# Турнирное дерево

В состязаниях на вылет выявляется победитель среди нескольких команд. Лучше всего, если количество команд изначально равно какой-нибудь степени двойки, например 16 или 64. Розыгрыш состоит из нескольких туров, в каждом из которых все оставшиеся команды разбиваются на пары для игры. Каждый проигравший в паре выбывает из соревнований, а остальные переходят на следующий круг. Команда, выигравшая в финале, объявляется победителем.

Предположим, нам надо найти максимум в списке `p = [3,1,4,1,5,9,2,6]` длиной $N = 8$. На рис. 1.6 показан розыгрыш на вылет, в первом туре которого попарно сравниваются восемь значений, и те, что больше, переходят на второй[1]. В туре «Великолепной Восьмерки» выбывают четыре значения, и остается `[3,4,9,6]`, из тура «Чемпионской Четверки» в финал выходят `[4,9]`, и в результате победителем становится `9`[2].

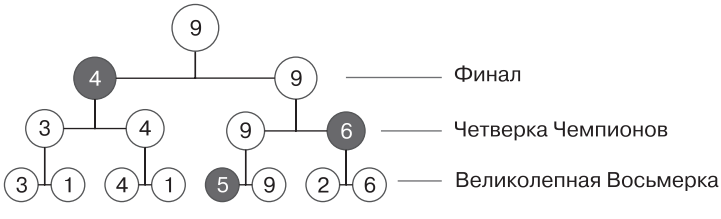


**Рис. 1.6.** Турнирное дерево с восемью участниками

Для применения турнирного дерева требуется семь сравнений (по одному на игру), и это обнадеживает: как мы уже говорили, это означает, что на поиск максимума в наборе данных размером $N$ требуется $N - 1$ сравнений. Если запоминать все эти сравнения, нетрудно показать, что второе наибольшее значение находится быстро.

Где может «прятаться» второе наибольшее значение, если победителем объявлено `9`? Начнем со значения `4`, раз уж оно добралось до финала и проиграло

---

[1] В случае ничьей, то есть равных значений, берется первое из них. — *Примеч. авт.*

[2] Автор использует термины турнира баскетбольной лиги США первого дивизиона: региональные полуфиналы (Sweet Sixteen), региональные финалы (Elite Eight) и национальные полуфиналы (Final Four). Знатокам турнира эти термины хорошо известны, но я решил перевести эти названия, сохраняя принцип «первые буквы повторяются». — *Примеч. пер.*

только там. Однако победитель, 9, участвовал еще в двух играх, так что надо проверить еще двух проигравших — 6 из тура «Четверки Чемпионов» и 5 из тура «Великолепной Восьмерки». Получается, что второй ответ — 6.

Чтобы выяснить, что 6 — второе наибольшее значение, для длины списка 8 достаточно двух дополнительных сравнений — «4 меньше 6?» и «6 меньше 5?». То, что $8 = 2^3$, а сравнений было $3 - 1 = 2$ — не совпадение. Действительно, для $N = 2^K$ необходимо $K - 1$ дополнительных сравнений, при этом $K$ оказывается количеством туров в розыгрыше.

Для $8 = 2^3$ элементов алгоритму требуется розыгрыш из трех туров. На рис. 1.7 приведен розыгрыш из пяти туров, соответствующий 32 элементам. Если удвоить количество элементов, потребуется еще один тур. Иными словами, в туре под номером $K$ может участвовать $2^K$ элементов. Нужно найти максимум среди 64 элементов? Потребуется шесть туров, потому что $2^6 = 64$.

