# Введение в алгоритмы

@pvavilin

24 июля 2022 г.

# Outline

# Зачем нужны алгоритмы?

## Задача

Подсчитать, какое количество сочетаний по три элементов входного массива даст в сумме 0.

# Зачем нужны алгоритмы?

```
def counter(a: List[int]) -> int:
    N:int = len(a)
    counter:int = 0
    for i in range(N):
        for j in range(i+1, N):
            for k in range(j+1, N):
                s = a[i] + a[j] + a[k]
                if s == 0:
                    counter += 1
    return counter
```

# Результат наивной реализации

*% python ./first_try.py 1K*
*2 вызовов для 1K данных: лучший результат равен 42.02*
*% python ./first_try.py 2K*
*2 вызовов для 2K данных: лучший результат равен 340.84*

# Бинарный поиск

```
def binary_search(
        lst:List[int], target:int
) -> int:
    start:int = 0
    end:int = len(lst) - 1
    while(start <= end):
        mid = (start + end) // 2
        if(lst[mid] > target):
            end = mid - 1
        elif(lst[mid] < target):
            start = mid + 1
        else:
            return mid
    return -1
```

# Быстрый ThreeSum

```
def counter(a: List[int]) -> int:
    # O(NlogN)
    arr:List[int] = sorted(a)
    N:int = len(arr)
    counter:int = 0
    for i in range(N):
        for j in range(i+1, N):
            # O(logN)
            if binary_search(
                arr, -(arr[i]+arr[j])
            ) > j:
                counter += 1
    return counter
```

# Результат быстрого ThreeSum

*% python fast_threesum.py 1K*
*2 вызовов для 1K данных: лучший результат равен 1.64*
*% python fast_threesum.py 2K*
*2 вызовов для 2K данных: лучший результат равен 7.36*
*% python fast_threesum.py 4K*
*2 вызовов для 4K данных: лучший результат равен 31.31*

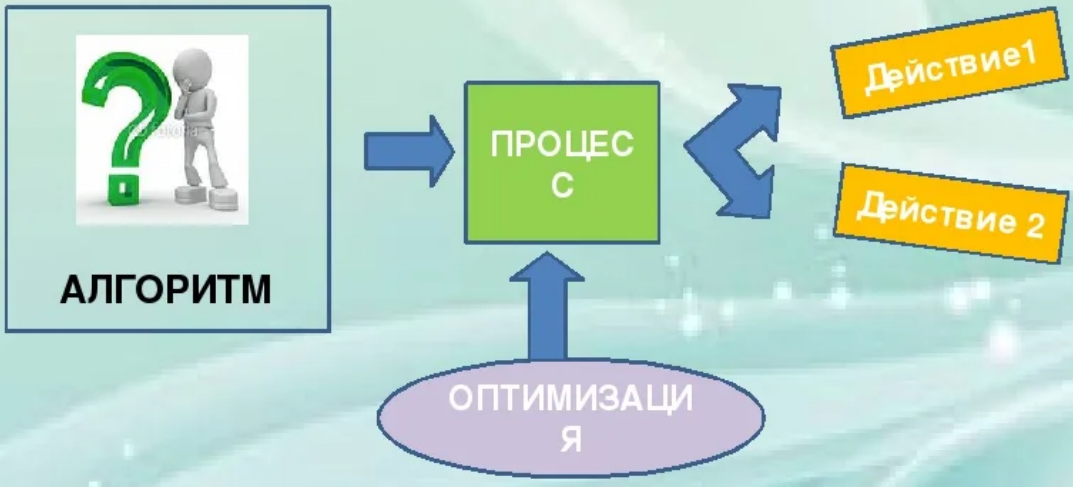
# Алгоритмическая сложность

| order of growth of time | | | | for a program that takes a few hours for input of size N | |
|---|---|---|---|---|---|
| description | function | 2x factor | 10x factor | predicted time for 10*N* | predicted time for 10*N* on a 10x faster computer |
| *linear* | $N$ | 2 | 10 | a day | a few hours |
| *linearithmic* | $N \log N$ | 2 | 10 | a day | a few hours |
| *quadratic* | $N^2$ | 4 | 100 | a few weeks | a day |
| *cubic* | $N^3$ | 8 | 1,000 | several months | a few weeks |
| *exponential* | $2^N$ | $2^N$ | $2^{9N}$ | never | never |

**Predictions on the basis of order-of-growth function**

# Нужны ли алгоритмы backend-разработчику?

## Зачем нужен алгоритм ребенку?




- возможность разложить любой процесс на составные элементы (действия);
- осознание их значения для достижения цели;
- устранение или замена элементов, которые не являются необходимыми, на более действенные.

