Глава 1

Алгоритмы сортировки

Сортировка — один из основных видов алгоритмов, применяемых во многих задачах. Возникает вопрос, какой из алгоритмов необходимо выбрать для решения конкретной задачи. Для этого необходимо рассмотреть вопрос об эффективности алгоритма. В данном курсе рассмотрим только основы анализа сложности. Более подробное рассмотрение будет на старших курсах.

1.1 Элементы анализа сложности алгоритмов

Одной из основных характеристик алгоритма является его эффективность.

Пример 1.1. Например, необходимо перемножить 3 матрицы: A размерностью 100×10, B размерностью 10×50 и C размерностью 50×5. Перемножать можно только $A \times B \times C$, иначе будут не совпадать соответствующие размерности. Возникает вопрос, где поставить скобки? Какой вариант даст меньше действий: $(A \times B) \times C$ или $A \times (B \times C)$? Для трех матриц можно подсчитать вручную. А если этих матриц больше, будет намного больше вариантов и для этой задачи уже требуется написать алгоритм, который будет определять как правильно поставить скобки.

Рассмотрим эти три матрицы. Результатом перемножения двух матриц размерностями $n \times m$ и $m \times k$ будет матрица размерностью $n \times k$. Для определения каждого элемента матрицы необходимо m раз выполнить операции сложения и умножения по формуле: $c[i][j] = a[i][k] \times b[k][j]$, т. е. всего выполнить $n \times k \times m$ операций.

Первый случай: $(A \times B) \times C$. Результатом перемножения в скобках будет матрица размерностью 100×50 и выполнено $100 \times 50 \times 10 = 50000$ операций. Результатом перемножения полученной матрицы на C будет матрица размерностью 100×5 и выполнено $100 \times 5 \times 50 = 25000$ операций. Всего сделано 75000 операций.

Второй случай: $A \times (B \times C)$. Результатом перемножения матриц в скобках будет матрица размерностью 10×5 и выполнено $10 \times 50 \times 5 = 2500$ операций. Результатом перемножения матрицы A и полученной матрицы будет матрица размерностью 100×5 и выполнено $100 \times 10 \times 5 = 5000$ операций. Всего сделано 7500 операций.

Видно, что второй случай требует в 10 раз меньше операций. Теперь представим, что размерность представленных матриц, например, 1000×1000 элементов и матриц 10. Очевидна необходимость правильной расстановки скобок. □

Для оценки эффективности в середине 50-ых годов 20 века было введено понятие RAM-машины. Это некий гипотетический компьютер (Random Access Machine), т. е. машина с произвольным доступом к памяти. Обладает следующими свойствами:

1. Для исполнения простой операции ($+, -, *, =$, `if`) требуется ровно один временной шаг.
2. Каждое обращение к памяти занимает один временной шаг. При этом неограничен объем оперативной памяти.
3. Цикл и подпрограмма состоят из нескольких простых операций. Время исполнения зависит от количества итераций или характеристик подпрограммы.

Естественно, это очень упрощенная абстракция работы компьютера, однако она позволяет оценивать основное время работы алгоритма. Время работы алгоритма оценивается в шагах, необходимых для решения задачи.

Для того, чтобы говорить о том, плохим или хорошим является алгоритм, нужно знать как он работает для всех экземпляров задачи. Разделяют наилучший, наихудший и средний случай сложности алгоритмов. Проще всего объяснить на примере алгоритма сортировки.

Сложность алгоритма в наихудшем случае — функция, определяемая максимальным количеством шагов, необходимым для обработки любого экземпляра задачи. Для сортировок — количество шагов, необходимое для сортировки отсортированной в обратном порядке последовательности.

Сложность алгоритма в наилучшем случае — функция, определяемая минимальным количеством шагов, необходимым для обработки любого экземпляра задачи. Для сортировок — количество шагов, необходимое для сортировки уже отсортированной последовательности.

Сложность алгоритма в среднем случае — функция, определяемая средним количеством шагов, необходимым для обработки всех экземпляров задачи. Для сортировок — количество шагов, необходимое для сортировки случайной последовательности.

Для большинства алгоритмов можно оценить работу в наилучшем и наихудшем случае. В среднем случае чаще всего можно оценить только приблизительно.

Можно оценить сложность алгоритма, подсчитав количество шагов, выполняемое для решения определенной задачи. Но сложно судить о полученном результате, если получился, например, такой многочлен $f(n) = a + bn + cn^2$. Поэтому вводятся асимптотические обозначения:

$\Theta(g(n))$. Функция $f(n)$ принадлежит множеству $\Theta(g(n))$, если $\exists_1 > 0, c_2 > 0, n_0$, такие, что для $\forall n \geqslant n_0$ выполняется $0 \leqslant c_1 g(n) \leqslant f(n) \leqslant c_2 g(n)$. Говорят, что $g(n)$ является асимптотически точной оценкой функции $f(n)$. (см. рисунок 1.1а)

$O(g(n))$. Функция $f(n)$ принадлежит множеству $O(g(n))$, если $\exists > 0, n_0$, такие, что для $\forall n \geqslant n_0$ выполняется $0 \leqslant f(n) \leqslant cg(n)$. Говорят, что $O(g(n))$ определяет асимптотически верхнюю границу функции $f(n)$. (см. рисунок 1.1б)

$\Omega(g(n))$. Функция $f(n)$ принадлежит множеству $\Omega(g(n))$, если $\exists > 0, n_0$, такие, что для $\forall n \geqslant n_0$ выполняется $0 \leqslant cg(n) \leqslant f(n)$. Говорят, что $g(n)$ определяет асимптотически нижнюю границу функции $f(n)$. (см. рисунок 1.1в)

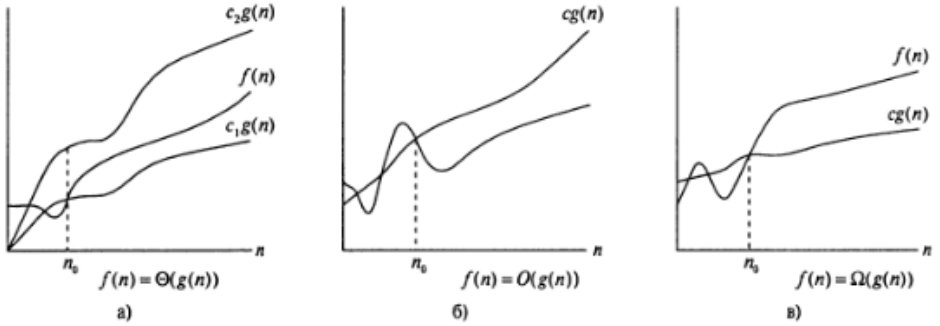


Рис. 1.1: Иллюстрация асимптотических обозначений. Рисунок взят из книги [*Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн*. Алгоритмы. Построение и анализ]

Пример 1.2. Определим, что $f(n) = \frac{n^2}{2} - 3n = \Theta(n^2)$. Для этого необходимо определить такие n_0, c_1, c_2, для которых выполняется соотношение $c_1 n^2 \leqslant \frac{n^2}{2} - 3n \leqslant c_2 n^2$.

Разделим все части неравенства на n^2, получим $c_1 \leqslant \frac{1}{2} - \frac{3}{n} \leqslant c_2$.

Правое неравенство выполняется для всех $n \geqslant 1$ и $c_2 \geqslant 0.5$. Левое неравенство выполняется для всех $n \geqslant 7$ и $c_1 \leqslant \frac{1}{14}$. Тогда, например, для $n_0 = 7, c_1 = \frac{1}{14}, c_2 = 0.5$. указанное выражение выполняется. □

Вообще говоря, можно доказать, что для полинома вида $p(n) = \sum_{i=0}^{d} a_i n^i$, где $a_d > 0$ $p(n) = \Theta(n^d)$.

Очень редко требуется оценка сверху и снизу. Чаще всего при анализе сложности алгоритмов используется $O(g(n))$.

В качестве примера того, что для анализа эффективности необходимо знать только объем исследуемых данных, рассмотрим следующий пример.

Пример 1.3. Есть два компьютера — A и B. Компьютер A более быстрый и выполняет миллиард инструкций в секунду, B — более медленный и выполняет десять миллионов инструкций в секунду.

Пусть есть два алгоритма. Один выполняется за время $c_1 n$, где n — объем данных, а c_1 — константа, включающая в себя все данные процессора, компилятора и т. д. Второй алгоритм выполняется за время $c_2 n \log n$, где под $\log n$ понимается $\log_2 n$. Пусть $c_1 < c_2$.

На компьютере A выполняется первый алгоритм, на компьютере B — второй. Пусть $c_1 = 2, c_2 = 50$.

Определим время работы алгоритмов для $n = 100$:

Для первого алгоритма — $\frac{2 \times (10^2)^2}{10^9} = 2 \times 10^{-5} c$.

Для второго алгоритма — $\frac{50 \times 10^2 \times \log_2 10^2}{10^7} = \frac{50 \times 10^2 \times \log 10^2}{10^7 \times \log 2} \approx 3.3 \times 10^{-5} c$.

Определим время работы алгоритмов для $n = 10^6$:

Для первого алгоритма — $\frac{2\times(10^6)^2}{10^9} = 2000c$.

Для второго алгоритма — $\frac{50\times10^6\times\log_2 10^6}{10^7} = \frac{50\times10^6\times\log 10^6}{10^7\times\log 2} \approx 100c$.

Видно, что для малых n время работы практически совпадает, а с ростом n второй алгоритм начинает работать быстрее, несмотря на более быстрый компьютер и меньшую константу. □

Таким образом, важным является не сама функция, а *скорость роста функции*.

