Лабораторная работа 8. Рекурсивные функции

Цели и задачи работы: изучение рекурсивного программирования, методов разработки эффективных алгоритмов.

Методика выполнения работы:

1) Изучить технологию использования стека при рекурсивном программировании.

2) Написать рекурсивную программу решения поставленной задачи.

3) Протестировать программу.

Требования к отчету: отчет по лабораторной работе должен содержать титульный лист, задание, исходный текст программы, тесты.

Теоретическая часть

Как можно понятно объяснить, что такое рекурсия 4-летнему ребенку? Это довольно известный вопрос для собеседований, в интернете можно найти множество вариантов ответов. Если бы было необходимо объяснить рекурсию четырехлетнему ребенку, сначала следовало бы объяснить ее кому-то на год младше себя, он пускай бы объяснил кому-то на год младше его самого. И пускай бы все объясняли рекурсию по цепочке, пока 5-летний не объяснил бы ее 4-летнему. Готово.

Что касается программирования, рекурсия – это приём программирования, полезный в ситуациях, когда задача может быть естественно разделена на несколько аналогичных, но более простых задач. В процессе выполнения задачи в теле функции могут быть вызваны другие функции для выполнения подзадач. Частный случай подвызова – когда функция вызывает сама себя. Это как раз и называется рекурсией. Например:

```
void some_function(int n) {
   if (n != 0) return some_function(n - 1);
}
```

Приведенная выше функция сама по себе бесполезна, однако демонстрирует понятие рекурсии. Выполнение вызова random_function(n) не сможет продолжиться, пока не завершится вызов random_function(n-1), и так далее. По сути, мы откладываем выполнение текущего состояния функции, пока другой вызов той же функции не завершится и не вернет результат. Компилятор сохраняет состояние вызова функции, а затем переходит к следующему вызову функции и так далее. Состояния функции компилятор сохраняет в стеке и использует для вычислений и обратного отслеживания.

Если задачу можно разделить на похожие подзадачи, которые можно решить по отдельности и скомбинировать их решения для получения решения всей задачи, мы можем утверждать, что для данной задачи может существовать рекурсивное решение. При рекурсии метод решает небольшую часть задачи, разбивает задачу на меньшие и вызывает сам себя для решения каждой из этих подзадач. Обычно рекурсию применяют, когда небольшую часть задачи легко решить, а саму задачу просто разложить на составные части.

Рекурсия не часто необходима, но при аккуратном использовании она позволяет создавать элегантные решения, как в этом примере, где алгоритм сортировки иллюстрирует отличное применение рекурсии:

Пример алгоритма сортировки, использующего рекурсию:

```
void QuickSort( int firstIndex, int lastIndex, String [] names ) {
    if ( lastIndex > firstIndex ) {
            int midPoint = Partition( firstIndex, lastIndex, names );
           QuickSort( firstIndex, midPoint1, names );
            QuickSort( midPoint+1, lastIndex, names )
    }
}
```

В этом фрагменте алгоритм сортировки разрезает массив на две части и затем вызывает сам себя для сортировки каждой половины массива. Когда ему будет передан участок массива, слишком короткий для сортировки, т. е. когда (lastIndex <= firstIndex), он перестанет вызывать сам себя.

Для малой группы задач рекурсия позволяет создать простые, элегантные решения. Для несколько большей группы задач она позволяет создать простые, элегантные, трудные для понимания решения. Для большинства задач она создает исключительно запутанные решения — в таких случаях использование простых итераций обычно более понятно. Поэтому применять рекурсию нужно выборочно.

Ярким примером рекурсии являются числа Фибоначчи. Числа Фибоначчи {0; 1; 1; 2; 3; 5; 8; 13; 21; 34;…} известны с XII века и способ их получения имеет вид:

$$F_n = \begin{cases} 0, \text{если } n = 0, \\ 1, \text{если } n = 1, \\ F_{n-1} + F_{n-2}, \text{если } n > 1. \end{cases}$$

Это рекуррентный способ записи последовательностей. Часто последовательности задаются только таким способом. Такой алгоритм добывания элементов последовательности Фибоначчи очень легко запрограммировать:

```
int fibo(int n) {
        if (n == 0) return 0;
        if (n == 1) return 1;
        return fibo(n-1) + fibo(n-2);
}
```

Корректен ли этот алгоритм? Да, мы просто реализовали его определение. Каково время его работы? Этот вопрос сложнее.