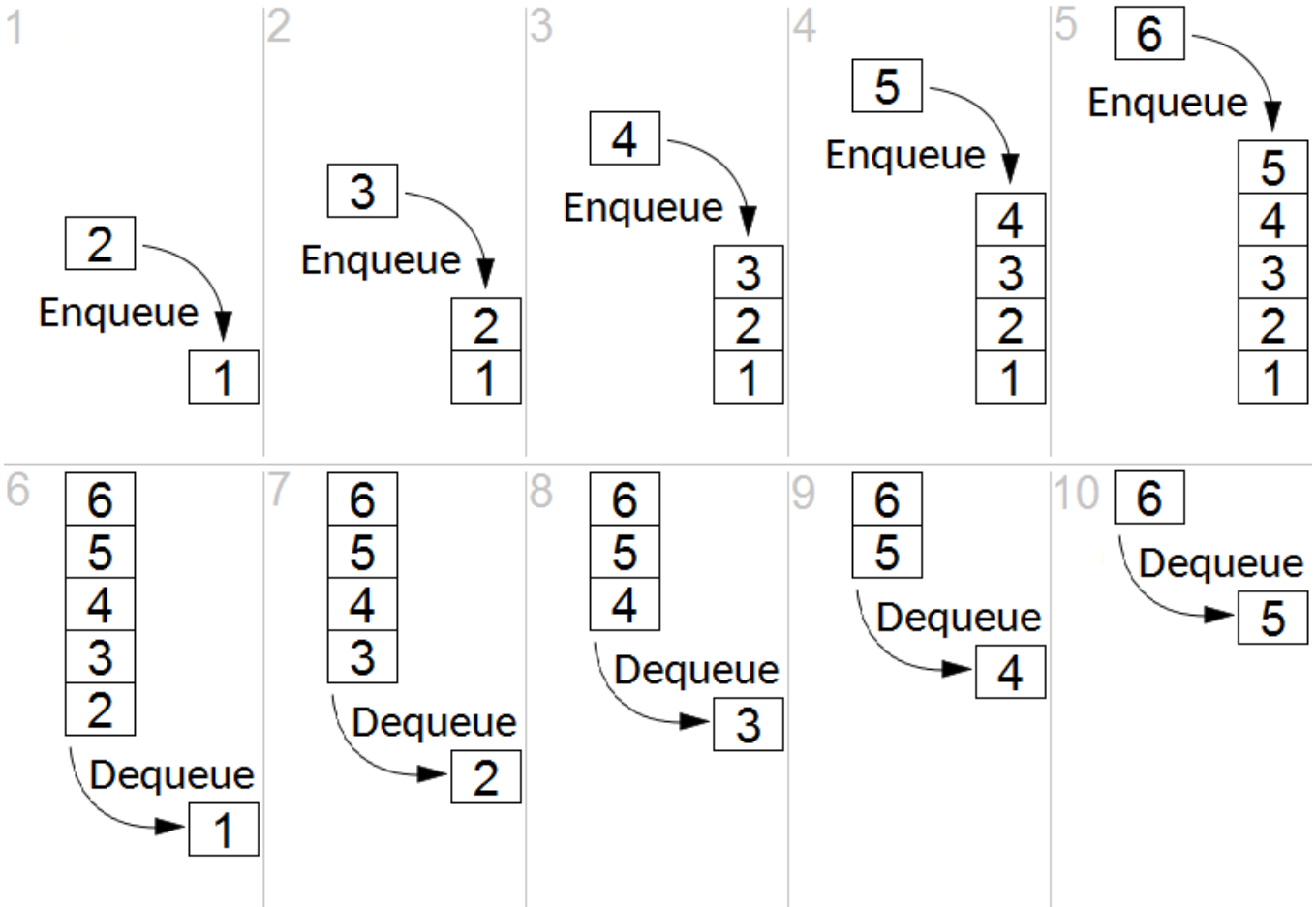
# Какие алгоритмы нужнее всего?

