НИУ ВШЭ - Нижний Новгород

November 11, 2024

Алгоритмы и структуры данных

Лекция 1. Массивы, вычислительная сложность, тестирование

Илья Сергеевич Бычков
ibychkov@hse.ru

Повторение

Вопросы

Алгоритм, программа

Вычислительные машины

Языки программирования

Трансляция

Данные, системы счисления, память

Представление данных

Битовые операции

Лекция 1.
Массивы, вычислительная сложность, тестирование

Массивы, вычислительная сложность, тестирование

План лекции

Структура данных: массив

В большинстве случаев объем данных, обрабатываемых алгоритмам, достаточно велик, что не позволяет ограничиваться только использованием переменных (если конечно вы не решаете только задачи типа “A+B Problem”).

В таких случаях на помощь приходят разнообразные структуры данных.

Структура данных — это контейнер, который хранит данные и обеспечивает работу с ними в соответствии с определенными правилами.

Первая и самая простая структура данных (часто ее даже не относят к структурам данных), которую мы рассмотрим, называется массив.

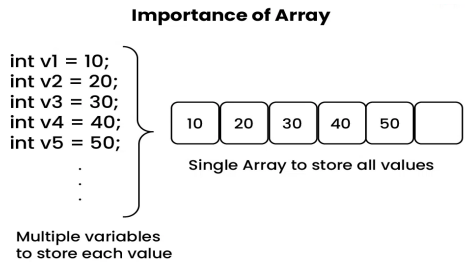
Структура данных: массив

Массив

Массив - это структура данных, для которой характерно:

1. Хранение элементов одного типа
2. Размещение элементов последовательно в памяти
3. Единое имя для всех элементов

Полезно думать о массиве как об определенном наборе переменных одного типа.



(1) Введение в массивы, Источник - GeeksForGeeks

Структура данных: массив

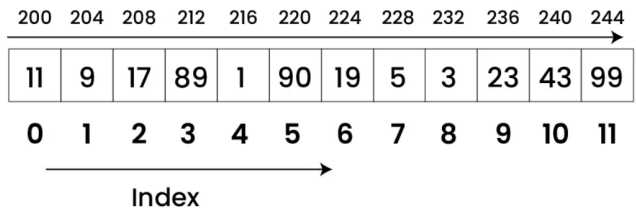
Устройство массива

Имя массива указывает на адрес в памяти его самого первого элемента.

Доступ к элементам массива осуществляется с помощью индексов или смещений.

Самый первый элемент имеет нулевое смещение относительно адреса массива в памяти.

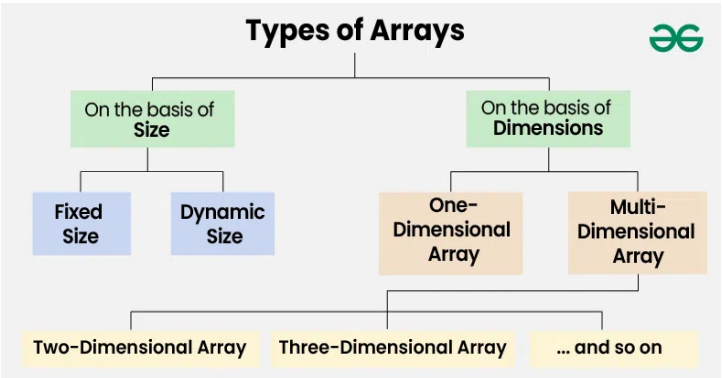
Каждый элемент массива имеет свой адрес и занимает одинаковое и фиксированное число байт, которое зависит от типа данных.



(2) Строение массива, Источник - GeeksForGeeks

Структура данных: массив

Типы массивов



(3) Типы массивов, Источник - GeeksForGeeks

Структура данных: массив

Алгоритмы на массивах

Операции для прямой модификации массивов:

- замена элемента/добавление в конец
- вставка элемента на выбранную позицию
- удаление элемента на выбранной позиции

