
1. Линейный поиск

Как работает:

Последовательно перебирает элементы списка, начиная с первого, и сравнивает их с искомым значением. Останавливается, когда элемент найден или список закончился.

Пример: Поиск имени в неотсортированном списке контактов.

Где использовать:

- Небольшие наборы данных.
- Данные без определенного порядка.

Плюсы:

- Прост в реализации.
- Не требует сортировки данных.

Минусы:

- Медленный для больших массивов (временная сложность: $O(n)$).

Ссылка: <https://elib.pnzgu.ru/files/eb/gxiEozRBcbXY.pdf>

2. Бинарный поиск

Как работает:

Работает только с **отсортированными данными**. Делит массив на две части, сравнивает искомый элемент с элементом в середине. Если значение меньше — продолжает поиск в левой половине, если больше — в правой. Повторяет до успеха.

Пример: Поиск слова в словаре.

Где использовать:

- Большие отсортированные массивы.

Плюсы:

- Очень быстрый (временная сложность: $O(\log n)$).

Минусы:

- Требуется предварительной сортировки.

Ссылка: <https://elib.pnzgu.ru/files/eb/gxiEozRBcbXY.pdf>

3. Интерполяционный поиск

Как работает:

Улучшенная версия бинарного поиска. Предсказывает позицию искомого элемента, используя формулу, которая учитывает разброс значений (например, ищет число между 1 и 1000, начиная ближе к 500, если искомое значение 750).

Пример: Поиск в отсортированном списке цен.

Где использовать:

- Отсортированные данные с **равномерным распределением** (например, числа, даты).

Плюсы:

- Средняя сложность $O(\log \log n)$, что быстрее бинарного поиска.

Минусы:

- Неэффективен при неравномерном распределении данных.

Ссылка: <https://dzen.ru/a/XotYHWVp9mFV24nz>

4. Экспоненциальный поиск

Как работает:

1. Находит диапазон, где может находиться элемент, увеличивая границу в геометрической прогрессии (1, 2, 4, 8...).
2. Применяет бинарный поиск в найденном диапазоне.

Пример: Поиск в бесконечном отсортированном массиве.

Где использовать:

- Когда искомый элемент расположен ближе к началу массива.

Плюсы:

- Эффективнее линейного поиска (сложность: $O(\log n)$).

Минусы:

- Сложнее в реализации, чем бинарный поиск.

Ссылка: <https://toptechnologies.ru/ru/article/viewid=24620>

5. Поиск в глубину (DFS)

Как работает:

Идет по одному пути до конца (до тупика), затем возвращается и исследует другие ветви. Использует **стек** или рекурсию.

Пример: Поиск выхода из лабиринта.

Где использовать:

- Обход деревьев и графов.
- Поиск всех возможных решений (например, в головоломках).

Плюсы:

- Требуется мало памяти (если используется рекурсия).

Минусы:

- Может «зациклиться» в бесконечных графах.

Ссылка: <https://dzen.ru/a/XotYHWVp9mFV24nz>

6. Поиск в ширину (BFS)

Как работает:

Проверяет все узлы на текущем уровне, прежде чем перейти к следующему. Использует **очередь**.

Пример: Поиск кратчайшего пути в социальной сети (кто ближе всего к вам знаком).

Где использовать:

- Поиск кратчайшего пути в невзвешенных графах.
- Обход социальных сетей, сайтов.

Плюсы:

- Гарантированно находит кратчайший путь.

Минусы:

- Требуется много памяти (сложность по памяти: $O(n)$).

Ссылка: <https://toptechologies.ru/ru/article/viewid=24620>

7. Поиск в хеш-таблицах

Как работает:

1. Ключ преобразуется в числовой индекс с помощью **хеш-функции**.
2. Данные сохраняются по этому индексу.
3. Поиск происходит за один шаг, если нет коллизий.

Пример: Поиск номера телефона по имени в контактах.

Где использовать:

- Базы данных, кэши.
- Реализация словарей.

Плюсы:

- Идеальная сложность: $O(1)$.

Минусы:

- Коллизии замедляют поиск до $O(n)$ в худшем случае.
Ссылка: <https://elib.pnzgu.ru/files/eb/gxiEozRBcbXY.pdf>
-

8. Поиск в бинарном дереве

Как работает:

Каждый узел дерева содержит ключ. Поиск начинается с корня:

- Если искомое значение меньше ключа — переход в левое поддерево.
- Если больше — в правое.

Пример: Поиск файла в упорядоченной файловой системе.

Где использовать:

- Базы данных с частыми вставками и поиском.

Плюсы:

- Средняя сложность: $O(\log n)$.

Минусы:

- В несбалансированном дереве сложность может деградировать до $O(n)$.

Ссылка: <https://dzen.ru/a/XotYHWVp9mFV24nz>

9. Поиск прыжками (Jump Search)

Как работает:

1. Прыгает с шагом \sqrt{n} по массиву, пока не найдет элемент больше искомого.
2. Возвращается и выполняет линейный поиск в предыдущем блоке.

Пример: Поиск в отсортированном списке книг.

Где использовать:

- Отсортированные массивы, где бинарный поиск неудобен.

Плюсы:

- Сложность $O(\sqrt{n})$, лучше линейного поиска.

Минусы:

- Требуется сортировки.

Ссылка: <https://elib.pnzgu.ru/files/eb/gxiEozRBcbXY.pdf>

10. Поиск Фибоначчи

Как работает:

Использует числа Фибоначчи для деления массива на части. Напоминает бинарный поиск, но делит массив в пропорции, близкой к золотому сечению.

Пример: Оптимизация поиска в больших отсортированных массивах.

Где использовать:

- Когда сложно вычислить деление массива пополам (например, на устройствах с ограниченными ресурсами).

Плюсы:

- Эффективнее бинарного поиска для некоторых типов данных.

Минусы:

- Сложная реализация.

Ссылка: <https://toptechnologies.ru/ru/article/viewid=24620>

Источники:

1. <https://elib.pnzgu.ru/files/eb/gxiEozRBcbXY.pdf>
2. <https://toptechnologies.ru/ru/article/viewid=24620>
3. <https://dzen.ru/a/XotYHWVp9mFV24nz>