Посмотрим на дерево вызовов этой функции (рисунок 1):

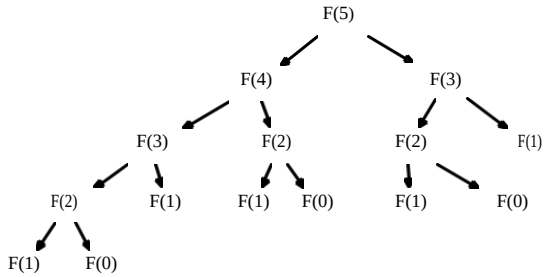


Рисунок 1 - Дерево вызовов для F(5)

Попытаемся оценить количество вызовов этой функции. Количество вызовов функции для $n = 0$ и $n = 1$ равно одному. Обозначив количество вызовов через *t(n)*, получаем *t(0) > F(0) и t(1) = F(1). Для n > 1 t(n) = t(n -1) + t(n - 2) > F(n))*. Следовательно, при рекурсивной реализации алгоритма количество вызовов превосходит число Фибоначчи для соответствующего *n*.

Сами же числа Фибоначчи удовлетворяют отношению

$$\lim_{n\to\infty}\left(\frac{F_n}{F_{n-1}}\right) = \Phi$$

где $\Phi = \frac{\sqrt{5}+1}{2}$, то есть, $F_n \sim C\Phi^n$, последний сомножитель и определяет сложность алгоритма.

Как же так, алгоритм прост, но почему так медленно исполняется? Проблема в том, что мы много раз повторно вычисляем значение функции от одних и тех же аргументов.

Требуемая для исполнения память характеризует сложность алгоритма по памяти.

- Каждый вызов функции создаёт новый контекст функции или фрейм вызова.
- Каждый фрейм вызова содержит все аргументы, локальные переменные и служебную информацию.
- Максимальное количество фреймов, которое создаётся, равно глубине рекурсии.
- Сложность алгоритма по занимаемой памяти равна *O(N).*

Третий вопрос: можно ли ускорить алгоритм? Да, и достаточно просто. Если мы введём добавочный массив для сохранения предыдущих значений функции, дело пойдёт на лад и уже вычисленные значения функции не будут вычисляться повторно:

```
int fibo(int n) {
        const int MAXN = 1000;
        static int c[MAXN];
        if (n == 0) return 0;
        if (n == 1) return 1;
        if (c[n] > 0) return c[n];
        return c[n] = fibo(n-1) + fibo(n-2);
}
```

Алгоритм остаётся рекурсивным, но дерево рекурсии становится вырожденным (рисунок 2).

Рисунок 2 - Вырожденное дерево вызовов для F(5)

Обратите внимание на серьёзную проблему: пока мы реализовали только алгоритмическую часть решения нашей проблемы, то есть установили верный порядок исполнения операций. А вот с данными всё плохо. Значение функции растёт слишком быстро и уже при небольших значениях *n* число выйдет за пределы разрядной сетки.

Мы здесь столкнулись с тем, что в реальных программах, исполняющихся на реальных компьютерах, существуют ограничения на операнды машинных команд. Их размер зависит и от архитектуры компьютера, и от исполняющейся операционной

системы. Самые популярные пока 32-битные операционные системы стремительно уступают место 64-битным. 32-битная архитектура на Intel-совместимых компьютерах называется X86 или IA32. 64-битных архитектур две, одна уже малопопулярна, разработка её прекращена это архитектура IA64, процессоры Itanium. Вторая, как ни странно, была разработана конкурентом Intel, компанией AMD, и она так и называется AMD64 (правда, Intel называет её EMT64), другое её название X64. В языках Си и C++ на архитектурах X86 и X64 стандартный тип данных int занимает ровно 32 бита. Чтобы воспользоваться 64-разрядной арифметикой, достаточно использовать тип данных long long или int64_t. Математики полагают, что основной единицей данных для вычисления является int, и оценивают сложность алгоритмов, исходя из этого.

