, третий цикл исполняется 1 раз, при `finish=start+1` — 2 раза и т.д. При `start=1` третий цикл последовательно исполняется 1, 2, ..., N раз, при `start=2` — 1, 2, ..., $N-1$ раз и т.д. При `start=N-1` третий цикл исполняется 1 и 2 раза, при `start=N` – 1 раз.

Общее количество исполнений третьего цикла (обозначим его S):

$$
\begin{aligned}
S = {} & 1 + 2 + \ldots + N\text{-}1 + N \\
& + 1 + 2 + \ldots + N\text{-}1 + \\
& + \ldots + \\
& + 1 + 2 + \\
& + 1
\end{aligned}
\qquad (1)
$$

Каждая строчка (1) — сумма арифметической прогрессии. Подсчитав эти суммы по известной формуле, получим

$$S = \frac{N(N+1)}{2} + \frac{(N-1)N}{2} + \cdots + \frac{2 \cdot 3}{2} + \frac{1 \cdot 2}{2} \qquad (2)$$

Теперь просуммируем (1) по столбцам: сложим N единиц, N–1 двоек и т.д.:

$$\begin{aligned} S &= N\cdot 1+(N-1)\cdot 2+\ldots+(N-(N-2))\cdot(N-1)+(N-(N-1))\cdot N= \\ &= N+2N-1\cdot 2+\ldots+(N-1)N-(N-2)(N-1)+N\cdot N-(N-1)N= \\ &= N(1+2+\ldots(N-1)+N)-(1\cdot 2+\ldots+(N-2)(N-1)+(N-1)N)= \\ &= N\frac{N(N+1)}{2}-(1\cdot 2+\ldots+(N-2)(N-1)+(N-1)N) \end{aligned} \tag{3}$$

Сопоставляя (2) и (3), получаем:

$$S=\frac{N^2(N+1)}{2}-(2S-N(N+1))$$

Решая это уравнение относительно S, находим

$$S=\frac{N(N+1)(N+2)}{6}$$

Как и следовало ожидать, точный результат соответствует полученной "на глаз" кубической оценке вычислительной сложности алгоритма.

Оценим время выполнения. Предположим, что в секунду можно выполнить миллион сложений, а остальные операции вообще не занимают времени. Тогда за секунду можно решить задачу для N = 180, за минуту — для N = 710, за час — для N = 2783. При N = 10 000 потребуется около двух суток счета.

Конечно, оценка эта приблизительная и оптимистичная, но она дает хорошее представление о реальной эффективности кубического алгоритма. В отличие от экспоненциального, кубический алгоритм позволяет решить задачу за реально достижимое время. Однако время это достаточно велико и с ростом N растет сравнительно быстро. При увеличении N в 10 раз необходимое время увеличивается примерно в 1000 раз. Конечно, это много, но по сравнению с экспоненциальным алгоритмом, где увеличение N ***на*** какую-то величину приводило к росту времени ***в*** разы, выигрыш не просто значителен, это качественно другая ситуация.

На практике подобный алгоритм можно применять в ситуации, когда задачу требуется решить один раз (например, при проектировании каких-то систем), но для систем реального времени, в которых нужно быстро принимать решения, он уже не годится.

Решение 2

Усовершенствовать кубический алгоритм из предыдущего решения помогает простая идея: при фиксированном положении начала непрерывного участка подсчет суммы можно вести одновременно с перебором концов.

Главный фрагмент программы при этом выглядит так:

```
maxSum := a[1];
for start := 1 to N do begin
   sum := 0;
   for finish := start to N do begin
      sum := sum + a[finish];          {***}
      if sum > maxSum then maxSum := sum
   end
end;
```

Здесь основная операция (она опять отмечена звездочками) находится в двух, а не в трех вложенных циклах. Точный подсчет количества выполнений этой операции совсем несложен. Внутренний цикл выполняется N раз при `start = 1`, $N-1$ раз при `start = 2` и т.д. до единственного выполнения при `start = N`.

Общее количество выполнений

$$S = N + (N-1) + \ldots + 1 = \frac{N(N+1)}{2}$$

Вычислительная сложность этого алгоритма — $O(N^2)$, это *квадратичный* алгоритм. При уже привычном оптимистическом предположении о миллионе сложений в секунду за одну секунду можно решить задачу для $N = 1413$, а для $N = 10\,000$ потребуется примерно 50 секунд.

Конечно, в реальных условиях время уходит на все действия, а не только на сложение, да и миллион сложений в секунду обеспечит далеко не всякий компьютер, но тем не менее вывод очевиден: данное решение позволяет получить результат за действительно приемлемое время.