## Зависит от задачи, области применения

- алгоритмы на строках
  нужны например биоинформатикам, для работы с последовательностями ДНК
- алгоритмы на деревьях
  - компиляторы
  - машинное обучение
  - построение маршрутов
  - парсинг сайтов

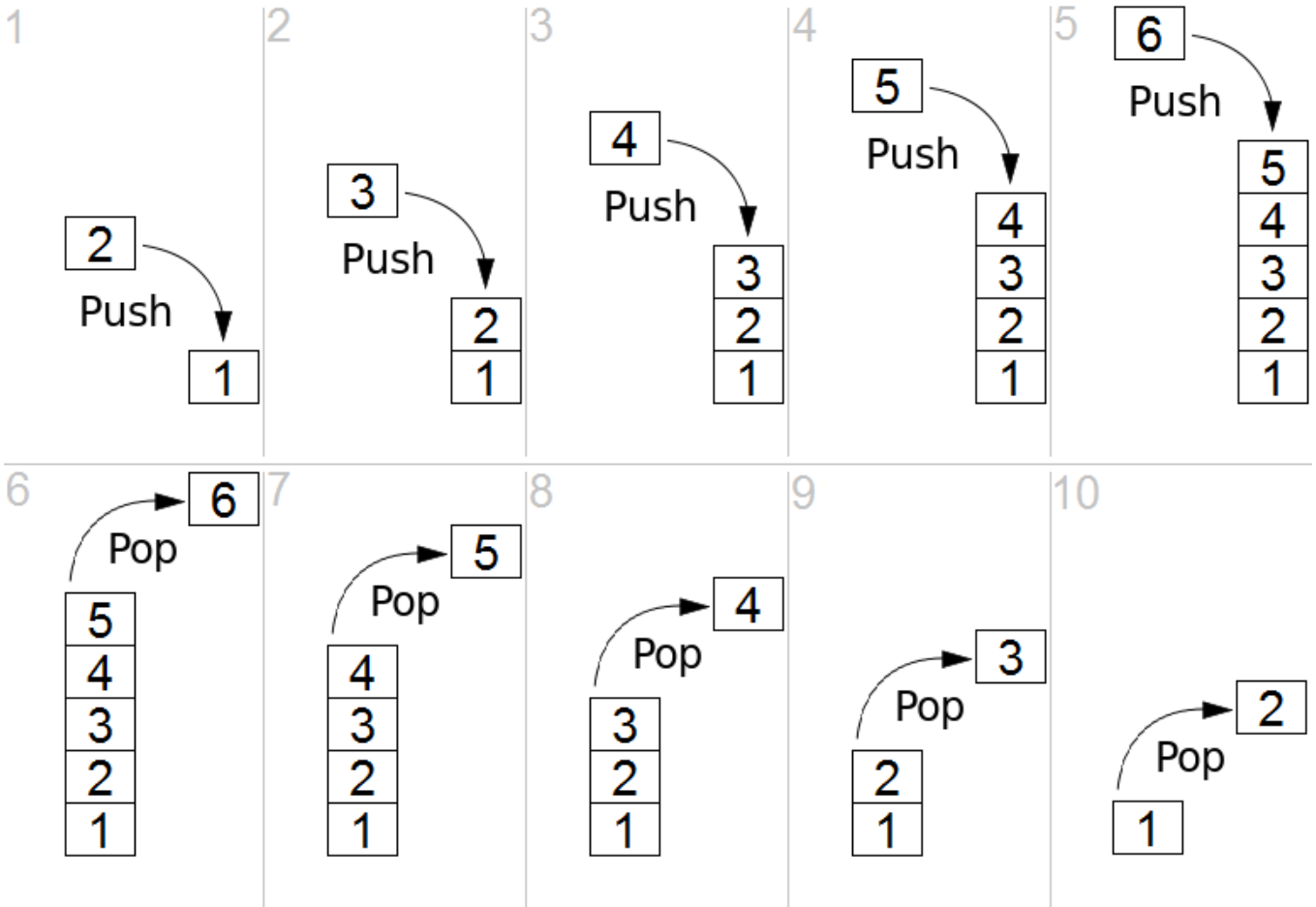
# Структуры данных. Очередь

## FIFO First In First Out



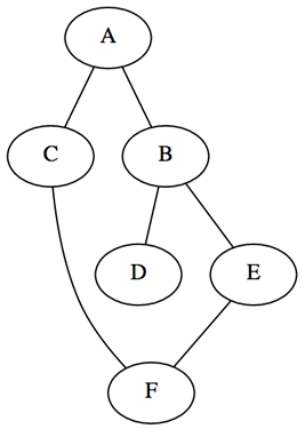
# Структуры данных. Стек

## LIFO Last In First Out



# Структуры данных. Граф

```
graph = {
        'A': ['B', 'C'],
        'B': ['A', 'D', 'E'],
        'C': ['A', 'F'],
        'D': ['B'],
        'E': ['B', 'F'],
        'F': ['C', 'E']
}
```



# Поиск вглубину. Depth-First Search

```
def dfs(graph, start, goal):
    stack = [(start, [start])]
    while stack:
        (v, p) = stack.pop()
        paths = set(graph[v]) - set(p)
        for nxt in paths:
            if nxt == goal:
                yield p + [nxt]
            else:
                stack.append((nxt, p+[nxt]))
print(list(dfs(graph, 'A', 'F')))
[['A', 'B', 'E', 'F'], ['A', 'C', 'F']]
```

# Глупая сортировка / сортировка дурака

```
def sort_alg(l):
    while True:
        c = 0
        for i in range(len(l)-1):
            if l[i] > l[i+1]:
                l[i+1],l[i] = l[i],l[i+1]
            else:
                c += 1
        if c == (len(l) - 1): return l
print(sort_alg([1, 3, 2, 0]))
```

```
[0, 1, 2, 3]
```

# Результат глупой сортировки

- Эффективность глупой сортировки: $\mathcal{O}(N^3)$

