|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Вопрос** | **Что это** | **Асимптотика** | **На чем реализуется** | **Ограничения** | **Применимость** |
| **Оценка сложности по времени** | Это метод, который позволяет оценить время работы написанного кода.  • 1 операция - 1 пункт сложности  • 1 итерация = k операций в итерации – k пунктов сложности  • Условие = t логических операций и действий – t пунктов сложнсти  -Обозначается Т(n)  Обычно алгоритм оценивают сверху (О(n)) и снизу(Ω(n))  (асимптотическая оценка), а также приводят среднюю асимптотику, наблюдаемую в большинстве случаев. | - | - | - | В любой программе |
| **Оценка сложности по памяти** | Это метод, который позволяет оценить память, которую использует программа, написанная данным кодом.   * 1 переменная - 1 пункт сложности * Массив размером n - n пунктов сложности * Строка длинной n - n пунктов сложности   Обозначается как M(n) | - | - | - | В любой программе |
| **Сортировка вставками** | Эта устойчивая сортировка, работающая по  следующему принципу (опишем принцип сортировки по возрастанию):  1. Пусть массив теперь делится на 2 части: отсортированную (первый элемент) и не отсортированную (остальные элементы), будем хранить конец отсортированной части в переменной k  2. Берется указатель (или индекс) числа (пусть это  будет I = k + 1), стоящего после отсортированной части.  3. Другим указателем (пусть это будет j) идем по отсортированной части (j = k и до 0), проверяем, значение элемента m[j] > m[i]:  a. если да, то меняем их друг с другом (swap (m[i], m[j])), увеличиваем i на 1 и переходим опять на пункт 2  b. иначе присваиваем m[j] = m[j – 1], уменьшаем j на 1, повторяем пункт 3 | ПО ВРЕМЕНИ  Лучший: О(n) Средний: O(n^2)  Худший: O(n^2)  ПО ПАМЯТИ  O(1), т.к. мы храним только указатель на конец отсортированной  части | Реализуется на массиве исходных данных (например, чисел) | Ограничения могут быть только на объем входных данных | В задачах, где необходимо отсортировать  массив из любых элементов, может понадобиться как проверка работы более сложных сортировок |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| С**ортировка слиянием (merge sort)** | Это устойчивая (зависит от реализации, но реализация устойчивой и неустойчивой очень схожа) сортировка, работающая по принципу «Разделяй и властвуй» через рекурсию (на спуске делим, на подъеме сортируем и объединяем):   1. Берется массив и делится на 2 части 2. Сортируем 1-ю часть и 2-ю часть так:    * Если длинна частей = 1, то они уже отсортированы    * Иначе работаем с обеими частями как в пункте 1 3. Объединяем 2 отсортированные части так, чтобы новый массив тоже был отсортирован методом 2-х указателей: указатели i и j указывают на начало 2-х   частей, сравниваем a[i] < b[j], если да, то кладем a[i] в новый массив и увеличиваем i на 1, иначе кладем b[j] в новый массив и увеличиваем j на 1. Когда одна из частей закончилась (i == n or j == m), кладем  оставшиеся числа из другой части в новый массив и возвращаем новый массив как ответ. | ПО ВРЕМЕНИ  О(n log(n)), т.к. мы  пройдемся по всем n элементам массива log n раз (глубина  рекурсии)  ПО ПАМЯТИ  O(n), т.к. нужен только доп. массив | Реализуется на рекуррентном  спуске по массиву | Ограничение по глубине рекурсии |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Быстрая сортировка (quick sort)** | Это неустойчивая сортировка, работающая по одному из следующих принципов:  I. Разбиение Хоара  1. На каждой итерации в массиве выбирается опорный элемент из середины, относительно которого  сортируется массив (для улучшения асимптотики берется случайное число из массива и меняется с центральным, это одна из возможных оптимизаций кода).  