Квадратичные алгоритмы могут эффективно применяться в реальных системах. Они работают быстро, 10-кратное увеличение N приводит к росту времени примерно в 100 раз, и во многих случаях это вполне допустимо. Однако при действительно больших значениях N квадратичный алгоритм может работать довольно долго.

Решение 3

По-настоящему эффективное решение этой задачи можно получить, применив технику построения однопроходных алгоритмов.

Предположим, что мы обработали $k-1$ элементов исходной последовательности и нашли максимально возможную сумму для этого участка S_{k-1}. Добавим к последовательности один элемент a_k и посмотрим, как он может повлиять на максимальную сумму. Очевидно, возможны два случая: максимальная сумма либо останется прежней, либо увеличится. Если сумма будет увеличена, то новая сумма будет получена суммированием участка, заканчивающегося элементом a_k (в противном случае эта сумма была бы получена на одном из предыдущих шагов). Пусть T_k — наибольшая из

всех сумм участков, заканчивающихся элементом a_k. Если бы эта величина была нам известна, мы легко нашли бы максимальную сумму всех участков в пределах первых k элементов:

$$S_k = \max\left(S_{k-1}, T_k\right)$$

Попробуем, используя тот же прием, вычислить T_k. Один из участков, заканчивающихся элементом a_k, состоит из единственного элемента a_k, а все остальные включают элемент a_{k-1}. Сумма первого участка, очевидно, равна a_k, а наибольшая сумма среди остальных получается прибавлением a_k к T_{k-1} — наибольшей сумме среди участков, заканчивающихся элементом a_{k-1}. Следовательно,

$$T_k = \max\left(a_k, a_k + T_{k-1}\right)$$

Таким образом, зная значения T — наибольшей суммы, включающей последний элемент последовательности, и S — общей наибольшей суммы, мы можем найти новые значения этих величин после добавления к последовательности одного элемента. В тривиальном случае, когда в последовательности всего один элемент, имеем $S_1 = T_1 = a_1$.

Добавляя по одному элементу и применяя полученные рекуррентные отношения, получим требуемый ответ. Вот соответствующий фрагмент программы:

```
maxSum := a[1]; sum := a[1];
for i := 2 to N do begin
   if sum < 0
      then sum := a[i]
      else sum := sum + a[i];
   if sum > maxSum then maxSum := sum
end;
```

Полученное решение линейно. При росте N количество действий и время счета растут пропорционально N: если размер последовательности увеличится в 10 раз, время тоже увеличится десятикратно. При традиционных оптимистических предположениях на решение задачи при N = 10 000 одной секунды хватит с большим запасом.

Всегда ли существует эффективный алгоритм?

Рассмотренные примеры могут навести на мысль, что действительно эффективный алгоритм можно придумать для любой задачи. К сожалению, это не так.

Рассмотрим еще одну задачу. Дан набор из N чисел и некоторое число S. Необходимо выбрать некоторые числа из набора так, чтобы их сумма оказалась в точности равной S.

Очевидная идея — применить тот же метод, что и в первом решении задачи о построении возрастающего ряда. Будем перебирать все подмножества и считать их суммы. Если среди них окажется нужная, ответ будет найден, и, если нам повезет, это может случится достаточно быстро. Однако в худшем случае требуемую сумму получить не удастся и, чтобы убедиться в этом, придется перебрать все подмножества.

Итак, в худшем случае придется перебрать 2^N подмножеств. Таким образом, алгоритм относится к экспоненциальным, а значит, практически неприемлем, так как позволяет гарантированно получить ответ только при сравнительно небольших значениях N.

Если все заданные числа положительны, можно применить перебор с возвратом и отсечь подмножества с заведомо слишком большой суммой, но мы уже знаем, что этот метод приводит хотя и к более быстрому, но тоже к экспоненциальному алгоритму.

Можно ли как-то улучшить ситуацию? Этот с виду простой вопрос представляет собой одну из главных проблем современной информатики. Для многих задач (сегодня их насчитывается несколько тысяч, и список регулярно пополняется) известны экспоненциальные алгоритмы решения, а полиномиальные (то есть имеющие сложность $O\left(N^K\right)$ для какого-нибудь, пусть даже большого K) неизвестны. Такие задачи называют *NP-полными.* Задача о нахождении точной суммы относится именно к этому классу. Полиномиальный алгоритм ее решения неизвестен, но нет и доказательства, что этот алгоритм не существует.