Как вам уже известно, программа на C/C++ в процессе компиляции преобразуется в машинный код, понятный процессору. Каждая команда машинного кода располагается в памяти и имеет свой адрес. Команды располагаются последовательно (друг за другом). При вызове подпрограммы приложение запоминает адрес команды вызова – адрес возврата. Чтобы после выполнения подпрограммы продолжить выполнение (вернуться) с того места, где был произведен вызов функции. После вызова управление программой передается подпрограмме по адресу ее машинных команд. Также при вызове подпрограммы в нее могут передаваться аргументы (параметры). Передаваемые значения в подпрограмму приложение тоже запоминает. Адрес возврата, передаваемые аргументы, служебная информация вызова и локальные переменные подпрограммы называются фреймом активации. Фрейм активации записывается в программный стек (или просто «стек») – особую область памяти, организованную по типу стек. При выполнении подпрограмма считывает аргументы из программного стека и продолжает выполнение кода. Поэтому не имеет значения, в каком месте происходит вызов подпрограммы. В программном стеке всегда хранится информация о том, с какого места программа должна продолжить выполнение после подпрограммы. При запуске бесконечной рекурсии из-за ограниченности памяти и поэтому фиксированного размера стека может возникнуть ситуация, когда программа попытается внести в него данные, а свободной области не останется. В этом случае возникает ошибка stack overflow (с англ. «стек переполнен»), и произойдет экстренное завершение работы приложения.

Рекомендации по использованию рекурсии

1. Убедитесь, что рекурсия остановится. Проверьте метод, чтобы убедиться, что он включает нерекурсивный путь. Обычно это значит, что в методе присутствует проверка, останавливающая дальнейшую рекурсию, когда в ней отпадает необходимость.

2. Предотвращайте бесконечную рекурсию с помощью счетчиков безопасности. Счетчиком должна быть такая переменная, которая не будет создаваться при каждом вызове метода.

3. Ограничьте рекурсию одним методом. Циклическая рекурсия (A вызывает B вызывает C вызывает A) представляет опасность, потому что ее сложно обнаружить. Осмысление рекурсии в одном методе — достаточно трудоемкое занятие, а понимание рекурсии, охватывающей несколько методов, — это уже слишком. Если возникла циклическая рекурсия, обычно можно так перепроектировать методы, что рекурсия будет ограничена одним из них. Если это не получается, а вы все равно считаете, что рекурсия — это лучший подход, используйте счетчики безопасности в качестве страховки.

4. Следите за стеком. При использовании рекурсии вы точно не знаете, сколько места в стеке будет занимать ваша программа. Кроме того, тяжело заранее предсказать, как будет вести себя программа во время выполнения. Однако вы можете предпринять некоторые усилия для контроля ее поведения.

Во-первых, если вы добавили счетчик безопасности, одним из принципов установки его предела является возможный размер используемого стека. Сделайте этот предел достаточно низким, чтобы предотвратить переполнение стека.

Во-вторых, следите за выделением памяти для локальных переменных в рекурсивных функциях, особенно если эти переменные достаточно объемны. Иначе говоря, лучше использовать new для создания объектов в куче, чем позволять компилятору генерировать автоматические объекты в стеке.

5. Не используйте рекурсию для факториалов. Одна из проблем с учебниками по вычислительной технике в том, что они предлагают спорные примеры рекурсии. Типичными примерами являются вычисление факториала. Рекурсия — мощный инструмент, и очень странно использовать ее в случае факториала.

> Если бы программист, работающий у меня, применял рекурсию для вычисления факториала, я бы нанял кого-то другого.
>
> Стив Макконелл

Задание 1

1. Дано число m. Требуется в последовательности цифр 1 2 3 4 … 9 расставить знаки «+» и «-» так, чтобы значением получившегося выражения было число m. Например, m=122. Знаки расставляются как 12+34-5-6+78+9.

2. В аудитории стоят M компьютеров, на жестком диске каждого из которых имеется некоторое известное количество свободной памяти. Можно ли инсталлировать на них заданное число N программных систем, если каждая система требует для размещения не менее S[j] (j=1,…,N) памяти.

3. Имеются N двузначных чисел. Можно ли их соединить знаками сложения и умножения, чтобы получить заданное число S?

4. На веревке висят прямоугольные скатерти и салфетки так, что они занимают всю веревку. Один предмет был украден. Можно ли перевесить оставшиеся предметы так, чтобы веревка снова стала занятой полностью? Длина веревки, количество предметов и размеры каждого предмета известны.