1.2 Общие вопросы алгоритмов сортировок

Сортировка заключается в переупорядочении элементов таким образом, чтобы их ключи следовали в соответствии с четко определенными правилами (например, по возрастанию или в алфавитном порядке).

Основной характеристикой алгоритма сортировки является время, затрачиваемое на его выполнение. В принципе, все алгоритмы сортировки можно разделить на три класса:

- Алгоритмы, выполняющиеся за время, пропорциональное N^2.
- Алгоритмы, выполняющиеся за время, пропорциональное $N \log N$.
- Алгоритмы, выполняющиеся за линейное время. Но это алгоритмы не основанные на сравнениях и заменах.

Дополнительная память, используемая алгоритмом сортировки, является вторым по важности фактором. Все методы сортировки по этому фактору можно разделить на три класса:

- Алгоритмы, не требующие дополнительной памяти (кроме, возможно памяти для хранения одного элемента).
- Алгоритмы, использующие для доступа к данным указатели или индексы и требующие дополнительной памяти для размещения N указателей или индексов.
- Алгоритмы, требующие дополнительной памяти для размещения еще одной копии массива.

В случае наличия элементов с несколькими ключами возникает одна из важных характеристик алгоритмов сортировки — **устойчивость**.

Алгоритм сортировки является устойчивым, если при наличии дублированных ключей, сохраняется относительный порядок размещения.

Например, есть список студентов, отсортированный по фамилии. В случае сортировки этого списка по годам рождения алгоритмы устойчивой сортировки оставят списки студентов, родившихся в один год, в алфавитном порядке.

Еще иногда говорят об эффективной сортировке, т. е. прекращающей свою работу в случае отсортированной последовательности.

1.3 Обменные сортировки

В дальнейшем во всех примерах (если не указано обратное) будут приводится только итерации, во время которых произошел обмен. Полужирным шрифтом будут выделяться стоящие на своих местах элементы (согласно алгоритму). Обмены показаны двумя разными цветами.

1.3.1 Глупая сортировка

Начнем с абсолютно неиспользуемой сортировки, называемой *глупой*. Идея такова:

Алгоритм 1: Глупая сортировка

Вход: A — массив размерности N
Выход: Отсортированный массив
начало алгоритма
  цикл пока *не дошли до конца массива* **выполнять**
    если $a[i] > a[i+1]$ **то**
      · меняем их местами;
      · возвращаемся в начало массива;
конец алгоритма

Так как меняем местами только соседние элементы и неравенство строгое, то сортировка является устойчивой.

Рассмотрим сортировку на примере 7 элементов $\{8, 3, 9, 6, 1, 2, 5.\}$

8	3	9	6	1	2	5
3	8	9	6	1	2	5
3	8	6	9	1	2	5
3	6	8	9	1	2	5
3	6	8	1	9	2	5
3	6	1	8	9	2	5
3	1	6	8	9	2	5
1	3	6	8	9	2	5
1	3	6	8	2	9	5
1	3	6	2	8	9	5
1	3	2	6	8	9	5
1	2	3	6	8	9	5
1	2	3	6	8	5	9
1	2	3	6	5	8	9
1	2	3	5	6	8	9
1	2	3	5	6	8	9

Попробуем оценить сложность этой сортировки.

В лучшем случае все легко. Есть отсортированный массив. Проходим 1 раз по нему, убеждаемся, что он отсортирован и завершаем алгоритм. Следовательно, в лучшем случае сложность $O(n)$.

В худшем случае будет рассматривать отсортированный в обратном порядке массив. Подсчитаем отдельно число сравнений и число замен. На i-том шаге (работаем в нулевой нотации) делаем $i+1$ сравнение и 1 замену.

Так как после каждой замены происходит возврат в начало, то, например, чтобы поставить на свое место четвертый элемент $(2, 3, 4, 1)$:

- $i = 0, 1, 2, 3; -$ 4 сравнения, 1 замена: $2, 3, 1, 4$.
- $i = 0, 1, 2; -$ 3 сравнения, 1 замена: $2, 1, 3, 4$.
- $i = 0, 1; -$ 2 сравнения, 1 замена: $1, 2, 3, 4$.
- $i = 0; -$ 1 сравнение, i-ый элемент стоит на своем месте.

То есть, для того, чтобы поставить на свое место i-ый элемент, необходимо выполнить $i+1$ замену и $\sum\limits_{k=1}^{i+1} k = i\frac{1+i+1}{2} = \frac{i^2+2i}{2}$ сравнений.

Всего необходимо поменять N элементов. Следовательно, число замен: $\sum\limits_{i=0}^{N-1} i+1 = \frac{N^2+N}{2}$. Число сравнений: $0.5\sum\limits_{i=0}^{N-1} i^2+2i = N\frac{N^2+2N}{2} = \frac{N^3+2N^2}{4}$. Таким образом, сложность алгоритма в худшем случае $O(n^3)$, что совершенно не подходит для сортировок больших объемов данных.

1.3.2 Сортировка пузырьком

Сделаем простейшее улучшение алгоритма: поменяв местами два элемента не возвращаемся назад, а идем дальше. Тогда при одном проходе минимальный элемент окажется на своем месте. Очевидно, что алгоритм будет быстрее. Такая сортировка называется сортировкой *пузырьком,* (под пузырьком понимается «самый легкий» элемент, который «поднимается» наверх).

Алгоритм 2: Сортировка пузырьком

Вход: A — массив размерности N
Выход: Отсортированный массив
начало алгоритма
  цикл для $i = 0$ *до* $N-1$ **выполнять**
    цикл для $j = N-1$ *до* i **выполнять**
      если $a[j-1] > a[j]$ **то**
        · меняем их местами;
конец алгоритма

Поскольку обмен идет между соседними элементами и в одном порядке, сортировка пузырьком является устойчивой.

Рассмотрим алгоритм на примере массива, рассмотренного в предыдущем разделе — $\{8, 3, 9, 6, 1, 2, 5\}$. Полужирным шрифтом показаны элементы, расположенные на своих местах и к которым не происходит обращения на следующей итерации.

8	3	9	6	1	2	5
$i = 0$						
8	3	9	1	6	2	5
8	3	1	9	6	2	5
8	1	3	9	6	2	5
1	8	3	9	6	2	5
$i = 1$						
1	8	3	9	2	6	5
1	8	3	2	9	6	5
1	8	2	3	9	6	5
1	2	8	3	9	6	5
$i = 2$						
1	**2**	8	3	9	5	6
1	**2**	8	3	5	9	6
1	**2**	3	8	5	9	6
$i = 3$						
1	**2**	**3**	8	5	6	9
1	**2**	**3**	5	8	6	9
$i = 4$						
1	**2**	**3**	**5**	6	8	9
Результат						
1	**2**	**3**	**5**	**6**	**8**	9

Определяем сложность. Оценим отдельно сравнения и обмены.

Сравнения В данной сортировке количество сравнений одинаково в любом случае. На i-том шаге выполняется i сравнений.

Всего $N - 1$ шаг, следовательно, всего выполняется $\sum_{i=0}^{N-1} i = (N-1)\frac{N}{2} = \frac{N^2-N}{2}$ сравнений.

Обмены Отличие лучшего случая от худшего только в количестве обменов.

Для лучшего случая обменов не происходит.

Для худшего случая на i-том шаге выполняется $i + 1$ обмен (нумерация начинается с нуля). Всего $N - 1$ шаг, следовательно, всего выполняется $\sum_{i=0}^{N-1} i + 1 = (N-1)\frac{N}{2} = \frac{N^2-N}{2}$ обменов.

Таким образом, сложность сортировки пузырьком $O(n^2)$ в любом случае. Но она неэффективна и работает медленнее всего из сортировок подобного типа.

Самый простой способ модификации — остановить алгоритм, если при очередном проходе по массиву не произойдет ни одного обмена.

Попробуем дальше модифицировать алгоритм.

1.3.3 Сортировка перемешиванием

Иногда ее называют еще шейкер-сортировкой (англ. cocktail sort). Идея алгоритма проста. Сначала проходим в прямом порядке, в итоге максимальный элемент встает на свое место. Потом проходим в обратном порядке и ставим на свое место минимальный элемент. Уменьшаем диапазон и повторяем процесс.

Поскольку обмен происходит между соседними элементами в одном направлении (i и $i + 1$), сортировка является устойчивой.

Рассмотрим алгоритм на примере массива, рассмотренного в предыдущем разделе — $\{8, 3, 9, 6, 1, 2, 5\}$. Полужирным шрифтом показаны элементы, расположенные на своих местах, к которым не происходит обращения при следующей итерации.

Алгоритм 3: Сортировка перемешиванием

Вход: A — массив размерности N
Выход: Отсортированный массив
начало алгоритма
- $begin = -1, end = n - 1$;
- **цикл пока** *возможны обмены* **выполнять**
  - считаем, что обменов не было;
  - увеличиваем $begin$;
  - **цикл для** $j = begin$ *до* end **выполнять**
    - **если** $a[j] > a[j+1]$ **то**
      - меняем их местами;
      - отмечаем что обмен был;
  - **если** *обменов не было* **то**
    - завершаем алгоритм
  - считаем, что обменов не было;
  - уменьшаем end;
  - **цикл для** $j = end$ *до* $begin$ **выполнять**
    - **если** $a[j] > a[j+1]$ **то**
      - меняем их местами;
      - отмечаем что обмен был;

конец алгоритма

8	3	9	6	1	2	5
$[0, 6]$						
$begin \longrightarrow end$						
3	8	9	6	1	2	5
3	8	6	9	1	2	5
3	8	6	1	9	2	5
3	8	6	1	2	9	5
3	8	6	1	2	5	9
$end \longrightarrow begin$						
3	8	1	6	2	5	**9**
3	1	8	6	2	5	**9**
1	3	8	6	2	5	**9**
$[1, 5]$						
$begin \longrightarrow end$						
1	3	6	8	2	5	**9**
1	3	6	2	8	5	**9**
1	3	6	2	5	8	**9**
$end \longrightarrow begin$						
1	3	2	6	5	**8**	**9**
1	2	3	6	5	**8**	**9**
$[2, 4]$						
$begin \longrightarrow end$						
1	**2**	3	5	6	**8**	**9**

Определяем сложность. Оценим отдельно сравнения и обмены.