2. Заводится 2 указателя i и j – на начало и конец массива соответственно, при этом считаем, что все элементы, которые находятся слева от опорного ДОЛЖНЫ быть меньше него, а справа ДОЛЖНЫ быть больше него.  3. Дальше повторяем пункты, пока i ≤ j:  • Пока элемент по индексу i меньше, чем опорный, увеличиваем i на 1  • Пока элемент по индексу j больше, чем опорный, уменьшаем j на 1  • Если i ≤ j, то меняем элементы по этим индексам местами, увеличиваем i на 1, уменьшаем j на 1 и возвращаемся на пункт 3  4. Запускаем сортировку заново, но теперь на отрезке массива от L до j и от i до R, где L, R – начало и конец массива соответственно.  II. Разбиение Ломуто  1. На каждой итерации опорным элементом является последний в массиве.  2. Заводится 2 указателя i и j, где i указывает на конец части массива, в которой все элементы меньше или равны опорному (изначально = -1, т.к нет пока такой части), j указывает на начало массива.  3. Пока j < len(массив):  • Если элемент по индексу j ≤ опорного, то мы добавляем его в отсортированную часть, т.е.  увеличиваем индекс i на 1 и меняем элементы на индексах i и j  • Иначе ничего не делаем  • Увеличиваем j на 1  4. Запускаем сортировку на отрезках от L до i и от i + 1  до R, где L, R – начало и конец массива | ПО ВРЕМЕНИ  Лучший: O(nlogn) Средний: O(nlogn)  Худший: O(n^2)  ПО ПАМЯТИ  Лучший: O(logn) Средний: O(logn)  Худший: O(n) | Реализуется на  рекурентном вызове сортировки на отдельных частях массива | Ограничениепо глубине рекурсии | Для быстрой сортировки (как ни странно) массива любых элементов |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Сортировка подсчетом** | Это линейная сортировка (устойчивость зависит от реализации), которая работает по следующему  принципу:  I. Неустойчивая реализация   1. Заводим массив-алфавит, где храним данные в виде: возможное число – его кол-во в исходном массиве (массив алфавит должен быть отсортирован по   индексам)   1. Проходимся по начальному массиву и заполняем массив-алфавит 2. Проходимся по массиву алфавиту, и пока значение по i-тому индексу > 0 выводим индекс i и уменьшаем значение по i-тому индексу на 1, затем увеличиваем индекс i на 1   II. Устойчивая реализация   1. Заводим массив-алфавит а так же массив, куда будем записывать ответ 2. Проходимся по исходному массиву и заполняем массив алфавит так же, как в неустойчивой реализации 3. Теперь, начиная с i = 1 и до конца массива-алфавита, проходимся по массиву-алфавиту и делаем C[i] = C[i] + C[i – 1] – так мы создадим массив, в котором для i-того элемента будет храниться индекс последнего элемента в массиве ответов B 4. Теперь проходимся по начальному массиву С   КОНЦА, делаем так: ставим число A[i] из изначального массива на позицию, которая численно равна  значению C[A[i]] от этого элемента, т.е. B[C[A[i]]] = A[i] и уменьшаем C[i] на 1  Модификации:   1. если не известен диапазон чисел, можно за линию найти минимум и максимум в массиве, тогда числа от минимума до максимума будут составлять диапазон 2. если числа отрицательные, можно завести дельту, на которую сначала будем увеличивать все числа, а при добавлении в ответ уменьшать, чтобы индексы были   неотрицательными | K – размер алфавита, т.е. кол-во всех возможных чисел   1. **я реализация**   ПО ВРЕМЕНИ  Лучший: O(n + k) Средний: O(n + k) Худший: O(n + k)  ПО ПАМЯТИ  Лучший: O(k) Средний: O(k) Худший: O(k)   1. **я реализация**   ПО ВРЕМЕНИ  Лучший: O(n) Средний: O(n) Худший: O(n)  ПО ПАМЯТИ  Лучший: O(n + k) Средний: O(n + k) Худший: O(n + k) | Реализуется на массиве-алфавите, в котором хранится кол-во каждого возможного числа  из массива и, если используем 2-ю реализацию, на массиве ответе | Не всегда известен алфавит, например,  если надо сортировать какие-то сложные  структуры или дробные числа | Если надо очень быстро отсортировать  числа, и при этом  известен диапазон, откуда они берутся |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Цифровая сортировка** | Это линейная устойчивая сортировка, которая работает по следующему принципу:   1. У каждого числа смотрим на разряд единиц. 2. Используем УСТОЙЧИВУЮ линейную сортировку (если не устойчивая, то сортировка работать не будет, т.