5. Имеется N контейнеров высоты H. Задано множество предметов, каждый из которых имеет свою высоту. Можно ли разместить предметы в этих контейнерах так, чтобы груз не выступал над контейнером?

6. Для заданного набора слов требуется построить линейный кроссворд. Если окончание одного слова совпадает с началом следующего более чем в одной букве (например, ЛОГИКА-КАСКАД), то такие слова можно объединить в цепочку. Нужно получить любую такую цепочку.

7. Для заданного набора слов требуется построить линейный кроссворд минимальной длины. Если окончание одного слова совпадает с началом следующего более чем в одной букве (например, ЛОГИКА-КАСКАД), то такие слова можно объединить в цепочку, представляющую собой линейный кроссворд.

8. Разместить на шахматной доске максимальное количество коней так, чтобы они не находились друг у друга «под боем».

9. Задан набор слов. Построить из них любую цепочку таким образом, чтобы символ в начале следующего совпадал с одним из символов в середине предыдущего (не первым и не последним).

10. Задан массив целых чисел. Построить из них любую последовательность таким образом, чтобы последняя цифра предыдущего числа совпадала с первой цифрой следующего.

11. Задача раскраски карты. Страны на карте заданы матрицей смежности. Если страны i,j имеют на карте общую границу, то элемент матрицы A[i,j] равен 1, иначе 0.

Смежные страны не должны иметь одинакового цвета. «Раскрасить» карту минимальным количеством цветов.

12. Задача проведения границы на карте («создание военных блоков»). Страны на карте заданы матрицей смежности. Если страны i,j имеют на карте общую границу, то элемент матрицы A[i,j] равен 1, иначе 0. Необходимо разбить страны на две группы так, чтобы количество пар смежных стран из противоположных групп было минимальным.

13. В последовательности символов могут встречаться только цифры и знаки "+", при этом последовательность представляет собой формулу сложения однозначных чисел. Напишите рекурсивную функцию, определяющую значение формулы.

14. Напишите рекурсивную программу для формирования разметки линейки. При нанесении делений на линейку на ней должна быть метка в точке ½ длины шкалы, метка немного покороче через каждые ¼ длины, еще более короткая через 1/8 длины и так далее. Количество разбиений – степень двойки - задается.

15. Имеется N домов и K красок. Проверить, можно ли покрасить дома имеющимся набором красок, чтобы цвет повторялся не раньше, чем через 2 дома.

16. В группе из N студентов каждый студент назвал трех человек, с которыми хотел бы делать лабораторную в паре. Составить пары из группы, если это возможно.

17. Требуется найти минимальный путь обхода всего множества городов, не посещая один и тот же город дважды (задача коммивояжера).

18. Имеется M типов материалов и K типов клеев, каждый из которых пригоден для склеивания нескольких типов материалов. Требуется попарно склеить P (P <= M) известных типов материалов. Найти минимальное число клеев, которые потребуются для этого.

19. Напишите рекурсивную функцию, проверяющую правильность расстановки скобок в строке. При правильной расстановке выполняются условия:

(а) количество открывающих и закрывающих скобок равно.

(б) внутри любой пары открывающая – соответствующая закрывающая скобка, скобки расставлены правильно.

Примеры неправильной расстановки:)(, ())(, ())(() и т.п.

20. В строке могут присутствовать скобки как круглые, так и квадратные скобки. Каждой открывающей скобке соответствует закрывающая того же типа (круглой – круглая, квадратной- квадратная). Напишите рекурсивную функцию, проверяющую правильность расстановки скобок в этом случае. Пример неправильной расстановки: ([ )].

21. Число правильных скобочных структур длины 6 равно 5: ()()(), (())(),)(()), ((())), (()()). Напишите рекурсивную программу генерации всех правильных скобочных структур

длины 2n. Указание: правильная скобочная структура минимальной длины «()». Структуры большей длины получаются из структур меньшей длины, двумя способами:

(а) если меньшую структуру взять в скобки,

(б) если две меньших структуры записать последовательно.

Задание 2

Вариант 1

Имея входную строку (s) и шаблон (p), реализуйте сопоставление шаблонов с подстановочными знаками с поддержкой '?' и '*' где: '?' соответствует любому одиночному символу, '*' соответствует любой последовательности символов (включая пустую последовательность). Сопоставление должно охватывать **всю** входную строку (не частичную).