Сравнения В данной сортировке количество сравнений одинаково в любом случае.

- 0 шаг — N сравнений в прямом порядке и $N - 1$ сравнений в обратном (максимум стоит на последнем месте и больше не рассматривается). Всего $a_1 = 2N - 1$ сравнений.
- 1 шаг — $N - 2$ сравнений в прямом порядке (минимум стоит на первом месте и больше не рассматривается) и $N - 3$ в обратном. Всего $a_2 = 2N - 5 = a_1 - 4$ сравнений.
- 2 шаг — $N - 4$ сравнений в прямом порядке и $N - 5$ в обратном. Всего $a_3 = 2N - 9 = a_2 - 4$

Следовательно, на i-том шаге выполняется $2N - 1 - 4i$ сравнений. Всего $\frac{N}{2}$ шагов, следовательно, всего выполняется $\sum\limits_{i=0}^{\frac{N}{2}} 2N - 1 - 4i = \frac{N^2 - N}{2}$ сравнений.

Обмены Отличие лучшего случая от худшего только в количестве обменов.

Для лучшего случая обменов не происходит.

Для худшего случая на каждом сравнении выполняется один обмен. Следовательно, выполняется $\frac{N^2-N}{2}$ обменов.

В среднем случае сортировка перемешиванием работает быстрее, чем сортировка пузырьком.

1.3.4 Сортировка чет-нечет

Будем сравнивать не каждый элемент с последующим, а парами ($a[i]$ с $a[i-1]$) с шагом 2. Для четных i проход начинается с нуля, для нечетных — с единицы. Это модификация сортировки пузырьком для параллельных алгоритмов, поэтому она остается устойчивой.

Алгоритм 4: Сортировка чет—нечет

Вход: A — массив размерности N
Выход: Отсортированный массив
начало алгоритма
- **цикл для** $i = 0$ *до* $N-1$ **выполнять**
  - **если** i *четное* **то**
    - **цикл для** $j = 2$ *до* $N-1$ *с шагом 2* **выполнять**
      - **если** $a[j-1] > a[j]$ **то**
        - · меняем их местами;
  - **иначе**
    - **цикл для** $j = 1$ *до* $N-1$ *с шагом 2* **выполнять**
      - **если** $a[j-1] > a[j]$ **то**
        - · меняем их местами;

конец алгоритма

В примере для каждого шага сначала показано одинаковыми цветами какие именно элементы будут сравниваться. То есть, после каждого описания индекса, строка таблицы просто показывает какие элементы будут сортироваться, а уже следующие строки — это только те итерации алгоритма, где произошли обмены. Они показаны разными цветами.

8	3	9	6	1	2	5
$i = 0$						
8	3	9	6	1	2	5
8	3	9	1	6	2	5
$i = 1$						
8	3	9	1	6	2	5
3	8	9	1	6	2	5
3	8	1	9	6	2	5
3	8	1	9	2	6	5
$i = 2$						
3	8	1	9	2	6	5
3	1	8	9	2	6	5
3	1	8	2	9	6	5
3	1	8	2	9	5	6
$i = 3$						
3	1	8	2	9	5	6
1	3	8	2	9	5	6
1	3	2	8	9	5	6
1	3	2	8	5	9	6
$i = 4$						
1	3	2	8	5	9	6
1	2	3	8	5	9	6
1	2	3	5	8	9	6
1	2	3	5	8	6	9
$i = 5$						
1	2	3	5	8	6	9
1	2	3	5	6	8	9

Оценим сложность. Также, как и в случае сортировки пузырьком, разделим сравнения и обмены.

Сравнения Как и в случае пузырьковой сортировки для сравнений нет разницы для лучшего, худшего и среднего случая. Во всех случаях для i-того происходит $\frac{N}{2}$ сравнений. Всего алгоритм содержит $N-1$ шагов. Следовательно, всего происходит $\frac{N(N-1)}{2}$ сравнений.

Обмены оценим для лучшего и худшего случая отдельно.

Лучший случай Для полностью отсортированного массива обменов не происходит.

Худший случай Для отсортированного в обратном порядке массива обмен происходит после каждого сравнения, следовательно, всего происходит $\frac{N(N-1)}{2}$ обменов.

Таким образом, в худшем и лучшем случаях сортировка чет—нечет демонстрирует тоже число обменов и сравнений, что и сортировка пузырьком. Сложность во всех случаях $O(n^2)$.

Сортировка чет—нечет работает быстрее сортировки пузырьком в случае параллельного алгоритма, так как четные и нечетные индексы можно сортировать независимо.

1.3.5 Сортировка расческой

В предыдущем разделе элементы сравнивались парами. Другой вариант модификации алгоритма заключается в том, чтобы сравнивать не соседние элементы, а элементы, находящиеся на каком-то расстоянии. И потом это расстояние сокращать, до тех пор, пока оно не станет равно единицы. Последним проходом досортировать пузырьковой сортировкой. Оказывается, что такая модификация существенно ускоряет время работы алгоритма.

Расстояние между элементами выбирается равным размеру массива, разделенного на фактор уменьшения f. На каждой следующей итерации расстояние уменьшается в f раз до тех пор, пока не дойдет до 1.

Теоретическим и опытным путем было доказано, что $f = \frac{1}{1-e^{-\varphi}} \approx 1.247\ldots$, где $\varphi = \frac{1+\sqrt{5}}{2}$ — «золотое сечение»

Алгоритм 5: Сортировка расческой

Вход: A — массив размерности N
Выход: Отсортированный массив
начало алгоритма
- $\cdot\ gap = N/1.247$;
- **цикл пока** $gap >= 1$ **выполнять**
  - **если** $gap < 1$ **то**
    - $gap = 1$
  - **цикл для** $j = 0$ *до* $N - gap$ **выполнять**
    - **если** $a[j] > a[j + gap]$ **то**
      - $\cdot$ меняем их местами;
  - $\cdot\ gap/ = 1.247$;

конец алгоритма

В начале каждого шага итерации одинаковыми цветами показаны сравниваемые элементы. Далее различными цветами показаны только обмены.

8	3	9	6	1	2	5
шаг $= 5$						
8	3	9	6	1	2	5
2	3	9	6	1	8	5
шаг $= 4$						
2	3	9	6	1	8	5
1	3	9	6	2	8	5
1	3	5	6	2	8	9
шаг $= 3$						
1	3	5	6	2	8	9
1	2	5	6	3	8	9
шаг $= 2$						
1	2	5	6	3	8	9
1	2	3	6	5	8	9
шаг $= 2$						
1	2	3	6	5	8	9
1	2	3	5	6	8	9

В данном случае сложно придумать худший случай, так как оказывается, что для отсортированного в обратном порядке массива необходимо сделать меньше обменов, чем для случайного массива. Будем под худшем случаем понимать массив, такой что каждое сравнение сопровождается обменом.

- 1 шаг — число сравнений $N - \lfloor \frac{N}{f} \rfloor$.
- 2 шаг — число сравнений $N - \lfloor \frac{N}{f^2} \rfloor$.
- i-тый шаг — число сравнений $N - \lfloor \frac{N}{f^i} \rfloor$.
- k-тый шаг — число сравнений $N - \lfloor \frac{N}{f^k} \rfloor = N - 1$.

То есть, выполняется k шагов. Определяем из выражения для геометрической прогрессии: $b_k = b_1 q^{k-1} \to k = 1 + \log_q \frac{b_k}{b_1}$. В данном случае $b_1 = N$, $b_n = 1$, $q = \frac{1}{f}$. Тогда $k \approx 5 \ln N + 1$. Всего сравнений $N \sum\limits_{i=0}^{5 \ln N} 1 - \frac{1}{f^i} \approx 5N \ln N - 4N$. (так как $\sum \frac{1}{f^i} \to 5$ при $N \to \infty$.)

Получается, что сложность $O(N^2)$, но в реальности сортировка стремится к $O(N \log N)$. Является самой быстрой из сортировок за время $O(N^2)$. Основной недостаток — это неустойчивая сортировка.

1.3.6 Гномья сортировка

Описанная не так давно голландским математиком Диком Груном сортировка названа по сортировке цветочных горшков гномами:

> Садовый гном сортирует линию цветочных горшков. Смотрит на следующий и предыдущий горшки: если они в правильном порядке, он шагает на один шаг вперед, иначе меняет их местами и шагает на один шаг назад. Граничные условия: если нет предыдущего горшка, гном шагает на шаг вперед; если нет следующего, горшки отсортированы.

Алгоритм 6: Гномья сортировка

Вход: A — массив размерности N
Выход: Отсортированный массив
начало алгоритма
- $i = 1$;
- **цикл пока** $i < N$ **выполнять**
  - **если** *начало списка* **то**
    - увеличиваем i;
  - **если** $a[i-1] <= a[i]$ **то**
    - увеличиваем i;
  - **иначе**
    - меняем местами;
    - уменьшаем i;

конец алгоритма

8	3	9	6	1	2	5
3	8	9	6	1	2	5
3	8	6	9	1	2	5
3	6	8	9	1	2	5
3	6	8	1	9	2	5
3	6	1	8	9	2	5
3	1	6	8	9	2	5
1	3	6	8	9	2	5
1	3	6	8	2	9	5
1	3	6	2	8	9	5
1	3	2	6	8	9	5
1	2	3	6	8	9	5
1	2	3	6	8	5	9
1	2	3	6	5	8	9
1	2	3	5	6	8	9

Оценим сложность. В худшем случае сортируем отсортированный в обратном порядке массив. В лучшем сортируем отсортированный массив.