к. например на числах 112 и 312 она спокойно сможет ошибиться), опираясь на цифры (алфавит будет от 0 до 9) 3. Повторяем с пункта 1, но теперь смотрим на десятки и так далее, пока не дойдем до самого старшего разряда среди всех чисел включительно   Так же данная сортировка может сортировать и строки, тогда алфавитом будет алфавит (простите за  тавтологию), а вместо разрядов мы смотрим сначала на последнюю букву, затем на предпоследнюю и так далее | d – максимальное  число разрядов, T(n)  – время работы линейной  сортировки, в нашем случае подсчетом  ПО ВРЕМЕНИ  Лучший: O(d T(n)) = O(d n)  Средний: O(d T(n)) = O(d n)  Худший: O(d T(n)) = O(d n)  ПО ПАМЯТИ  Лучший: O(n + k) Средний: O(n + k) Худший: O(n + k) | Реализуется на устойчивой линейной сортировке по разрядам | Можно сортировать только числа и строки | Быстрая сортировка чисел или строк |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Стек** | Это структура, работающая по принципу First Input Last Output, т.е. элемент, который положили в стек первым, достанут оттуда последним (как пирамидка детская,  если уж совсем не понятно)  Реализация основных команд стека:   1. Добавить элемент:   Увеличиваем указатель конца стека на 1 и на это место ставим элемент   1. Достать элемент:   Сверяем указатель на начало и конец стека, если конец  ≥ начало, то возвращаем элемент, на который указывает указатель конца стека   1. Удалить элемент:   Сверяем указатель на начало и конец стека, если конец  ≥ начало, то уменьшаем указатель конца на 1   1. Найти элемент:   Достаем значение из стека и удаляем его, пока не найдем нужный элемент и пока размер стека > 0 | Достать элемент: O(1) Положить элемент: O(1)  Найти элемент: O(n) | Реализуется на массиве, где будут храниться элементы, а так же создаются 2 указателя – на начало и конец стека | Ограничен по памяти, т.е. необходимо выделить какое-то  фиксированное кол-во памяти при реализации сразу | Для различных задач, например для скобочной  последовательности или постфиксной записи |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Очередь** | Это структура, работающая по принципу First Input First Output, т.е. элемент, который положили в очередь  первым, достанут оттуда первым (очередь все знают)  Реализация основных команд очереди:   1. Добавить элемент:   Увеличиваем указатель конца очереди на 1, если указатель выходит за пределы массива, то  приравниваем его к нулю, и на это место ставим элемент   1. Достать элемент:   Сверяем указатель на начало и конец стека, если конец  != начало, то возвращаем элемент, на который указывает указатель начала очереди   1. Удалить элемент:   Сверяем указатель на начало и конец стека, если конец  != начало, то увеличиваем указатель начала на 1, если указатель вышел за пределы массива, то  приравниваем его к нулю   1. Найти элемент:   Достаем значение из очереди и удаляем его, пока не  найдем нужный элемент и пока размер очереди > 0 | Достать элемент: O(1) Положить элемент: O(1)  Найти элемент: O(n) | Реализуется на массиве, где будут храниться элементы, а так же создаются 2 указателя – на начало и конец  очереди | Ограничена по памяти, т.е. необходимо выделить какое-то  фиксированное кол-во памяти при реализации сразу | Для решения различных задач |
| **Односвязный список** | Это структура, элементы которой могут храниться в любых ячейках памяти, т.е. она динамически выделяет память для элементов  Конечный элемент всегда ссылается на что-то, что  указывает, что это конец списка (например, на nullptr  или на себя же)   1. Добавить элемент:   Создаем элемент, который указывает на начало списка, и меняем указатель начала списка на этот элемент   1. Достать элемент:   Вернуть значение, на которое указывает указатель начала списка   1. Удалить элемент:   Указатель начала меняем на адрес следующего элемента, на который ссылался начальный элемент (т.