*% ./fool_sort.py 1K*
*2 вызовов для 1K данных: лучший результат равен 0.12*
*% ./fool_sort.py 2K*
*2 вызовов для 2K данных: лучший результат равен 0.53*
*% ./fool_sort.py 4K*
*2 вызовов для 4K данных: лучший результат равен 2.15*

# Пузырьковая сортировка

```
def sort_alg(l):
    for i in range(len(l)):
        for j in range(len(l[i+1:])):
            if l[j] > l[j+1]:
                l[j], l[j+1] = (l[j+1], l[j])
    return l

print(sort_alg([1, 3, -1, 2, 0]))
```

```
[-1, 0, 1, 2, 3]
```

# Результат пузырьковой сортировки

- Эффективность пузырьковой сортировки: $\mathcal{O}(N^2)$

*% ./bubble_sort.py 1K*
*2 вызовов для 1K данных: лучший результат равен 0.11*
*% ./bubble_sort.py 2K*
*2 вызовов для 2K данных: лучший результат равен 0.45*
*% ./bubble_sort.py 4K*
*2 вызовов для 4K данных: лучший результат равен 1.86*

# Сортировка слиянием (Merge Sort)

- Код
- мультик

Сортировка слиянием позволяет нам распараллелить процесс сортировки. Это очень эффективно на больших данных и широко используется в алгоритмах map/reduce.

# Результат Merge Sort

- Эффективность Merge Sort: $\mathcal{O}(NlogN)$

*% ./merge_sort.py 1K*
*2 вызовов для 1K данных: лучший результат равен 0.01*
*% ./merge_sort.py 4K*
*2 вызовов для 4K данных: лучший результат равен 0.03*
*% ./merge_sort.py 8K*
*2 вызовов для 8K данных: лучший результат равен 0.07*
*% ./merge_sort.py 32K*
*2 вызовов для 32K данных: лучший результат равен 0.31*