Лучший случай — один проход по циклу для сравнений. Обменов нет. Сложность $O(n)$.

Худший случай — на каждом шаге делаем i сравнений в прямом порядке и i сравнений и i обменов. Всего N шагов. Следовательно, выполняем $2\sum\limits_{i=0}^{N} i = 2\frac{N(N-1)}{2} = N^2 - N$ сравнений и $\frac{N^2-N}{2}$. Сложность $O(n^2)$.

1.3.7 Быстрая сортировка

Рассмотрим теперь идею модификации, которая была предложена Хоаром в 1960 году. Алгоритм, предложенный им, превратил одну из самых медленных сортировок в одну из самых быстрых. Единственное, что она стала неустойчивой, так как происходит обмен не между соседними элементами, а это приводит к неустойчивости.

Общая идея алгоритма такова:

1. Выбираем опорный элемент.

2. Меняем местами элементы так, чтобы слева были меньшие опорного, справа — большие.

3. Рекурсивно вызываем функцию для подмассивов.

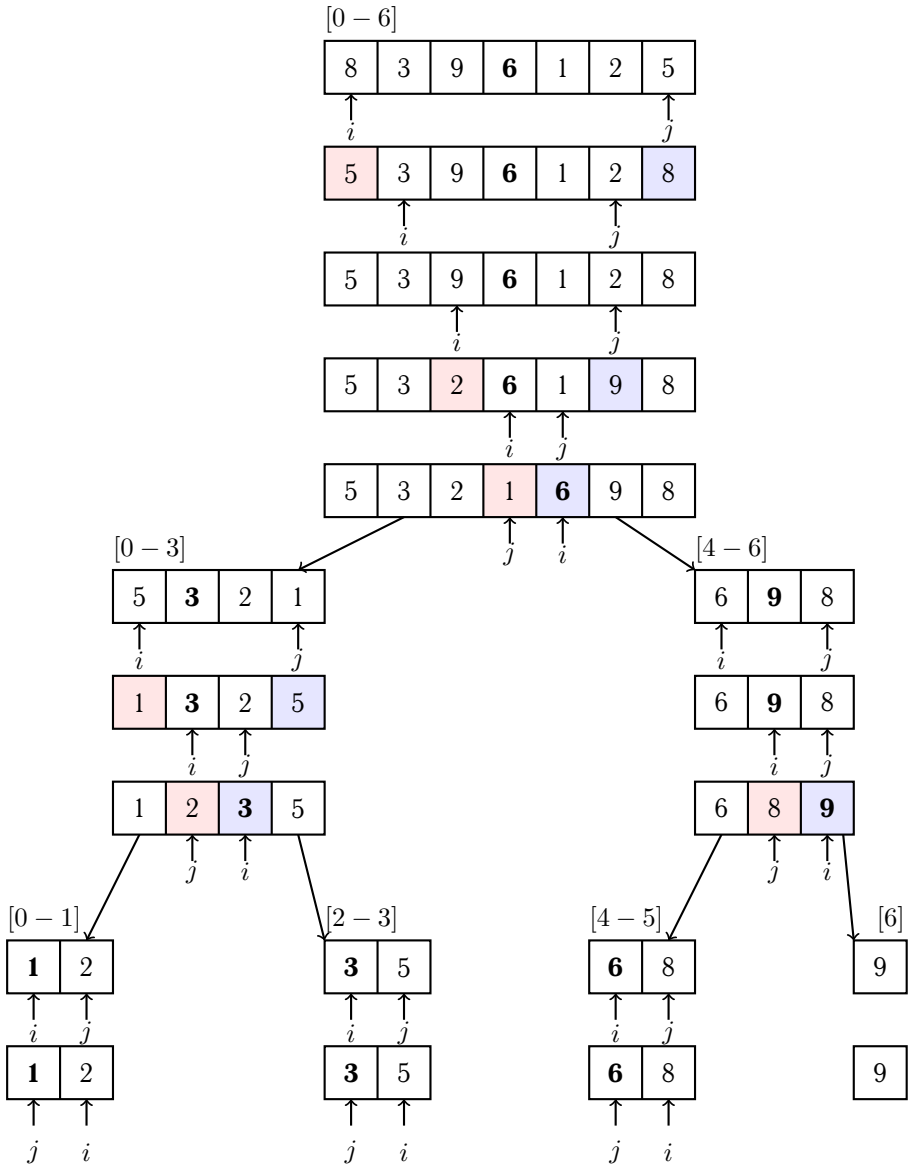
Рассмотрим данный алгоритм подробнее:

Алгоритм 7: Быстрая сортировка

Вход: A — массив, L — левая граница массива, R— правая граница массива
Выход: Отсортированный массив
начало алгоритма
- $i = L$;
- $j = R$;
- выбирается опорный элемент;
- **цикл пока** $i <= j$ **выполнять**
  - **цикл пока** $a[i] <$ *опорного* **выполнять**
    - увеличиваем i;
  - **цикл пока** $a[j] >$ *опорного* **выполнять**
    - уменьшаем j;
  - **если** $i <= j$ **то**
    - меняем местами $a[i]$ и $a[j]$;
    - уменьшаем j;
    - увеличиваем i;
- **если** *левый подмассив содержит больше одного элемента* **то**
  - вызываем функцию для $[L, j]$;
- **если** *правый подмассив содержит больше одного элемента* **то**
  - вызываем функцию для $[i, R]$;

конец алгоритма

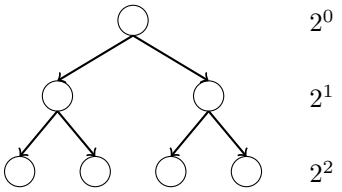
Рассмотрим пример сортировки. В качестве опорного элемента выбираем элемент, расположенный в середине массива. В данном случае будут продемонстрированы все шаги, а не только обмены. Полужирным шрифтом выделен опорный элемент, красным и синим цветами — обмены.



Оценим сложность. Число обменов оценить практически невозможно для худшего и среднего случаев. Для лучшего случая обменов не будет.

Число сравнений можно оценить только приблизительно. Как видно на рисунке, рекурсивный вызов похож на структуру дерево. Для каждого подмассива делается k сравнений, где k — размер подмассива. На каждом уровне дерева происходит N сравнений (один подмассив имеет размер $\frac{N}{i}$, а на каждом уровне i подмассивов). Осталось определить сколько уровней у дерева.

Лучший случай — отсортированный массив. Если выбрать средний элемент в качестве опорного, то массив будет делиться примерно пополам. Получаем сбалансированное бинарное дерево (у каждого узла только два потомка, количество узлов в левом поддереве и в правом поддереве отличается не более чем на единицу.)



Как видно из рисунка ниже, полностью заполненное дерево содержит $2^k - 1$ узлов, где k — высота дерева (количество уровней). Тогда $N = 2^k - 1 \rightarrow k = \log_2 N + 1$. В нашем примере получилось именно три уровня, как и следует из формулы.

Тогда число сравнений $O(N \log N)$.

Так как перейти от одного основания логарифма к другому можно по формуле $\log_a b = \frac{\log_c b}{\log_c a}$. В знаменателе константа, которой при оценке сложности пренебрегаем, поэтому обычно при оценке не пишут основание алгоритма.

Худший случай — массив подбираем специальным образом. Например, при выборе в качестве опорного элемента середины

массива используется следующий алгоритм:

Алгоритм 8: Специальный подбор массива

Вход: A — массив, размерности N
Выход: Измененный массив
начало алгоритма
  · заполняем массив числами от 1 до N;
  цикл для $i = 0$ *до* $N - 1$ **выполнять**
    · меняем местами $a[i]$ и $a[i/2]$;
конец алгоритма

Например, для $N = 9$ полученный массив $\{2, 4, 6, 8, 9, 5, 3, 7, 1\}$. Можно убедиться (проверьте самостоятельно), что в таком массиве в качестве опорного (середина массива) всегда будет выбираться максимальный элемент. Тогда на i-том шаге массив будет делится на две части: массив размером $i - 1$ и массив, состоящий из одного элемента. Очевидно, что таких шагов будет N. Следовательно, сложность алгоритма будет $O(N^2)$.

Оценка сложности алгоритма в среднем случае будет проведена в следующих курсах. Там же будет показано, что все рекурсивные вызовы подобные данному в среднем дают сложность $O(N \log N)$. Там же будет показано, что наименьшая оценка сложности для обменных сортировок $O(N \log N)$. Из всех сортировок в среднем быстрая сортировка действительно является самой быстрой.

Рекурсивный вызов требует затрат памяти. В среднем $O(N \log N)$. Модификации быстрой сортировки могут приводить в $O(\log N)$ дополнительной памяти. Например, рекурсивно вызывать только подмассив меньшей длины, а больший сортировать в цикле. В данном случае глубина рекурсии будет гарантировано $\log N$, а в лучшем случае можно пройти по циклу только один раз и сложность будет $O(N)$.

Можно выбрать различные варианты выбора опорного элемента:

1. Выбрать первый, последний или средний элементы массива. В таком случае возможно подобрать массив таким образом, чтобы сложность была $O(N^2)$.

2. Выбрать медиану — среднее арифметическое из первого, последнего и среднего элементов массива, причем полученный опорный элемент не обязательно должен находиться в массиве.