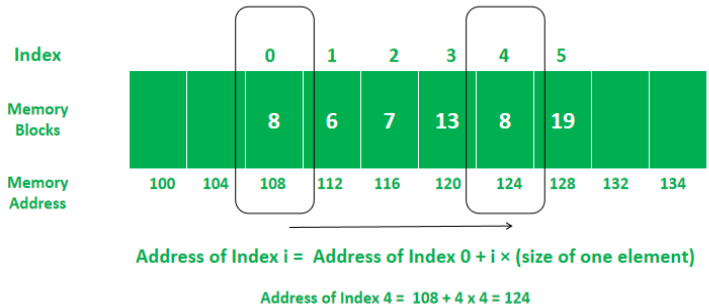
Алгоритмы на массивах:

- поиск элемента с заданными свойствами
- сортировка массива

Структура данных: массив

Доступ к элементам

Благодаря своей природе массив обладает эффективным произвольным доступом (random access) к данным. Любой элемент массива может быть считан или перезаписан за фиксированное число операций, которое не зависит от размера массива. Это называется константной сложностью и записывается как O(1).



(4) Доступ к элементам массива, Источник - GeeksForGeeks

Структура данных: массив

Замена элемента/добавление в конец

Рисуем на доске

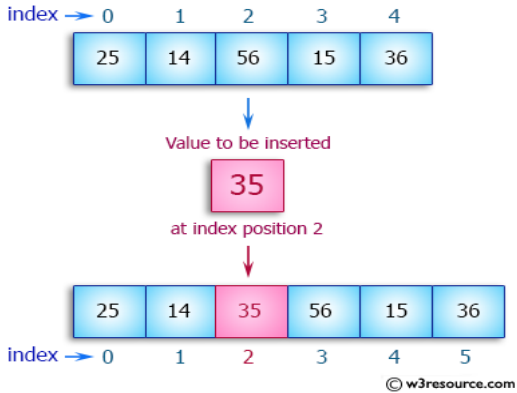
Структура данных: массив

Вставка элемента

Входные данные: массив, элемент для вставки, позиция для вставки

Результат: массив с вставленным элементов

Время операции: в худшем случае линейное по количеству элементов, O(n)



(5) Вставка в массив, Источник - w3resource

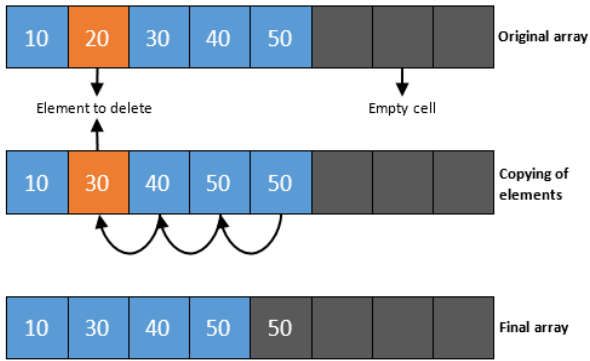
Структура данных: массив

Удаление элемента

Входные данные: массив, позиция удаляемого элемента

Результат: массив без удаленного элемента

Время операции: в худшем случае линейное по количеству элементов, O(n)



(6) Удаление из массива, Источник - неизвестен

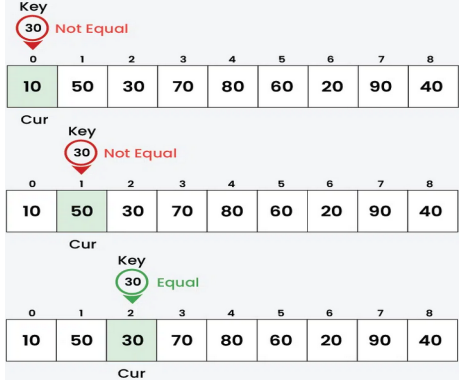
Структура данных: массив

Поиск элемента/элементов с заданными свойствами

Входные данные: массив, искомый элемент или критерий(функция)

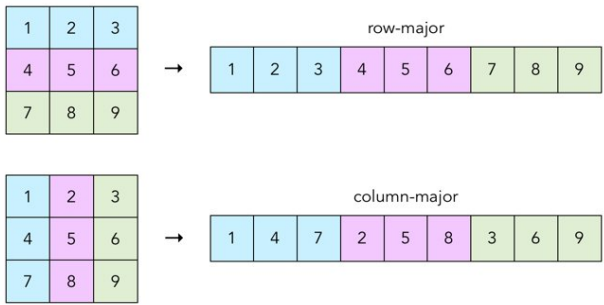
Результат: индекс или набор индексов искомых элементов

Время операции: в худшем случае линейное по количеству элементов, O(n)



(7) Поиск в массиве, Источник - GeeksForGeeks

Многомерные массивы как правило сохраняют свойство последовательного расположения элементов в памяти.