Таким образом, для всех *NP*-полных задач известны точные алгоритмы, но они практически неприемлемы. Многие из этих задач имеют вполне реальное практическое значение, они часто возникают, например, при экономических расчетах. Точное решение оказывается недостижимым, поэтому задачу приходится упрощать, рассматривать отдельные частные случаи, искать приближенные решения и т.д.

Возвращаясь к рассуждениям об эффективности и правильности, мы вынуждены сделать печальный вывод: иногда ради эффективности приходится жертвовать правильностью. Лучше получить за приемлемое время хоть какое-то решение, чем вечно ждать абсолютно точного. Например, в задаче о построении заданной суммы можно искать не абсолютно точную сумму, а достаточно близкую к ней. Построение быстрых алгоритмов поиска приближенных решений — важная часть современной информатики. Надо только всегда ясно понимать, какое решение мы ищем: точное или приближенное.

Всегда ли существует алгоритм?

Итак, мы выяснили, что для некоторых задач существуют точные, но неприемлемые на практике алгоритмы.

Оказывается, существуют и такие задачи, для которых в принципе нельзя составить алгоритм решения, и эту невозможность можно строго математически доказать.

Рассмотрим в качестве примера задачу, близкую всем программистам, даже начинающим. Известно, что некоторые программы обладают очень неприятным свойством — они зацикливаются, то есть работают бесконечно долго (если, конечно, их не прервать). Причем одна и та же программа может нормально завершаться при одних входных данных и зацикливаться при других. И если запущенная программа надолго замолчала, мы часто не знаем, в чем дело: идет нормальный процесс или произошло зацикливание.

Для решения этой проблемы предлагается разработать специальную программу. Эта программа (назовем ее программа P) должна получать на входе текст некоторой программы Q и набор входных данных D. Результатом работы программы P должно быть сообщение о том, что произойдет, если запустить программу Q с данными D: нормальное завершение или зацикливание.

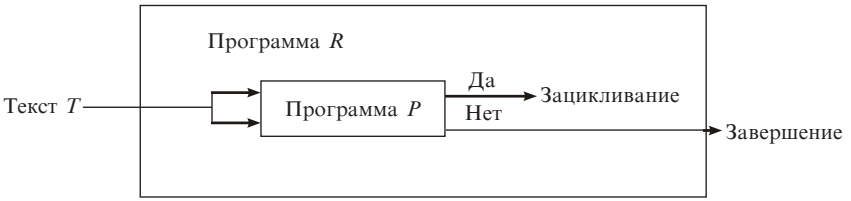
Графически работу программы P можно представить так:

То же самое можно описать и формулой:

$$P(Q,D) = \begin{cases} \text{да, если программа } Q \text{ с данными } D \text{ завершается} \\ \text{нет, если программа } Q \text{ с данными } D \text{ зацикливается} \end{cases}$$

Предположим, нам удалось успешно разработать программу P. Построим на ее основе программу R. Программа R будет получать на входе некоторый текст, делать его копию, а затем вызывать программу P, подавая ей одну копию исходного текста в качестве программы, а другую — в качестве данных. Если в результате работы P получится ответ “да”, программа R должна зациклиться (мы ведь умеем нарочно написать бесконечный цикл, правда?). Если же в результате работы P получится “нет”, программа R должна завершиться.

Вот графическое представление программы R:



В виде алгоритма программу $R(T)$ можно описать так:

```
если P(T,T) = да
   то     R(T) зацикливается
   иначе  R(T) завершается
все
```

Посмотрим теперь, что произойдет при попытке выполнить $R(R)$, то есть при подаче на вход программы R текста самой R.

Программа R вызовет программу P для ответа на вопрос: что произойдет, если вызвать программу R с данными R? Если P ответит “да”, R зациклится; если P ответит “нет”, R завершится.

Что же получается? Если по мнению программы P программа R с данными R зацикливается, то программа R с данными R благополучно завершается! Если же по мнению P программа R с данными R завершается, то в реальности программа R с данными R зацикливается!

Получили противоречие. Поскольку все рассуждения построены на основе предположения о том, что нам удалось написать программу P, это предположение следует считать неверным.

Следовательно, разработать программу для решения задачи об останове и зацикливании невозможно в принципе. Эта задача относится к классу алгоритмически неразрешимых.

К счастью, алгоритмически неразрешимые задачи встречаются в реальной жизни довольно редко. Но знать об их существовании надо. Хотя бы для того, чтобы еще раз убедиться: не все в этом мире можно запрограммировать.