е. tail = tail.next) и удаляем старый элемент   1. Поиск элемента как в стеке/очереди | Достать элемент: O(1) Положить элемент: O(1)  Найти элемент: O(n) | Реализуется на  структуре элемента, который хранит в  себе значение и указатель на следующий (или предыдущий) элемент, а так же  указатель на начало (конец) | - | Для задач, где начальный размер массива неизвестен и нужно  динамически выделять память |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Двусвязный список** | Это структура, которая схожа с односвязным списком, но каждый элемент хранит еще и предыдущий элемент  Все операции схожи с односвязным списком, но их можно разворачивать (например, добавить в конец или взять из начала и т.д.) | Достать элемент: O(1) Положить элемент: O(1)  Найти элемент: O(n) | Реализуется на  структуре элемента, который хранит в  себе значение и указатель на следующий и  предыдущий элемент, а так же  указатель на начало  и конец | - | Может  понадобиться для реализации двухсторонней очереди, а так же для реализации  стека и очереди при помощи всего одной структуры |
| **Циклический список** | Это либо односвязный список, конец которого ссылается на начало, либо двухсвязный список, следующий элемент конца и предыдущий элемент начала которого равны ссылке на начало и конец соответственно  Операции схожи с односвязным/двухсвязным списками с учетом того, что надо всегда при добавлении/удалении элемента сохранять  цикличность списка  На односвязном списке:  Циклический однонаправленный список  На двухсвязном списке:  Циклический двунаправленный список | Достать элемент: O(1) Положить элемент: O(1)  Найти элемент: O(n) Сдвинуть циклически массив на x: O(1) | Реализуется на односвязном или двухсвязном списке, конец и начало которых связанны | - | Например, задача на циклический сдвиг, или если надо решать задачу с  числами, которые стоят по кругу |
| **Стек на списках** | Это реализация структуры стека на односвязном  списке, по факту односвязный список – это и есть стек | Достать элемент: O(1) Положить элемент: O(1)  Найти элемент: O(n) | Реализуется на односвязном списке | - | То же, что и у обычного стека, но память  динамическая |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Очередь на**  **списках** | Это реализация структуры очереди на двухсвязном списке, но так же удаление и получение элемента  происходит с последним, а не с начальным элементом | Достать элемент: O(1) Положить элемент: O(1)  Найти элемент: O(n) | Реализуется на двухсвязном списке | - | То же, что и у обычной очереди, но память  динамическая |
| **Бинарный поиск** | Это метод поиска элемента V в отсортированном по какому-то критерию массиве (нам пока важно знать, что массив просто отсортирован, до бин поиска по ответу мы еще не дошли), реализация:   1. Берем начало отрезка массива L и R (изначально 0 и   n – 1 соответственно), повторяем все пункты пока R > L  + 1   1. Теперь находим элемент M (M – индекс среднего элемента), стоящий в середине отрезка, т.е. M = (L + R)   / 2   1. Теперь в зависимости от того, что мы хотим,   сравниваем значение, стоящее на середине отрезка, и данный элемент V:   * + Если мы ищем первое вхождение элемента в массиве (например, в массиве 1,2,3,3,3,3,3,4 первое вхождение элемента 3 на позиции 2, т.к. индексация с нуля, а последнее вхождение на позиции 6), то если значение среднего элемента СТРОГО МЕНЬШЕ, чем V, то делаем L = M, иначе R = M   + Если мы ищем последнее вхождение элемента в массиве, то если значение среднего элемента МЕНЬШЕ ИЛИ РАВНО, чем V, то делаем L = M, иначе R = M  1. Повторяем все то же самое, начиная с пункта 2 2. Возвращаем индекс найденного элемента или -1   если не нашли его | Поиск элемента: O(logn), потому что при каждой новой итерации мы работаем с  массивом, меньшим предыдущего ровно в 2 раза -> кол-во вызовов бин поиска = logn | Реализуется на  списке и указателях на начало и конец отрезка | Можно применять ТОЛЬКО на  отсортированном массиве | Поиск первого и последнего  элемента в массиве, проверка на нахождение  элемента в массиве и т.д. |