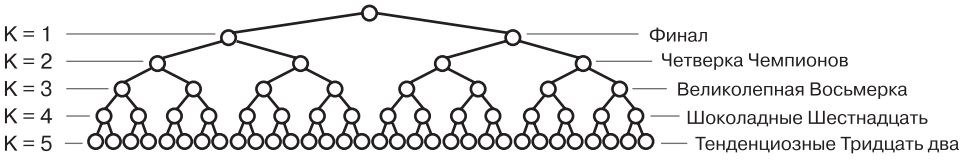


**Рис. 1.7.** Турнирное дерево с 32 участниками

Чтобы определить, сколько туров потребуется для произвольного $N$, используем *логарифмическую* функцию `log()` — это обратная к показательной функции, `exp()`. Для $N = 8$ элементов на весь розыгрыш требуется три тура, потому что $2^2 = 8$, и, стало быть, $\log_2 8 = 3$. В нашей книге, как и в большинстве задач оценки сложности, используется логарифм с основанием 2 — двоичный.

Большинство настольных калькуляторов (а заодно Microsoft Excel) по команде log() вычисляют десятичный логарифм (по основанию 10). В них также есть команда ln() для вычисления натурального логарифма, основание которого равно константе e (примерно 2.7182818). Чтобы вычислить двоичный логарифм с помощью любой из этих функций, надо поделить результат на логарифм двух: $\log(N) / \log(2)$.

Если $N$ — степень двойки, например 64 или 65 536, в розыгрыше будет $\log_2(N)$ туров, а это значит, что потребуется еще $\log_2(N) - 1$ сравнений. В примере 1.6 приведен алгоритм, который сводит к минимуму количество сравнений за счет дополнительной памяти, где откладываются результаты этих сравнений.

**Пример 1.6.** Поиск двух наибольших значений A с помощью турнирного дерева

```
def tournament_two(A):
  N = len(A)
  winner = [None] * (N-1)          ❶
  loser = [None] * (N-1)
  prior = [-1] * (N-1)             ❷

    idx = 0
    for i in range(0, N, 2):       ❸
      if A[i] < A[i+1]:
        winner[idx] = A[i+1]
        loser[idx] = A[i]
      else:
        winner[idx] = A[i]
        loser[idx] = A[i+1]
      idx += 1

    m = 0                          ❹
    while idx < N-1:
      if winner[m] < winner[m+1]:  ❺
        winner[idx] = winner[m+1]
        loser[idx] = winner[m]
        prior[idx] = m+1
      else:
        winner[idx] = winner[m]
        loser[idx] = winner[m+1]
        prior[idx] = m
      m += 2                       ❻
      idx += 1

    largest = winner[m]
    second = loser[m]              ❼
    m = prior[m]
    while m >= 0:
      if second < loser[m]:        ❽
        second = loser[m]
      m = prior[m]

    return (largest, second)
```

❶ В этих списках мы станем хранить индексы победителей и проигравших в игре, которых будет $N - 1$.

❷ Когда значение в позиции `m` проходит на очередной тур, в `prior[m]` записывается позиция этого значения в предыдущем туре. Для первого тура такой информации нет, поэтому в начале этого списка хранится `-1`.

❸ Первый тур состоит из $N / 2$ игр, то есть требует $N / 2$ сравнений на «меньше» в парах «победитель — проигравший».

❹ Игры победителей во всех следующих турах с записью позиции выигравшего в каждой игре.

❺ Еще $N / 2 - 1$ сравнение.

❻ Увеличим `m` на 2 — это игра двух следующих победителей. Когда `idx` достигнет $N - 1$, в `winner[m]` окажется наибольшее значение.

❼ Первый кандидат на второе место; надо еще проверить остальных проигравших чемпиону, возможно, они больше подходят для второго места.

❽ Не больше $\log_2(N) - 1$ дополнительное сравнение на «меньше».

Рисунок 1.8 показывает работу нашего алгоритма. Начальный шаг превращает $N$ значений исходного списка `A` в $N / 2$ значений в `winners` и `losers`; на рис. 1.6 это четыре пары. На следующем шаге с каждым проходом цикла `while` победитель и проигравший в очередной игре под номером `idx` (они находятся на соседних позициях `m` и `m+1`) помещаются соответственно в `winner[idx]` и `loser[idx]`.