Ограничения:

- 0 <= s.length, p.length <= 2000.
- s содержит только строчные английские буквы.
- P содержит только строчные латинские буквы '?' или '*'.

Пример 1	Пример 2	Пример 3
Ввод: s = "aa", p = "a" **Вывод:** false **Объяснение:** "a" не соответствует всей строке "aa".	**Ввод:** s = "aa", p = "*" **Вывод:** true **Объяснение:** '*' соответствует любой последовательности.	**Ввод:** s = "cb", p = "?a" **Вывод:** false **Объяснение:** '?' соответствует 'c', но вторая буква - 'a', которая не соответствует 'ь'.

Вариант 2

Вам дано положительное целое число primeFactors. Требуется построить натуральное число n, удовлетворяющее следующим условиям:

- Количество простых множителей n (не обязательно различных) **не превышает** primeFactors.
- Обратите внимание, что делитель n хорош**,** если он делится на каждый простой множитель числа n. Например, если n = 12, то его простые множители равны [2,2,3], тогда 6 и 12 хорошие делители, а 3 и 4 нет.

Определите *количество хороших делителей* n. Обратите внимание, что простое число — это большее натуральное число, чем 1, которое не является произведением двух меньших натуральных чисел. Простые множители числа n — это список простых чисел, произведение которых равно n.

Ограничения: 1 <= primeFactors <= 10^9.

Пример 1	**Пример 2**
Ввод: PrimeFactors = 5 **Вывод:** 6 **Объяснение:** 200 — допустимое значение n. У него 5 простых делителей: [2,2,2,5,5] и 6 хороших делителей: [10,20,40,50,100,200]. Не существует другого значения n, которое имеет не более 5 простых множителей и больше хороших делителей.	**Ввод:** PrimeFactors = 8 **Вывод:** 18

Вариант 3

По входным числам n и k, вернуть последовательность перестановок. Набор [1, 2, 3, ..., n] содержит в общей сложности n! уникальных перестановок. Перечислив и пометив все перестановки по порядку, мы получим следующую последовательность чисел n_k для n = 3:

- $n_1 =$ "123";
- $n_2 =$ "132";
- $n_3 =$ "213";
- $n_4 =$ "231";
- $n_5 =$ "312";
- $n_6 =$ "321".

Ограничения:

- 1 <= n <= 9.
- 1 <= k <= n!

Пример 1	**Пример 2**	**Пример 3**
Ввод: n = 3, k = 3 **Выход:** «213»	**Ввод:** n = 4, k = 9 **Выход:** «2314»	**Ввод:** n = 3, k = 1 **Выход:** «123»

Вариант 4

Алиса и Боб соревнуются в стрельбе из лука. На соревнованиях установлены следующие правила:

1. Сначала Алиса стреляет numArrows стрел, а затем Боб numArrows стрел.
2. Затем баллы рассчитываются следующим образом:
    1. Цель имеет целочисленные части подсчета очков в диапазоне от 0 до 11 **включительно**.
    2. Для **каждой** части мишени со счетом k (между 0 и 11) предполагается, что Алиса и Боб выпустили a_k and b_k стрел в этой части соответственно. Если $a_k \geq b_k$, то Алиса получает k очков. Если $a_k < b_k$, то Боб получает k очков.
    3. Однако если $a_k == b_k == 0$, то **никто не** получает k баллов.

Например, если Алиса и Боб оба выпустили 2 стрелы в секцию со счетом 11, то Алиса получает 11 очков. С другой стороны, если Алиса выпустила 0 стрел в секцию с 11 очками, а Боб выстрелил 2 стрелы в ту же секцию, то Боб получает 11 очков.

Вам дано целое число numArrows и целочисленный массив aliceArrows размера 12, который представляет собой количество стрел, выпущенных Алисой на каждом участке подсчета очков от 0 до 11. Теперь Боб хочет **максимизировать** общее количество очков, которое он может получить. Возвратите *массив,* bobArrows *который представляет количество стрел, выпущенных Бобом на* ***каждом*** *участке подсчета очков от* 0 *до* 11. Сумма значений bobArrows должна равняться numArrows. Если у Боба есть несколько способов заработать максимальное количество очков, верните **любой** из них.