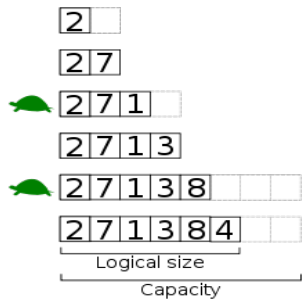
(8) Многомерные массивы, Источник - неизвестен

Структура данных: массив

Динамические массивы

Под динамическими массивами обычно имеют ввиду:

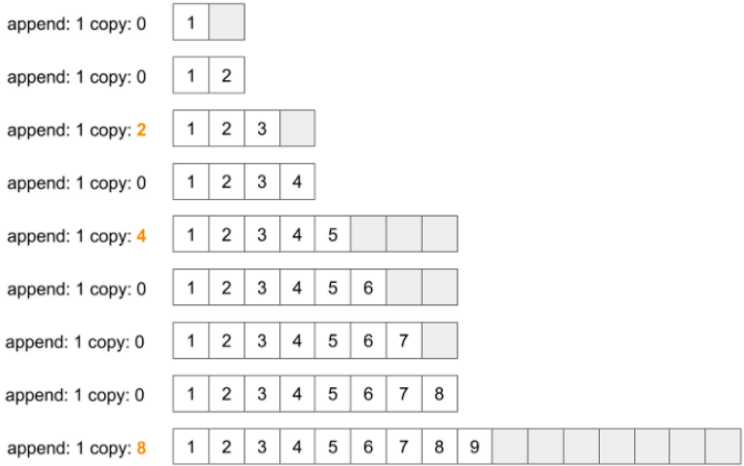
- массивы, размер которых не известен заранее, память для них выделяется на этапе работы программы
- структуры данных, имеющих внутри массивы и реализующие специальную логику по автоматическому увеличению их размера при необходимости



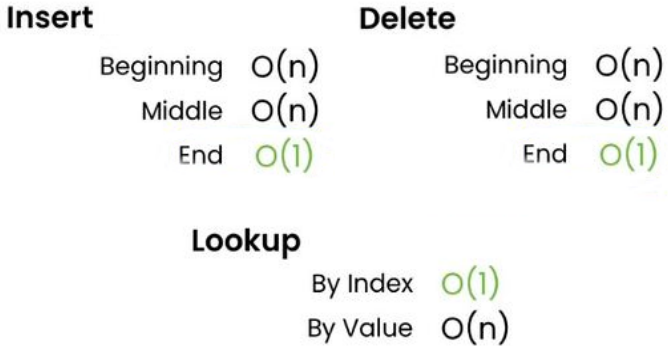
(9) Источник - Wikipedia

Структура данных: массив

Динамические массивы



(10) Пример динамического массива, Источник - Link



(11) Вычислительная сложность операций на массивах

Структура данных: массив

Плюсы и минусы

- ➕ Постоянное время доступа к любому элементу по индексу
- ➕ Содержат только данные
- ➕ Локальность в памяти
- ➖ Нельзя изменять размер в процессе выполнения
- ➖ Необходимость сдвигать элементы при вставках/удалениях

(12) Достоинства и недостатки массивов

Массивы, вычислительная сложность, тестирование

План лекции

Вычислительные задачи и корректность алгоритмов

Вычислительные задачи

Каждый интересующий нас алгоритм (в рамках данного курса) можно рассматривать как специализированный инструмент для решения конкретной вычислительной задачи.

Вычислительная задача определяет связи между набором входных данных и ответом. А алгоритм описывает процедуру воспроизведения этой связи для всех существующих вариантов входных данных.

Экземпляр задачи - конретные набор входных данных для данной вычислительной задачи и соответствующий ему правильный ответ (набор выходных данных).

Задача 1. Дана последовательность A из n различных натуральных чисел, необходимо найти номер элемента с максимальным значением в A.