3. Выбрать на каждом шаге в качестве опорного случайный элемент массива. В таком случае вероятность худшего случая очень мала.

Быстрая сортировка легко применяется к двусвязным спискам и другим структурам, допускающим проход как в прямом, так и в обратном порядке.

Также быстрая сортировка удобна для параллельных алгоритмов.

1.4 Сортировки вставками

Рассмотрим теперь алгоритм немного другой природы. В жизни, если надо отсортировать какие-то бумаги, большинство воспользуется именно сортировкой вставками.

Общая идея алгоритма такова: подмассив длины i уже отсортирован. $i + 1$-вый элемент вставляют на нужное место в отсортированном массиве.

Рассмотрим два алгоритма сортировки вставками: собственно алгоритм сортировки вставками и его модификацию, называемую алгоритм сортировки Шелла.

1.4.1 Сортировка вставками

Можно сказать, что это модификация гномьей сортировки. Гном идет вперед, пока не найдет неправильно стоящие горшки, запоминает на каком горшке он остановился, меняет горшки местами и идет назад, пока не поставит горшок на место. Однако дальше идет не на шаг вперед, а сразу начинает с горшка, который он запомнил.

В примере показаны только итерации с обменами. Разными цветами показаны элементы, которые обменяли.

Алгоритм 9: Сортировка вставками

Вход: A — массив размерности N
Выход: Отсортированный массив
начало алгоритма
  цикл для $i = 1$ *до* $N - 1$ **выполнять**
    · $j = i$;
    цикл пока $j > 0$ *и* $a[j] < a[j-1]$ **выполнять**
      · меняем их местами;
      · уменьшаем j;
конец алгоритма

8	3	9	6	1	2	5
$i = 1$						
3	8	9	6	1	2	5
$i = 2$						
$i = 3$						
3	8	6	9	1	2	5
3	6	8	9	1	2	5
$i = 4$						
3	6	8	1	9	2	5
3	6	1	8	9	2	5
3	1	6	8	9	2	5
1	3	6	8	9	2	5
$i = 5$						
1	3	6	8	2	9	5
1	3	6	2	8	9	5
1	3	2	6	8	9	5
1	2	3	6	8	9	5
$i = 6$						
1	2	3	6	8	5	9
1	2	3	6	5	8	9
1	2	3	5	6	8	9

Оценим сложность. Число обменов и сравнений одинаково.

Лучший случай. Рассматриваем отсортированный массив. Условие цикла `while` сразу возвращает `false`. Следовательно происходит одно сравнение. Всего таких шагов $N - 1$. Следовательно, сложность алгоритма $O(N)$.

Худший случай. Рассматриваем отсортированный в обратном порядке массив. В данном случае на i-том шаге происходит i сравнений. Всего шагов — $N-1$. Тогда, всего сравнений $\sum_{i=0}^{N-1} i = \frac{N(N-1)}{2} = \frac{N^2-N}{2}$. Следовательно, сложность алгоритма $O(N^2)$.

Алгоритм сортировки вставками один из самых быстрых из «квадратичных» сортировок. Для малых $N < 100$ лучше использовать его.

Иногда алгоритмы типа быстрой сортировки модифицируют следующим образом: разбивают на подмассивы до тех пор, пока размер подмассивов не станет меньше 100. А дальше используют сортировку вставками.

1.4.2 Сортировка Шелла

Также как и в случае сортировки расческой Шелл предложил сравнивать не соседние элементы, а стоящие друг от друга на каком-то расстоянии. Это ускоряет достаточно ускоряет сортировку. Сам Шелл предложил следующую последовательность шагов: $\frac{N}{2}, \frac{N}{4}, \ldots, 1$. Оценить сложность достаточно сложно. В худшем случае сортировка дает сложность $O(N^2)$.

Впоследствии были получены последовательности шагов, которые позволяют довести сложность до $O(N^{\frac{4}{3}})$:

1. $d_i = 2^i - 1 \leqslant N$ или $d_i = 3^i - 1 \leqslant N$ — сложность $O(N^{\frac{3}{2}})$. (Хиббард)
2. $$\begin{cases} d_i = 9 \times 2^i - 9 \times 2^{\frac{i}{2}} + 1, & \text{если } i \text{ четное} \\ d_i = 8 \times 2^i - 6 \times 2^{fraci+12} + 1, & \text{если } i \text{ нечетное} \end{cases}$$

   В лучшем случае сложность $O(N^{\frac{7}{6}})$, в худшем — $O(N^{\frac{4}{3}})$. Предложена Седжвиком.

3. Предложена Праттом $d_i = 2^i \times 3^j \leqslant \frac{N}{2}$. Сложность алгоритма составляет $O(N(\log n)^2)$.

В алгоритме шаг используется от максимального к единице.

Алгоритм 10: Сортировка Шелла

Вход: A — массив размерности N
Выход: Отсортированный массив
начало алгоритма
- · Определяем шаг;
- **цикл пока** $step > 1$ **выполнять**
  - **цикл для** $i = 0$ *до* $N - step$ **выполнять**
    - · $j = i$;
    - **цикл пока** $j >= 0$ *и* $a[j] > a[j + step]$ **выполнять**
      - · меняем их местами;
      - · уменьшаем j на шаг;
  - · переход к следующему шагу;

конец алгоритма

В примере для каждого шага сначала выделены одинаковыми цветами показаны элементы, которые сравниваются на данном шаге. Далее разными цветами показаны обмены.

В качестве шага выбрано $\frac{N}{2}$. Для лучшего примера рассмотрим массив из 8 элементов: $\{8, 3, 9, 6, 1, 2, 5, 7.\}$:

8	3	9	6	1	2	5	7
$step = 4$							
8	3	9	6	1	2	5	7
1	3	9	6	8	2	5	7
1	2	9	6	8	3	5	7
1	2	5	6	8	3	9	7
$step = 2$							
1	2	5	6	8	3	9	7
1	2	5	3	8	6	9	7
$step = 1$							
1	2	5	3	8	6	9	7
1	2	3	5	8	6	9	7
1	2	3	5	6	8	9	7
1	2	3	5	6	8	7	9
1	2	3	5	6	7	8	9

1.5 Сортировка выбором

Другой тип сортировки состоит в том, что в подмассиве $[i + 1, N]$ находится минимальный элемент и ставится на i-тую позицию.

Существует собственно сортировка выбором и пирамидальная сортировка.

1.5.1 Сортировка выбором

Бывает как устойчивая, так и неустойчивая сортировка.

Рассмотрим сначала алгоритм неустойчивой сортировки.

Алгоритм 11: Неустойчивая сортировка выбором

Вход: A — массив размерности N
Выход: Отсортированный массив
начало алгоритма
- **цикл для** $i = 0$ *до* N **выполнять**
  - · находим минимум из элементов массива от $[i + 1]$ до $[N - 1]$;
  - · меняем местами i-тый и минимальный элементы;

конец алгоритма

В примере различными цветами показаны только обмены. Полужирным шрифтом показаны элементы, которые уже отсортированы.

8	3	9	6	1	2	5
1	3	9	6	8	2	5
1	2	9	6	8	3	5
1	**2**	3	6	8	9	5
1	**2**	**3**	5	8	9	6
1	**2**	**3**	**5**	6	9	8
1	**2**	**3**	**5**	**6**	8	9

Оценим сложность. Число сравнений и обменов во всех случаях одинаково. Для нахождения минимума надо пройти по всему массиву вне зависимости от того, отсортирован массив или нет. Если массив уже отсортирован, то просто будет замена элементов, находящихся в одном месте. Можно добавить условие, что если минимальным является элемент, находящийся на i-том месте, то обменов не производить, но тогда добавится еще операция сравнения, следовательно, выигрыша по времени не произойдет.

Для того, чтобы найти на i-том шаге минимум, необходимо провести $N - i$ сравнений. Всего шагов $N - 1$. Тогда общее число сравнений $\sum\limits_{i=0}^{N-1} N - i = \frac{(N-1)(N+1)}{2} = \frac{N^2-1}{2}$. Следовательно, сложность $O(N^2)$.

Сортировка неустойчива, так как меняются не соседние элементы. Для того, чтобы ее сделать устойчивой, надо не менять местами минимальный и i-тый элементы, а вставлять минимальный элемент на i-тое место (соответственно, сначала удаляя его из массива). В случае массива это приводит к дополнительным временным затратам, так как вставка и удаление из массива требует в среднем $O(N)$ времени. А вот в случае со списками, где вставка и удаление требует $O(1)$, устойчивая быстрая сортировка затрачивает тоже время, что и неустойчивая.

Алгоритм 12: Устойчивая сортировка выбором

Вход: A — массив размерности N
Выход: Отсортированный массив
начало алгоритма
  цикл для $i = 0$ *до* N **выполнять**
    · находим минимум из элементов массива от $[i + 1]$ до $[N]$;
    · удаляем минимальный элемент;
    · вставляем минимальный элемент на i-тое место;
конец алгоритма

1.5.2 Пирамидальная сортировка

При использовании пирамидальной сортировки представляем массив в виде специальной структуры, называемой кучей (англ. heap).

Куча, или основанная на ее основе структура, называемая очередь с приоритетами, используется во многих алгоритмах.

Куча

Куча — это специальное бинарное дерево, обладающее следующими свойствами:

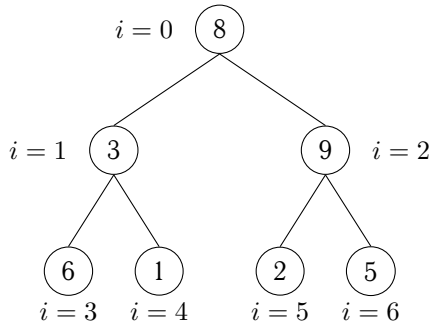
1. Дерево почти полное, то есть, заполнено на всех уровнях, кроме последнего. На последнем уровне элементы заполняются слева направо, пока не закончатся элементы.