# Сравнение алгоритмов сортировки

**Comparison sorts**

| Name | Best | Average | Worst | Memory | Stable | Method | Other notes |
|---|---|---|---|---|---|---|---|
| Quicksort | $n \log n$ | $n \log n$ | $n^2$ | $\log n$ | No | Partitioning | Quicksort is usually done in-place with $O(\log n)$ stack space.[5][6] |
| Merge sort | $n \log n$ | $n \log n$ | $n \log n$ | $n$ | Yes | Merging | Highly parallelizable (up to $O(\log n)$ using the Three Hungarians' Algorithm).[7] |
| In-place merge sort | — | — | $n \log^2 n$ | 1 | Yes | Merging | Can be implemented as a stable sort based on stable in-place merging.[8] |
| Introsort | $n \log n$ | $n \log n$ | $n \log n$ | $\log n$ | No | Partitioning & Selection | Used in several STL implementations. |
| Heapsort | $n \log n$ | $n \log n$ | $n \log n$ | 1 | No | Selection | |
| Insertion sort | $n$ | $n^2$ | $n^2$ | 1 | Yes | Insertion | $O(n + d)$, in the worst case over sequences that have $d$ inversions. |
| Block sort | $n$ | $n \log n$ | $n \log n$ | 1 | Yes | Insertion & Merging | Combine a block-based $O(n)$ in-place merge algorithm[9] with a bottom-up merge sort. |
| Timsort | $n$ | $n \log n$ | $n \log n$ | $n$ | Yes | Insertion & Merging | Makes *n-1* comparisons when the data is already sorted or reverse sorted. |
| Selection sort | $n^2$ | $n^2$ | $n^2$ | 1 | No | Selection | Stable with $O(n)$ extra space, when using linked lists, or when made as a variant of Insertion Sort instead of swapping the two items.[10] |
| Cubesort | $n$ | $n \log n$ | $n \log n$ | $n$ | Yes | Insertion | Makes *n-1* comparisons when the data is already sorted or reverse sorted. |
| Shellsort | $n \log n$ | $n^{4/3}$ | $n^{3/2}$ | 1 | No | Insertion | Small code size. |
| Bubble sort | $n$ | $n^2$ | $n^2$ | 1 | Yes | Exchanging | Tiny code size. |
| Exchange sort | $n^2$ | $n^2$ | $n^2$ | 1 | Yes | Exchanging | Tiny code size. |
| Tree sort | $n \log n$ | $n \log n$ | $n \log n$ (balanced) | $n$ | Yes | Insertion | When using a self-balancing binary search tree. |
| Cycle sort | $n^2$ | $n^2$ | $n^2$ | 1 | No | Selection | In-place with theoretically optimal number of writes. |
| Library sort | $n \log n$ | $n \log n$ | $n^2$ | $n$ | No | Insertion | Similar to a gapped insertion sort. It requires randomly permuting the input to warrant with-high-probability time bounds, which makes it not stable. |
| Patience sorting | $n$ | $n \log n$ | $n \log n$ | $n$ | No | Insertion & Selection | Finds all the longest increasing subsequences in $O(n \log n)$. |
| Smoothsort | $n$ | $n \log n$ | $n \log n$ | 1 | No | Selection | An adaptive variant of heapsort based upon the Leonardo sequence rather than a traditional binary heap. |
| Strand sort | $n$ | $n^2$ | $n^2$ | $n$ | Yes | Selection | |
| Tournament sort | $n \log n$ | $n \log n$ | $n \log n$ | $n$[11] | No | Selection | Variation of Heapsort. |
| Cocktail shaker sort | $n$ | $n^2$ | $n^2$ | 1 | Yes | Exchanging | A variant of Bubblesort which deals well with small values at end of list |
| Comb sort | $n \log n$ | $n^2$ | $n^2$ | 1 | No | Exchanging | Faster than bubble sort on average. |
| Gnome sort | $n$ | $n^2$ | $n^2$ | 1 | Yes | Exchanging | Tiny code size. |
| Odd–even sort | $n$ | $n^2$ | $n^2$ | 1 | Yes | Exchanging | Can be run on parallel processors easily. |

# Устойчивость сортировки

```
records = [
    {"A": "X", "B": 2},
    {"A": "X", "B": 1},
    {"A": "Y", "B": 1},
]
records.sort(key=lambda x: x["A"])
for r in records:
    print(f"{r['A']}, {r['B']}")
```

```
X, 2
X, 1
Y, 1
```

# Как изучать алгоритмы

- Яндекс.Практикум
- Coursera (Part I, Part II)
- Альманах алгоритмов: Т.Кормен, Ч.Лейзерсон, Р.Ривест, К.Штайн «Алгоритмы. Построение и анализ.»