Входные данные:

$$n \in \mathbb{N}, \qquad 1 \leq n \leq 10^4$$

$$A = a_1, a_2, ..., a_n$$

$$0 \leq a_i \leq 10^6 \qquad \forall i \in \mathbb{N} \qquad 1 \leq i \leq n$$

Выходные данные:

$$1 \leq i \leq n \qquad : \qquad a_i \geq a_k \qquad 1 \leq k \leq n$$

Возможные экземпляры задачи 1

$n = 1$ $A = [42] \to$ Правильный ответ: 1

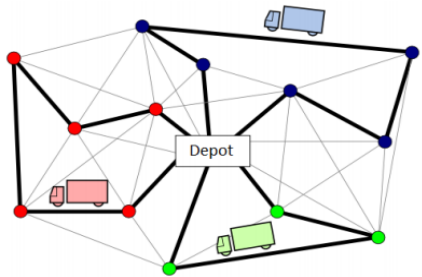
$n = 2$ $A = [10, 100] \to$ Правильный ответ: 2

$n = 8$ $A = [2, 4, 6, 8, 7, 5, 3, 1] \to$ Правильный ответ: 4

$n = 10^4$ $A = [1, 1, 1, ..., 1] \to$ Правильный ответ: ?

Вычислительные задачи и корректность алгоритмов

Более сложные задачи



(13) Vehicle Routing Problem, источник - Link

$$\min \sum_{i \in V} \sum_{j \in V} c_{ij} x_{ij}$$

subject to

$$\sum_{i \in V} x_{ij} = 1 \quad \forall j \in V \setminus \{0\}$$

$$\sum_{j \in V} x_{ij} = 1 \quad \forall i \in V \setminus \{0\}$$

$$\sum_{i \in V \setminus \{0\}} x_{i0} = K$$

$$\sum_{j \in V \setminus \{0\}} x_{0j} = K$$

$$\sum_{i \notin S} \sum_{j \in S} x_{ij} \geq r(S), \quad \forall S \subseteq V \setminus \{0\}, S \neq \emptyset$$

$$x_{ij} \in \{0, 1\} \quad \forall i, j \in V$$

(14) Vehicle Routing Problem, источник - Wikipedia

В современном мире вычислительные задачи возникают практически во все сферы жизни человека, например:

- Логистика: построение маршрутов, оптимизация доставок
- Интернет: организация компьютерных сетей, передача данных, шифрование
- Производство: оптимальное распределение ресурсов
- Искуственный интеллект (AI): алгоримты обучения, сжатия моделей

Как следствие, создание эффективных алгоритмов для решения вычислительных задач становится все более важным направлением.

Вычислительные задачи и корректность алгоритмов

Вычислительные задачи

Алгоритм для решения вычислительной задачи называется правильным (correct) если для любых возможных входных данных он завершается (halts) за конечное время и выдает правильный набор выходных данных.

В этом случае говорят, что алгоритм решает задачу. В противном случае алгоритм либо не завершается, либо выдает неверный ответ на один или более наборов входных данных (или даже завершается с ошибкой).

Как правило, корректность алгоритма приходится доказывать математически (для любых возможных наборов входных данных).

Самый простой метод решения задач, почти не требующий доказательств, это метод грубой силы (brute force) - полный перебор всех вариантов решений.

В редких случаях некорректные алгоритмы тоже могут быть полезны, например, если они потребляют очень мало вычислительных ресурсов и/или при этом степень ошибки в ответе известна и/или может контролироваться пользователем.

Эвристические алгоритмы (эвристики) - способны давать хороший результат на практике при умеренном или низком использовании вычислительных ресурсов. Однако, твердых гарантий по качеству решений нет.

Аппроксимационные алгоритмы - гарантируют качество решения в заявленных границах. Однако, на практике эти границы могут быть весьма широкими.

Массивы, вычислительная сложность, тестирование

План лекции

0. План лекции
1. Структура данных массив
2. Вычислительные задачи и корректность алгоритмов
3. Вычислительная сложность алгоритмов
4. Тестирование программ

Вычислительная сложность алгоритмов

Вычислительная сложность

Алгоритм должен быть корректным, то есть завершаться получением правильного ответа для задачи. Если алгоритм корректен, то имеет смысл говорить о том, насколько эффективно он использует предоставляемые ему ресурсы.

Алгоритмы обычно оценивают по времени выполнения и по используемой памяти.

Такие оценки называют вычислительной сложностью алгоритма. Чем выше вычислительная сложность, тем дольше работает алгоритм и/или больше памяти использует.

Выделяют два вида вычислительной сложности: временная сложность (time complexity) и пространственная сложность (space complexity).