Пример 1

Scoring Section	0	1	2	3	4	5	6	7	8	9	10	11
Alice's Arrows	1	1	-	1	-	-	2	1	-	1	2	-
Bob's Arrows	-	-	-	-	1	1	-	-	1	2	3	1
Points Obtained by Bob	-	-	-	-	4	5	-	-	8	9	10	11

Ввод:

numArrows = 9,

aliceArrows = [1,1,0,1,0,0,2,1,0,1,2,0]

Вывод: [0,0,0,0,1,1,0, 0,1,2,3,1]

Объяснение:

В приведенной выше таблице показано, как оценивается соревнование.

Боб зарабатывает общее количество очков 4 + 5 + 8 + 9 + 10 + 11 = 47.

Можно показать, что Боб не может получить оценку выше 47 баллов.

Пример 2

Scoring Section	0	1	2	3	4	5	6	7	8	9	10	11
Alice's Arrows	-	-	1	-	-	-	-	-	-	-	-	2
Bob's Arrows	-	-	-	-	-	-	-	-	1	1	1	-
Points Obtained by Bob	-	-	-	-	-	-	-	-	8	9	10	-

Ввод:

numArrows = 3,

aliceArrows = [0,0,1,0,0,0,0,0,0,0,0,2]

Вывод:

[0,0,0,0,0,0,0, 0,1,1,1,0]

Объяснение:

В приведенной выше таблице показано, как оценивается соревнование.

Боб получает общее количество очков 8 + 9 + 10 = 27.

Можно показать, что Боб не может получить оценку выше 27 баллов.

Вариант 5

Дана закодированная строка, верните её декодированную версию. Правило кодирования: k[encoded_string], где encoded_string квадратные скобки повторяются ровно k раз. Обратите внимание, что k это гарантированно положительное целое число.

Вы можете предположить, что входная строка всегда действительна; нет лишних пробелов, правильные квадратные скобки и т.д. Кроме того, вы можете предположить, что исходные данные не содержат никаких цифр и что цифры предназначены только для тех повторяющихся чисел k. Например, не будет ввода типа 3a или 2[4].

Ограничения:

- 1 <= s.length <= 30
- s состоит из строчных английских букв, цифр и квадратных скобок '[]'.
- s гарантированно будет **допустимым** входом.
- Все целые числа s находятся в диапазоне [1, 300].

Пример 1	**Пример 2**	**Пример 3**
Ввод: s = "3[a]2[bc]" **Выход:** "aaabcbc"	**Ввод:** s = "3[a2[c]]" **Вывод:** "accaccacc"	**Ввод:** s = "2[abc]3[cd]ef" **Вывод:** "abcabccdcdcdef"

Вариант 6

Для заданной строки expression чисел и операторов вернуть *все возможные результаты вычисления всех возможных способов группировки чисел и операторов*. Вы можете вернуть ответ в **любом порядке**.

Ограничения:

- 1 <= expression.length <= 20.
- expression состоит из цифр и оператора '+', '-', и '*'.
- Все целые значения во входном выражении находятся в диапазоне [0, 99].

Пример 1	**Пример 2**
Ввод: выражение = "2-1-1" **Вывод:** [0,2] **Объяснение:** ((2-1)-1) = 0 (2-(1-1)) = 2	**Ввод:** выражение = "2*3-4*5" **Вывод:** [-34,-14,-10,-10,10] **Объяснение:** (2*(3-(4*5))) = -34 ((2*3)-(4*5)) = -14 ((2*(3-4))*5) = -10 (2*((3-4)*5)) = -10 (((2*3)-4)*5) = 10