2. Значение узла всегда больше значений его потомков. Следовательно, корнем дерева будет будет максимальный элемент.

Как уже говорилось при описании быстрой сортировки, высота почти полного бинарного дерева равна $\log_2 N + 1$.

Куча можно описать с помощью массива. Потомками i-того узла являются элементы с индексами $2i + 1$ и $2i + 2$ (при условии нумерации с нуля).

Построим кучу на примере массива из 7 элементов: $\{8, 3, 9, 6, 1, 2, 5\}$. Сначала представим этот массив в виде бинарного дерева:



Вершиной дерева является нулевой элемент массива. На первом уровне слева направо расположены первый и второй элементы на массива. На втором уровне слева направо 3— 6 элементы и т. д., пока не закончится массив.

Теперь надо перестроить дерево так, чтобы оно удовлетворяло основному свойству кучи: значение родителя должно быть больше значений потомков.

Рассмотрим функцию «просеивания вниз», которая заключается в том, что i-тый элемент сравнивается со своими детьми и, если кто-то из детей имеет большее значение, родитель и ребенок меняются местами.

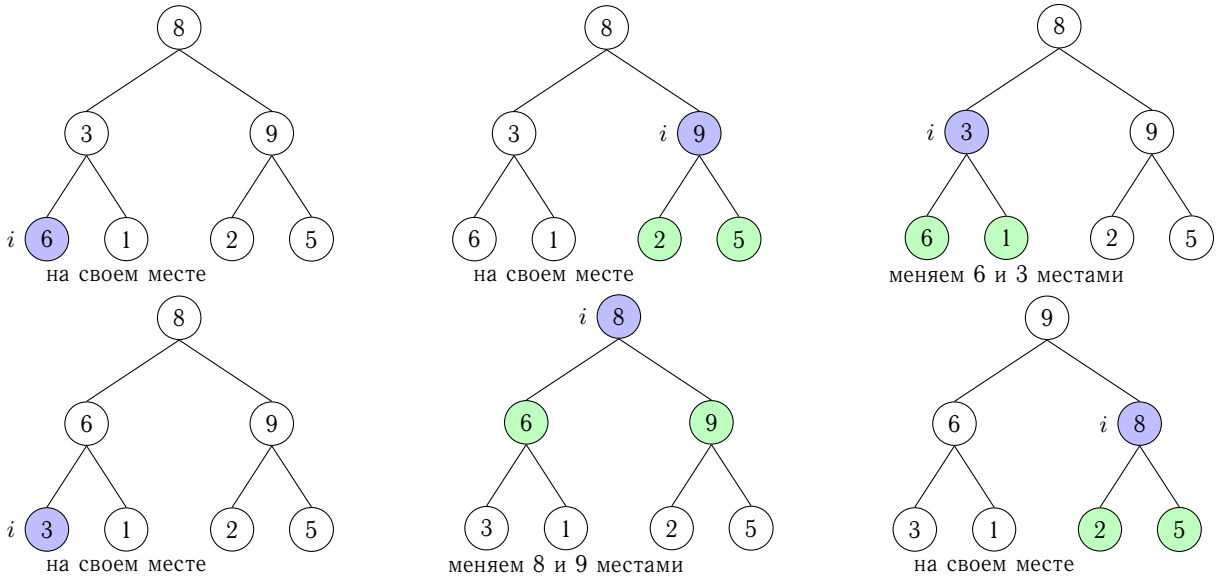
Алгоритм 13: «Просеивание» i-ого элемента в куче

Вход: A — массив, i — номер просеиваемого элемента, N — размер массива
Выход: Куча
начало алгоритма
- · полагаем, что i-тый элемент является максимальным;
- **цикл пока** `true` **выполнять**
  - **если** ∃ *левый ребенок для i-ого элемента* **то**
    - · находим максимум между ним и родителем;
  - **если** ∃ *правый ребенок для i-ого элемента* **то**
    - · находим максимум между ним и родителем;
  - **если** *максимум — это родитель* **то**
    - · прекращаем цикл;
  - **иначе**
    - · меняем местами максимум и родителя;
    - · $i = max$;

конец алгоритма

Очевидно, что наибольшее количество шагов необходимо выполнить для того, чтобы поместить элемент из нулевого уровня в максимальный. Как уже говорилось выше, высота дерева равна $\log N$. Всего элементов N. Следовательно, максимальное количество шагов, которое необходимо сделать, чтобы получить кучу, $N \log N$.

Для того, чтобы построить кучу, надо вызывать рассмотренную функцию в цикле от $\frac{N}{2}$ до 0:



В итоге, получили следующий массив: $\{9, 6, 8, 3, 1, 2, 5\}$. Видно, что нулевым элементом является максимальный элемент массива.

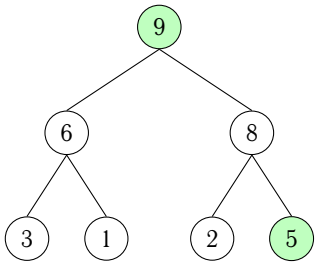
Пирамидальная сортировка

На вершине кучи всегда находится максимальный элемент. Следовательно, алгоритм пирамидальной сортировки заключается в следующем: меняем местами первый и последний элемент и перестраиваем пирамиду, уменьшая количество элементов на 1.

Алгоритм 14: Пирамидальная сортировка

Вход: A — массив, N — размер массива
Выход: Отсортированный массив
начало алгоритма
  цикл для $i = \frac{N}{2}$ *до 0* **выполнять**
    · строим пирамиду, вызывая функцию просеивания от (A, i, N);
  цикл для $i = N - 1$ *до 1* **выполнять**
    · меняем местами $a[0]$ и $a[i]$;
    · перестраиваем пирамиду, вызывая функцию просеивания от $(A, 0, i)$;
конец алгоритма

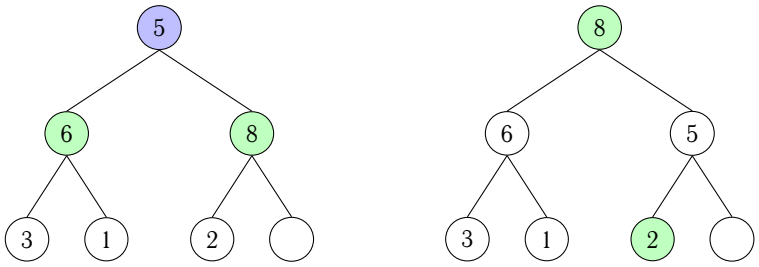
Пирамида выше уже построена. Рассмотрим на этом примере пирамидальную сортировку:



Массив выглядит следующим образом:

9	6	8	3	1	2	5

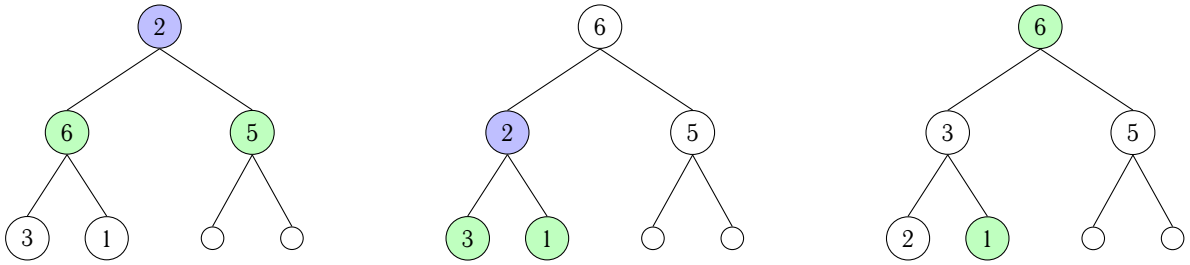
Делаем замену и перестраиваем пирамиду:



Массив выглядит следующим образом:

8	6	5	3	1	2	**9**

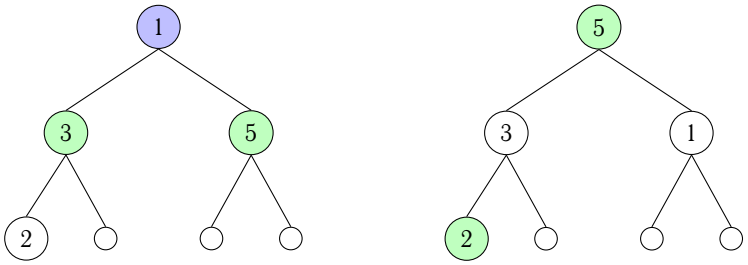
Делаем замену и перестраиваем пирамиду:



Массив выглядит следующим образом:

6	3	5	2	1	**8**	**9**

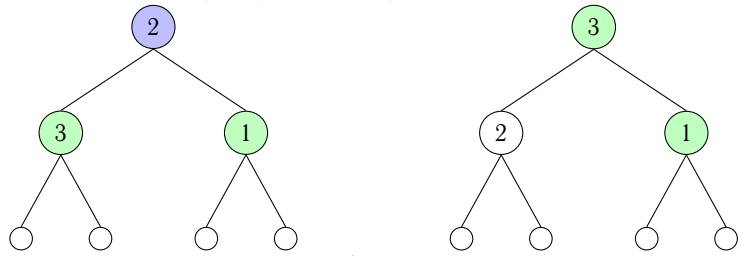
Делаем замену и перестраиваем пирамиду:



Массив выглядит следующим образом:

5	3	1	2	**6**	**8**	**9**

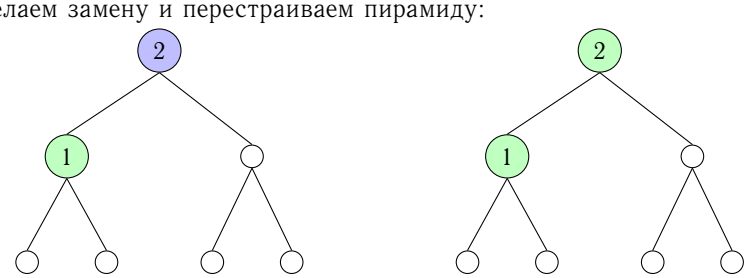
Делаем замену и перестраиваем пирамиду:



Массив выглядит следующим образом:

3	2	1	**5**	**6**	**8**	**9**

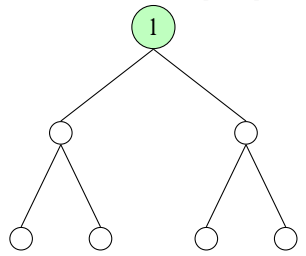
Делаем замену и перестраиваем пирамиду:



Массив выглядит следующим образом:

2	1	**3**	**5**	**6**	**8**	**9**

Делаем замену и перестраиваем пирамиду:



Массив отсортирован:

1	**2**	**3**	**5**	**6**	**8**	**9**

Оценим сложность алгоритма: выше уже говорилось, что функция просеивания требует $O(\log N)$. Всего шагов N. Следовательно, сложность алгоритма $O(N \log N)$.

Пирамидальная сортировка работает медленнее быстрой, ее нельзя распараллелить, она неустойчивая.

1.6 Сортировка слиянием

Идея слияния (англ. merge) используется очень часто. Если есть две отсортированные последовательности, то результатом слияния должна быть отсортированная последовательность. Алгоритм слияния можно объяснить на примере двух колод карт. Обе колоды отсортированы, лежат рубашками вниз. Берется верхние карты из каждой колоды, сравниваются, меньшая откладывается, берется следующая карта из этой колоды. Следовательно, требуется N шагов для алгоритма слияния. В алгоритме используется один массив, и сливаются отсортированные подмассивы $[l, m]$ и $[m+1, r]$, где l и r — левая и правая границы массива.

Алгоритм 15: Слияние двух отсортированных подмассивов

Вход: A — массив, l — левая граница массива, r — правая граница массива, m — место разбиения на подмассивы. Подмассивы $[l, m]$ и $[m+1, r]$ являются отсортированными.
Выход: Отсортированный подмассив $[l, r]$
начало алгоритма
- **если** $l \geqslant r$ *или* $m < l$ *или* $m > r$ **то**
  - · прекращаем работу алгоритма;
- **если** *в массиве 2 элемента и они не отсортированы* **то**
  - · меняем их местами;
  - · прекращаем работу алгоритма;
- · Создаем дополнительный массив buf размерностью $r - l + 1$;
- · $cur = 0$;
- · $i = l, j = m + 1$;
- **цикл пока** $r - l + 1 \neq cur$ **выполнять**
  - **если** $i > m$ **то**
    - · дописываем в buf оставшиеся элементы от j до r ;
  - **иначе если** $j > r$ **то**
    - · дописываем в buf оставшиеся элементы от i до m ;
  - **иначе если** $a[i] > a[j]$ **то**
    - · записываем в buf $a[j]$;
    - · увеличиваем j;
  - **иначе**
    - · записываем в buf $a[i]$;
    - · увеличиваем i;
- · записываем массив buf в начальный массив A, начиная с индекса l;

конец алгоритма

Рассмотрим на примере. Есть два отсортированных массива: $\{2, 5, 9\}$ и $\{1, 3, 4\}$. У первого массива красным цветом показан счетчик, у второго — синим. Зеленым цветом показан счетчик результирующего массива. В результирующим массив записывается минимальный из этих элементов.

2	5	9		1	3	4		1					
2	5	9		1	3	4		1	2				
2	5	9		1	3	4		1	2	3			
2	5	9		1	3	4		1	2	3	4		
2	5	9		1	3	4		1	2	3	4	5	9

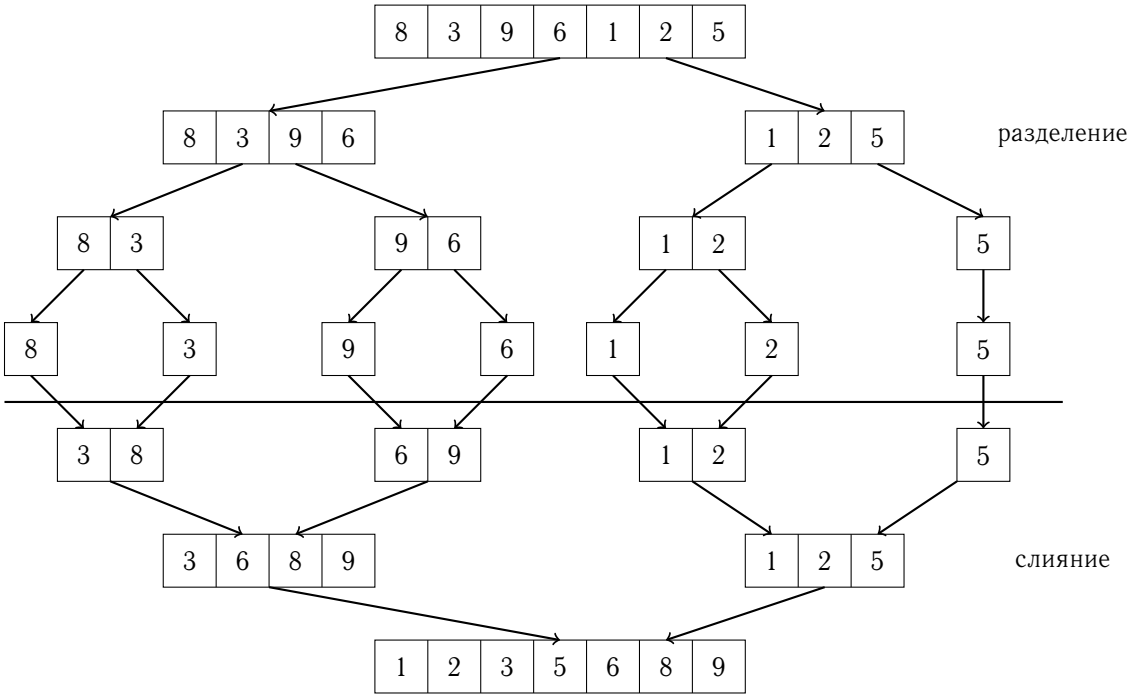
Слияние отсортированных массивов требует $O(N)$ времени. Осталось определить, как получить отсортированные подпоследовательности. Простой вариант — делить рекурсивно массив пополам до тех пока, пока в подмассивах не останется по одному элементу, а потом их последовательно «сливать».

Алгоритм 16: Сортировка слиянием

Вход: A — массив, l — левая граница массива, r — правая граница массива
Выход: Отсортированный массив
начало алгоритма
- **если** $l \geqslant r$ **то**
  - · прекращаем работу алгоритма;
- · $m = \frac{l+r}{2}$;
- · вызываем данную функцию от (A, l, m);
- · вызываем данную функцию от $(A, m + 1, r)$;
- · вызываем функцию слияния от (A, l, r, m);

конец алгоритма

Вызывается сортировка при $l = 0, r = N - 1$. Рассмотрим тот же пример, что и остальных случаях. Подробно слияние отдельных подмассивов рассматриваться не будет, только результат слияния.



Как видно из рисунка, при разделении массива на подмассивы получается почти полное бинарное дерево, следовательно, высота такого дерева $\log N$. Если на каждом шаге требуется N сравнений, то сложность алгоритма $N \log N$.

Сортировка слиянием работает медленнее, чем быстрая сортировка. Из достоинств— возможность работы со структурами с произвольным доступом, возможность работать на параллельных процессорах, устойчива. Из недостатков — требует дополнительной памяти $O(N)$.

1.7 Сортировки за линейное время

В некоторых случаях, когда необходимо сортировать элементы, удовлетворяющие определенным условиям, можно воспользоваться специальными алгоритмами сортировки, которые могут работать за более быстрое время, чем $N \log N$.

1.7.1 Сортировка подсчетом

Если известно, что элементы массива находятся в диапазоне $[0, k]$. Всего элементов N и $k < N$, то можно воспользоваться сортировкой подсчетом. Идея алгоритма состоит в следующем: так как $k < N$, то в массиве обязательно будут повторяющиеся элементы. Надо подсчитать количество повторяющихся элементов.

Алгоритм 17: Сортировка подсчетом

Вход: A — массив, N — размер массива, элементы массива находятся в диапазоне $[0, k]$
Выход: Отсортированный массив
начало алгоритма
- · создаем дополнительный массив B размерностью k;
- · заполняем его нулями;
- **цикл для** $i = 0$ *до* N **выполнять**
  - · увеличиваем $B[A[i]]$;
- **цикл для** $i = 0$ *до* k **выполнять**
  - · последовательно выводим $B[i]$ элементов i;

конец алгоритма

Например, есть массив:

3	0	1	2	3	1	4	3	3	1	0	4	3	1	0

Число элементов в массиве $N = 15$. Массив содержит элементы из диапазона $[0, 4]$, т. е., $k = 5$. Создаем дополнительный массив:

0	0	0	0	0

Считаем сколько элементов, равных 0, в массиве и записываем в нулевую ячейку дополнительного массива и т. д.

3	4	1	5	2

Теперь сначала записываем три нуля, потом четыре единицы, одну двойку и т. д.