Вычислительная сложность алгоритмов

Вычислительная сложность

Фактическое время работы алгоритма зависит от:

- скорости работы компьютера
- языка программирования, на котором реализован алгоритм
- компилятора или интерпретатора, который переводит программу в исполняемый код
- опыта разрабатывающего программу программиста
- параллельно выполняемых компьютером задач

Все эти факторы являются внешними по отношению к самому алгоритму.

Вычислительная сложность алгоритмов

Вычислительная сложность

Как оценить эффективность алгоритма, а не реализации?

Идея 1 Привязка оценки эффективности алгоритма к размеру используемых им входных данных и самим данным.

Например, в задачах на массивы таким размером обычно будет служить n - количество элементов в массиве.

Один проход по массиву - $\mathcal{O}(n)$ (линейная сложность)

Перебор всех возможных пар элементов (без учета порядка) - $\frac{n \cdot (n-1)}{2} = \mathcal{O}(n^2)$ (квадратичная сложность)

Выполение m вставок в массив - $\mathcal{O}(n \cdot m)$

Как оценить эффективность алгоритма, а не реализации?

Идея 2 Использование "элементарных" операций для подсчета времени работы алгоритма. Элементарными можно считать арифметические операции, ввод-вывод, условный оператор, работу с переменными.

```
void getElement(int *array, int position) {
  clock_t t = clock();

  for (int i = 0; i < NUM_ITERATIONS_PER_EXPERIMENT; i++) {
    int elementValue = 0;
    elementValue = array[position];
  }

  t = clock() - t;
  double timeSpent = ((double)t) / CLOCKS_PER_SEC;
  printf("Experiment: Read value at index %d. Time of %d runs = %f seconds \n",
         position, NUM_ITERATIONS_PER_EXPERIMENT, timeSpent);
}

int main()
{

  int array[ARR_SIZE];

  srand(999);

  for (int i = 0; i < ARR_SIZE - 1; i++) {
    array[i] = rand();
  }

  getElement(array, 0);
  getElement(array, ARR_SIZE / 2);
  getElement(array, ARR_SIZE - 1);

  return 0;
}
```

Можем считать элементарными

Составные операции

Как оценить эффективность алгоритма, а не реализации?

Идея 3 Оценка скорости роста количества операций при увеличении размера входных данных. Пренебрежение менее значимыми компонентами и константами.

Предположим, мы посчитали все операции, которые производит алгоритм и получили

$$T(n) = 5 \cdot n^3 + 4 \cdot n + 3$$

элементарных операций на входе размера n.
Мы можем отбросить слагаемые $4 \cdot n$ и 3, которые при больших n вносят очень маленький вклад в значение функции.
Более того, мы можем отбросить и множитель 5 в старшем слагаемом (через несколько лет компьютеры станут в пять раз быстрее) и сказать, что время работы алгоритма растёт также быстро как n^3.

Пусть даны две функции $f(n)$ и $g(n)$ натурального аргумента n, значениями которых являются положительные действительные числа.

Говорят, что $f(n)$ растёт не быстрее $g(n)$ или

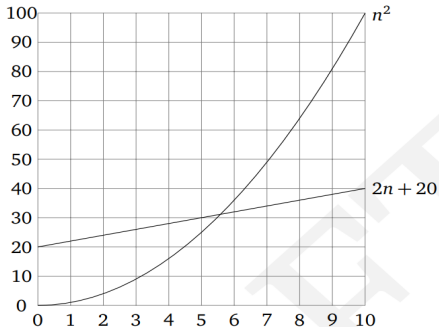
$$f(n) = \mathcal{O}(g(n))$$

если существует такая константа $c \geq 0$ и число n_0, такие что

$$f(n) \leq c \cdot g(n) \quad \forall n : n \geq n_0$$

Пусть имеются два алгоритма: первый выполняет $f(n) = n^2$ операций, а второй $g(n) = 2 \cdot n + 20$. Какой из алгоритмов работает быстрее?

Пусть имеются два алгоритма: первый выполняет $f(n) = n^2$ операций, а второй $g(n) = 2 \cdot n + 20$. Какой из алгоритмов работает быстрее?



(16) Источник - Dasgupta, Papadimitriu, Vazirani

Вычислительная сложность алгоритмов

О символика

Обозначение $f(n) = O(g(n))$ можно считать аналогом математической операции $\leq$. Для операций $\geq$ и $=$ приняты следующие обозначения.