Вариант 7

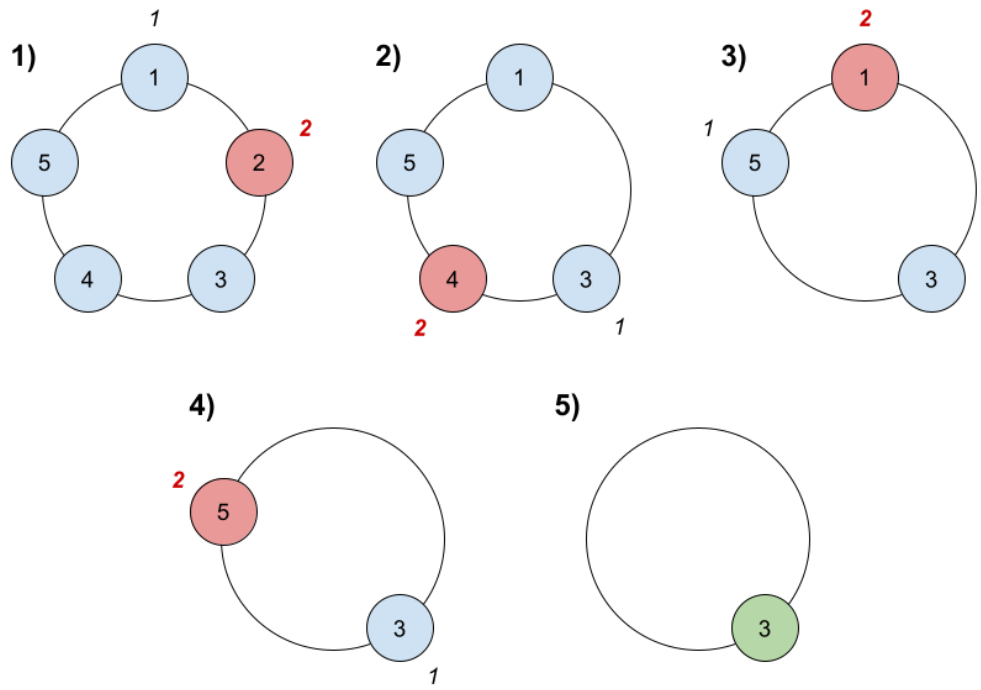
Есть n друзей, которые играют в игру. Друзья сидят в кругу и нумеруются от 1 до n по **часовой стрелке**. Более формально, движение по часовой стрелке от друга i приводит к другу (i+1), при условии 1 <= i < n, а движение по часовой стрелке от друга n (последнего) приводит к первому другу. Правила игры следующие:

1. **Начните** с первого друга.
2. Подсчитайте следующих k друзей по часовой стрелке, **включая** друга, с которого вы начали. Подсчет идет по кругу и может подсчитывать некоторых друзей более одного раза.
3. Последний подсчитанный вами друг покидает круг и проигрывает игру.
4. Если в кругу все еще больше одного друга, вернитесь к шагу 2, **начиная** с друга **сразу по часовой стрелке** от друга, который только что проиграл, и повторите.
5. В противном случае побеждает последний друг в кругу.

Учитывая количество друзей n и целое число k, определите *победителя игры*.

Ограничения: 1 <= k <= n <= 500.

Пример 1



Ввод:	**Выход:**
n = 5, k = 2	1

Объяснение

Вот шаги игры:

1) Начните с друга 1.

2) Отсчитайте 2 друзей по часовой стрелке, это друзья 1 и 2.

3) Друг 2 выходит из круга. Следующий старт - друг 3.

4) Отсчитайте 2 друзей по часовой стрелке, это друзья 3 и 4.

5) Друг 4 выходит из круга. Следующий старт - друг 5.

6) Отсчитайте 2 друзей по часовой стрелке, это друзья 5 и 1.

7) Друг 1 выходит из круга. Следующий старт - друг 3.

8) Отсчитайте 2 друзей по часовой стрелке, это друзья 3 и 5.

9) Друг 5 покидает круг. Остался только друг 3, так что он победитель.

Пример 2	
Ввод: n = 6, k = 5	**Выход:** 1
Объяснение Друзья уходят в следующем порядке: 5, 4, 6, 2, 3. Победителем является друг 1.	

Вариант 8

Вам дан массив целых чисел nums. Два игрока играют в игру с этим массивом: игрок 1 и игрок 2. Игроки 1 и 2 ходят по очереди, причем игрок 1 начинает первым. Оба игрока начинают игру со счетом 0. На каждом ходу игрок берет одно из чисел с любого конца массива (т.е. nums[0]или nums[nums.length - 1]), что уменьшает размер массива на 1. Игрок добавляет выбранное число к своему счету. Игра заканчивается, когда в массиве больше нет элементов.

Возвратите true, если игрок 1 может выиграть игру. Если очки обоих игроков равны, то игрок 1 всё равно остается победителем, и вы также должны вернуть true. Вы можете предположить, что оба игрока играют оптимально.