0	0	0	1	1	1	1	2	3	3	3	3	3	4	4

Массив отсортирован. Оценим сложность. Сначала делаем N шагов для того, чтобы подсчитать количество одинаковых элементов. Потом делаем $k + N$ шагов для того, чтобы записать новые элементы в массив. Следовательно, асимптотика $O(N + k)$. Требуется дополнительная память $O(k)$. Сортировку имеет смысл применять, когда $k \ll N$, поэтому можно считать, что сложность алгоритма $O(N)$. Если диапазон заранее неизвестен, то за линейное время можно найти минимум и максимум массива.

В случае, если в качестве сортируемых элементов выступает сложная структура, можно вместо одного дополнительного массива создать k дополнительных векторов или списков, куда будут записываться элементы с одинаковыми ключами, а потом объединить эти вектора.

Рассмотрим структуру, содержащую два поля: число и символ. Массив содержит $N = 15$ элементов. Будем сортировать по первому полю. Числа принадлежат диапазону $[0, 4]$.

$\{3,a\}$	$\{0,b\}$	$\{1,a\}$	$\{2,s\}$	$\{3,d\}$	$\{1,e\}$	$\{4,b\}$	$\{3,x\}$	$\{3,h\}$	$\{1,f\}$	$\{0,h\}$	$\{4,k\}$	$\{3,m\}$	$\{1,r\}$	$\{0,v\}$

Создаем пять дополнительных векторов (или списков), т. е., структуру, которая не требует выделения памяти сразу, так как неизвестна их длина. Записываем в i-тый вектор элементы, первое поле которых равно i.

$buf[0]$	$\{0,b\}$	$\{0,h\}$	$\{0,v\}$		
$buf[1]$	$\{1,a\}$	$\{1,e\}$	$\{1,f\}$	$\{1,r\}$	
$buf[2]$	$\{2,s\}$				
$buf[3]$	$\{3,a\}$	$\{3,d\}$	$\{3,x\}$	$\{3,h\}$	$\{3,m\}$
$buf[4]$	$\{4,b\}$	$\{4,k\}$			

Объединяем вектора.

$\{0,b\}$	$\{0,h\}$	$\{0,v\}$	$\{1,a\}$	$\{1,e\}$	$\{1,f\}$	$\{1,r\}$	$\{2,s\}$	$\{3,a\}$	$\{3,d\}$	$\{3,x\}$	$\{3,h\}$	$\{3,m\}$	$\{4,b\}$	$\{4,k\}$

Сортировка устойчивая.

1.7.2 Блочная сортировка

Если известен диапазон сортируемых данных, то их можно разбить на блоки по какому-нибудь признаку, а уже внутри блоков либо вызывать эту сортировку рекурсивно, либо сортировать какой-то элементарной сортировкой (например, вставками, которая для почти отсортированного массива имеет сложность $O(N)$.)

Представим, что есть какой-нибудь набор карточек (бумажных) и их надо отсортировать по алфавиту. Самый простой алгоритм: разбить сначала карточки по группам, например, начинающиеся на буквы А—Г, Д—З и т. д. Потом полученные группы сортируются вставками (вставляется каждая следующая карточка на свое место). Это и есть блочная, или карманная сортировка.

Алгоритм 18: Блочная сортировка

Вход: A — массив, N — размер массива
Выход: Отсортированный массив
начало алгоритма
- · определяем число корзин P;
- · находим минимум и максимум в массиве;
- · определяем диапазон чисел в корзине: $[min + k\frac{max-min}{P}, min + (k+1)\frac{max-min}{P}]$. Следовательно, $m = \frac{max-min}{P}$;
- **цикл для** $i = 0$ *до* N **выполнять**
  - · отправляем $a[i]$ в нужную корзину в зависимости от условия $k = \frac{a[i]-min}{m}$ где k— номер корзины;
  - **если** $k = P$ **то**
    - · добавляем элемент в последнюю корзину;
- **цикл для** $i = 0$ *до* P **выполнять**
  - · сортируем элементы в i-той корзине сортировкой вставками;
- · перезаписываем корзины от 0-вой до P-той в массив A;

конец алгоритма

Например, есть массив $\{15, 28, 1, 52, 48, 33, 6, 12, 17, 25, 41, 34, 14, 50, 32\}$

Число элементов в массиве $N = 15$. Выбираем $P = 5$ корзин (при хорошем раскладе в каждой корзине должно быть по три элемента). $min = 1, max = 52$. Следовательно, $m = \frac{max-min}{P} = \frac{52-1}{5} = 10$.

Делим по корзинам по правилу: $k = \frac{a[i]-1}{10}$. Для 52 $k = 5$, а номера корзин — $[0, 4]$. Следовательно, 52 добавляем в последнюю корзину:

$b[0]$	1	6		
$b[1]$	15	12	17	14
$b[2]$	28	25		
$b[3]$	33	34	32	
$b[4]$	52	48	41	50

Сортируем каждую корзину алгоритмом вставками. Получаем следующие корзины:

$b[0]$	1	6		
$b[1]$	12	14	15	17
$b[2]$	25	28		
$b[3]$	32	33	34	
$b[4]$	41	48	50	52

Объединяем корзины: $\{1, 6, 12, 14, 15, 17, 25, 28, 32, 33, 34, 41, 48, 50, 52\}$

Оценим сложность. Надо сделать N шагов, чтобы разбить по корзинам. Для малых, почти отсортированных массивов сортировка вставками работает за $O(N)$. Надо отсортировать P корзин. Всего сделать $O(PN)$ шагов. Но так как $P < N$, то можно говорить, что сортировка работает за линейное время.

1.8 Поразрядная сортировка

Когда рассматривали сортировку подсчетом для сложной структуры, создавали k векторов, разбивали элементы по этим векторам и потом объединяли. Такая же идея подходит для поразрядной сортировки. Числа сортируются по разрядам.

Поразрядная сортировка бывает двух видов:

- LSD (least significant digit) — сортировка проводится от младшего разряда к старшему. Итоговый порядок: короткие ключи идут раньше длинных, одинаковые ключи сортируются по алфавиту. Подходит для чисел.
- MSD (most significant digit) — сортировка проводится от старшего разряда к младшему. Подходит для строк.

Будем рассматривать только LSD сортировку.

1.8.1 Поразрядная сортировка LSD

Для данной сортировке необходимо знать два параметра — N (размер массива) и k — максимальное количество разрядов чисел. P — основание системы счисления. В нашем случае $P = 10$. Цифра k-ого разряда определяется по формуле: $\frac{a[i]}{P^k} \mod P$.

Алгоритм 19: Поразрядная сортировка

Вход: A — массив, N — размер массива, k — максимальный разряд
Выход: Отсортированный массив
начало алгоритма
- · создаем P дополнительных векторов;
- **цикл для** $i = 0$ *до* k **выполнять**
  - **цикл для** $j = 0$ *до* N **выполнять**
    - · добавляем $a[j]$ в $[\frac{a[j]}{P^i} \mod P]$ вектор;
  - · перезаписываем вектора от 0-вого до $P - 1$-ого в массив A;
  - · очищаем вектора;

конец алгоритма

Рассмотрим пример:

145	12	1766	3	288	99	3405	18	479	100	387	9095	88	206	473

Размер массива — $N = 15$. Основание системы счисления — $P = 10$. Максимальный разряд $k = 4$.
Распределяем элементы по векторам в соответствии с младшим разрядом:

$b[0]$	100		
$b[1]$			
$b[2]$	12		
$b[3]$	3	473	
$b[4]$			
$b[5]$	145	3405	9095
$b[6]$	1766	206	
$b[7]$	387		
$b[8]$	288	18	88
$b[9]$	99	479	

Последовательно объединяем вектора:

100	12	3	473	145	3405	9095	1766	206	387	288	18	88	99	479

Распределяем элементы измененного по векторам в соответствии с десятками (первым разрядом):

$b[0]$	100	3	3405	206
$b[1]$	12	18		
$b[2]$				
$b[3]$				
$b[4]$	145			
$b[5]$				
$b[6]$	1766			
$b[7]$	473	479		
$b[8]$	387	288	88	
$b[9]$	9095	99		

Последовательно объединяем вектора:

100	3	3405	206	12	18	145	1766	473	479	387	288	88	9095	99

Распределяем элементы измененного массива по векторам в соответствии с сотнями (вторым разрядом):

$b[0]$	3	12	18	88	9095	99
$b[1]$	100	145				
$b[2]$	206	288				
$b[3]$	387					
$b[4]$	3405	473	479			
$b[5]$						
$b[6]$						
$b[7]$	1766					
$b[8]$						
$b[9]$						

Последовательно объединяем вектора:

3	12	18	88	9095	99	100	145	206	288	387	3405	473	479	1766

Распределяем элементы измененного массива по векторам в соответствии с тысячами (третьим разрядом):

$b[0]$	3	12	18	88	99	100	145	206	288	387	473	479
$b[1]$	1766											
$b[2]$												
$b[3]$	3405											
$b[4]$												
$b[5]$												
$b[6]$												
$b[7]$												
$b[8]$												
$b[9]$	9095											

Последовательно объединяем вектора:

3	12	18	88	99	100	145	206	288	387	473	479	1766	3405	9095

Оценим сложность. Для поиска максимума необходимо сделать N шагов. Для оценки разряда — k шагов. Для разбиения по векторам, объединения и очистка векторов надо сделать по N шагов для каждой операции. Последние три цикла повторяются k раз.

Следовательно, всего надо сделать $N + k + 3kN$ шагов. Можно сказать, что сложность $O(kN)$. Для больших N и малых разрядов (а чаще всего $k \ll N$) можно сказать, что сложность линейная.