Омега (большая), f растет не медленнее g с точностью до константы:

$$f(n) = \Omega(g(n))$$

Тета (большая), f и g имеют одинаковый порядок роста

$$f(n) = \Theta(g(n))$$

Вычислительная сложность алгоритмов

О символика

$f(n) = \mathcal{O}(g(n))$ - функция $g(n)$ является "верхней границей"
$f(n)$ растет не быстрее $g(n)$ с точностью до константы

$$\exists c \geq 0, n_0 > 0 : f(n) \leq c \cdot g(n) \quad \forall n \geq n_0$$

$f(n) = \Omega(g(n))$ - функция $g(n)$ является "нижней границей"
$f(n)$ растет не медленнее $g(n)$ с точностью до константы

$$\exists c \geq 0, n_0 > 0 : g(n) \leq c \cdot f(n) \quad \forall n \geq n_0$$

$f(n) = \Theta(g(n))$ - функции $g(n)$ и $f(n)$ имеют одинаковый порядок роста

$$\exists c_1 \geq 0, c_2 \geq 0, n_0 > 0 : c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \quad \forall n \geq n_0$$

Вычислительная сложность алгоритмов

О символика

Алгоритм	Эффективность
o(n)	< n
O(n)	≤ n
Θ(n)	= n
Ω(n)	≥ n
ω(n)	> n

(17) Источник - Хабр

Несколько правил работы с О символикой

Благодаря O-символике мы можем заменить $3n^2 + 4n + 5$ на $\Theta(n^2)$, пренебрегая остальными слагаемыми. Вот несколько общих правил такого рода замен:

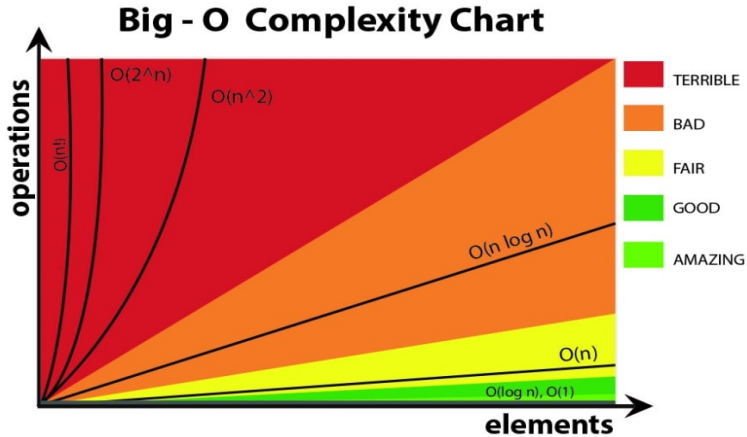
1. Постоянные множители можно опускать. Например, $14n^2$ можно заменить на n^2.

2. n^a растёт быстрее n^b для $a > b$. Например, в присутствии слагаемого n^2 можно пренебречь слагаемым n.

3. Любая экспонента растёт быстрее любого многочлена (полинома). Например, 3^n растёт быстрее n^5.

4. Любой полином растёт быстрее любого логарифма. Например, n (и даже $\sqrt{n}$) растёт быстрее $(\log n)^3$; мы не указываем основания логарифма, поскольку замена основания приводит лишь к умножению логарифма на константу. По тем же причинам n^2 растёт быстрее $n \log n$.

(18) Источник - Dasgupta, Papadimitriu, Vazirani



(19) Источник - указать

Вычислительная сложность алгоритмов

Таблица сложностей

размер / сложность	10	20	30	40	50	60
n	0,00001 сек.	0,00002 сек.	0,00003 сек.	0,00004 сек.	0,00005 сек.	0,00005 сек.
n^2	0,0001 сек.	0,0004 сек.	0,0009 сек.	0,0016 сек.	0,0025 сек.	0,0036 сек.
n^3	0,001 сек.	0,008 сек.	0,027 сек.	0,064 сек.	0,125 сек.	0,216 сек.
n^5	0,1 сек.	3,2 сек.	24,3 сек.	1,7 минут	5,2 минут	13 минут
2^n	0,0001 сек.	1 сек.	17,9 минут	12,7 дней	35,7 веков	366 веков
3^n	0,059 сек.	58 минут	6,5 лет	3855 веков	2×10^8 веков	$1{,}3 \times 10^{13}$ веков

(20) Источник - указать

Массивы, вычислительная сложность, тестирование

План лекции

0. План лекции
1. Структура данных массив
2. Вычислительные задачи и корректность алгоритмов
3. Вычислительная сложность алгоритмов
4. Тестирование программ

Тестирование программы - процесс проверки программы на наличие ошибок. Состоит в поиске наборов входных данных, на которых поведение алгоритма/программы может отличаться от ожидаемого. Тестирование и отладка(debugging) - крайне важные навыки.