Ограничения:

- 1 <= nums.length <= 20.
- 0 <= nums[i] <= $10^{7.}$

Пример 1	**Пример 2**
Ввод: nums = [1,5,2] **Вывод:** false **Объяснение:** Изначально игрок 1 может выбирать между 1 и 2. Если он выберет 2 (или 1), то игрок 2 может выбрать из 1 (или 2) и 5. Если игрок 2 выберет 5, то у игрока 1 останется 1 (или 2). Таким образом, итоговый счет игрока 1 равен 1 + 2 = 3, а игрока 2 — 5. Следовательно, игрок 1 никогда не будет победителем, и вам нужно вернуть false.	**Ввод:** nums = [1,5,233,7] **Вывод:** true **Объяснение:** Игрок 1 сначала выбирает 1. Затем игрок 2 должен выбрать между 5 и 7. Независимо от того, какое число выберет игрок 2, игрок 1 может выбрать 233. Наконец, у игрока 1 больше очков (234), чем у игрока 2 (12), поэтому вам нужно вернуть значение True, означающее, что игрок 1 может выиграть.

Вариант 9

Имеются два положительных целых числа n и k. Двоичная строка формируется следующим образом: S_n

- S_1 = "0"
- $S_i = S_{i-1}$ + "1" + reverse(invert(S_{i-1})), при i > 1.

Где «+» обозначает операцию конкатенации, reverse(x) возвращает перевернутую строку x, invert(x) инвертирует все биты в x (0 изменяется на 1, и 1 изменяется на 0). Например, первые четыре строки в приведенной выше последовательности:

- S_1 = "0"
- S_2 = "0**1**1"
- S_3 = "011**1**001"
- S_4 = "0111001**1**0110001"

Верните k-тый бит в S_n .

Ограничения:

- 1 <= n <= 20.
- 1 <= k <= 2^n – 1.

Пример 1	**Пример 2**
Ввод: n = 3, k = 1 **Выход:** "0" **Объяснение:** S_3 равно " **0** 111001". 1 -й бит равен «0».	**Ввод:** n = 4, k = 11 **Вывод:** "1" **Объяснение:** S_4 равно "0111001101 **1** 0001". 11 -й бит равен «1».

Вариант 10

Получив строку s, представляющую допустимое выражение, реализуйте базовый калькулятор для его вычисления и найдите результат вычисления. Примечание: вам не разрешается использовать какие-либо встроенные функции, которые оценивают строки как математические выражения, такие как eval().

Ограничения:

- 1 <= s.length <= $3 * 10^5$
- s состоит из цифр, '+', '-', '(', ')', и ' '.
- s представляет допустимое выражение.
- '+'не используется **как** унарная операция (т . е. "+1"и "+(2 + 3)"недействительна).
- '-'может использоваться как унарная операция (т . е. "-1"и "-(2 + 3)"действительна).
- Во входных данных не будет двух последовательных операторов.
- Каждое число и текущее вычисление помещаются в 32-битное целое число со знаком.

Пример 1	**Пример 2**	**Пример 3**
Ввод: s = "1 + 1" **Выход:** 2	**Ввод:** s = " 2-1 + 2 " **Выход:** 3	**Ввод:** s = "(1+(4+5+2)-3)+(6+8)" **Выход:** 23

Вариант 11

Найдите результат оценки заданного логического значения expression, представленного в виде строки. Выражение может быть либо:

- "t" эквивалентно True;
- "f" эквивалентно False;
- "!(expr)" вычисляет логическое НЕ внутреннего выражения expr;
- "&(expr1,expr2,...)" приводит к логическому И двух или более внутренних выражений expr1, expr2, ...;
- "|(expr1,expr2,...)" оценивает логическое ИЛИ двух или более внутренних выражений expr1, expr2, ...

Ограничения:

- 1 <= expression.length <= 2 * 10^4
- expression[i]состоит из символов в {'(', ')', '&', '|', '!', 't', 'f', ','}.
- expressionявляется допустимым выражением, представляющим логическое значение, как указано в описании.

Пример 1	**Пример 2**	**Пример 3**
Ввод: выражение = "!(f)" **Вывод:** правда	**Вход:** выражение = "|(f,t)" **Выход:** правда	**Ввод:** выражение = "&(t,f)" **Вывод:** ложь