Как в алгоритмических соревнованиях, так и в промышленной разработке программ важно последовательно и системно подходить к процессу тестирования.

Проверка алгоритмических задач также основывается на тестировании - тестирующая система подаёт на вход вашей программе различные данные и проверяет правильность ответа.

Тесты из условия. Как правило, в условии задач даётся не менее одного примера с входными данными и верным ответом для них. Эти тесты должны быть проверены в первую очередь, с высокой долей вероятности они есть в проверочном наборе.

Более того, на тестах из условия обычно можно проверить формат и вид выводимого ответа. В данном случае нужно быть предельно внимательным - часто даже небольшое отличие (не хватает пробелов между элементами, символы в разных регистрах) может привести к вердикту WRONG ANSWER, даже если по сути ответ правильный.

Пример

входные данные
3 3 2 -2
выходные данные
-2 2 3

Тесты наименьшего размера/тривиальные тесты.
Самое простое, что можно проверить самостоятельно - тесты наименьшего размера. Если речь идёт о коллекциях элементов (например, массивах) всегда имеет смысл проверять наименьший возможный размер (конечно, если это предусмотрено интервалам входных параметров).

Часто на таких маленьких примерах программы даже ломаются, уходят в бесконечные циклы или даже получают ошибки при обращении к неразрешенному участку памяти.

В других случаях, для наименьших тестов может отличаться или не работать логика, корректная для остальных.

Тестирование программ

Варианты тестов

Тесты с наибольшими возможными данными.
Такие тесты обычно содержат граничные допустимые значения переменных задачи.
Например, максимально возможное число элементов в соответствии с условием.
Или максимально большие значения элементов.

Тесты с наибольшими данными призваны проверить несколько вещей:

- укладывается ли предложенный Вами алгоритм в ограничение по времени
- укладывается ли предложенный Вами алгоритм в ограничение по памяти
- правильно ли выбраны типы данных внутри алгоритма, нет ли ошибок переполнения типов

Несмотря на то, что с опытом оценить время и память часто можно без использования дополнительных тестов, иногда бывает необходимо даже написать отдельную программу, которая генерирует большой тест.

Тестирование программ

Варианты тестов

В задаче про суммы максимальными тестами можно считать:

- тесты с граничными значениями элементов $a = 10^9$, $a = -10^9$
- тесты массивов с максимальным числом элементов - массив из 500 строк и 500 столбцов
- оба пункта выше одновременно

D. Сумма двумерного массива

ограничение по времени на тест: 1 секунда
ограничение по памяти на тест: 256 мегабайт
ввод: стандартный ввод
вывод: стандартный вывод

Дан двумерный массив, состоящий из n строк и m столбцов. Задача - посчитать сумму значений в каждой строчке и в каждом столбце. В результате у получится два массива длины n и m соответственно.

Входные данные

Первая строка входных данных содержит два числа через пробел: n ($1 \le n \le 500$) и m ($1 \le m \le 500$) — размер массива

Следующие n строк содержат m чисел, разделенных пробелами, каждое число — a_{ij} значение исходного массива ($-10^9 \le a_{ij} \le 10^9$).

Тесты с различными типами ответов.
Корректному алгоритму почти всегда необходимо по-разному реагировать на различные экземпляры или наборы экземпляров входных данных.

Например, для четного количества элементов массива логика может отличаться (из условия задачи) от логики для нечетного количества элементов.

Или, если речь идет о таком классе задач как игры, на часть примеров нужно будет ответить, например, "Выиграет первый игрок", а на другую "Выиграет второй игрок".

Важно придумать по несколько тестов для каждого возможного варианта ответов (или для каждого возмозного пути выполнения) вашего алгоритма.