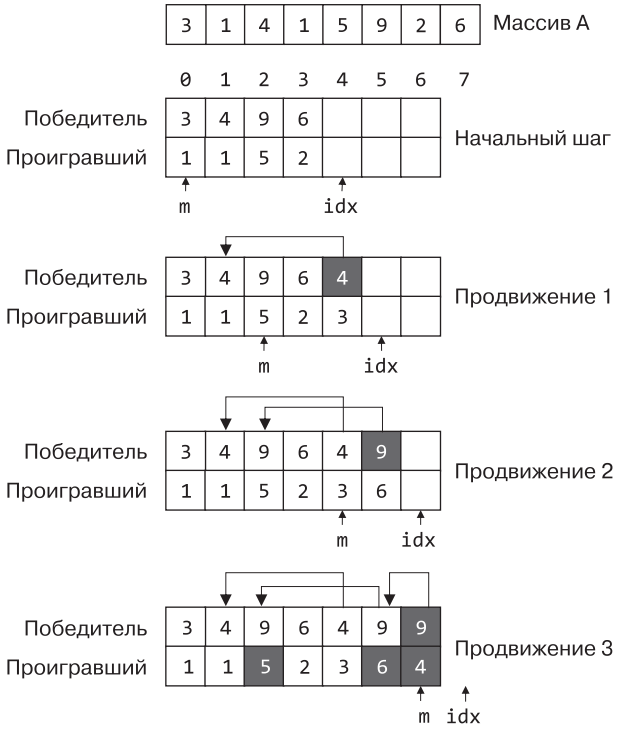


**Рис. 1.8.** Пошаговое выполнение алгоритма с использованием турнирного дерева

В `prior[idx]` мы записываем предыдущую позицию выигравшего, с которой он попал в эту игру (обозначено стрелкой справа налево). После трех шагов мы имеем всю информацию о розыгрыше, и алгоритм проверяет всех проигравших в играх с чемпионом — последовательность, которую можно проследить по стрелкам от чемпиона назад. Второе наибольшее значение можно найти всего двумя сравнениями, что больше — начальный кандидат (найденный в `loser[6]`), `loser[5]` или `loser[2]`.

Итак, мы описали алгоритм поиска двух наибольших элементов `A`, которому нужно всего $N - 1 + \log_2(N) - 1 = N + \log_2(N) - 2$ сравнений на «меньше» для любого $N$, равного степени двойки. А насколько практична функция `tournament_two()`? Работает ли она быстрее `largest_two()`? Если считать только сравнения элементов на «меньше», `tournament_two()` должна быть быстрее. На наборе данных длиной $N = 65\,536$ функция `largest_two()` выполняет 131 069 сравнений, а `tournament_two()` — только $65\,536 + 16 - 2 = 65\,550$, то есть примерно половину. Но история на этом не кончается.

Таблица 1.5 показывает, что функция `tournament_two()` значительно медленнее всех своих соперниц! Достаточно посмотреть, сколько времени у нее уходит на обработку ста случайных наборов данных, размер которых растет от 1024 до 2 097 152 элементов. Раз уж мы об этом заговорили, давайте еще добавим в таблицу производительность функций из примера 1.5. Если программу с примерами запускать на другом компьютере, конкретные результаты получатся другими, но *общая закономерность* сохранится.

**Таблица 1.5.** Сравнение времени работы в миллисекундах всех четырех алгоритмов

| N | double_two | mutable_two | largest_two | sorting_two | tournament_two |
|---|---|---|---|---|---|
| 1024 | 0.00 | 0.01 | 0.01 | 0.01 | 0.03 |
| 2048 | 0.01 | 0.01 | 0.01 | 0.02 | 0.05 |
| 4096 | 0.01 | 0.02 | 0.03 | 0.03 | 0.10 |
| 8192 | 0.03 | 0.05 | 0.05 | 0.08 | 0.21 |
| 16 384 | 0.06 | 0.09 | 0.11 | 0.18 | 0.43 |
| 32 768 | 0.12 | 0.20 | 0.22 | 0.40 | 0.90 |
| 65 536 | 0.30 | 0.39 | 0.44 | 0.89 | 1.79 |
| 131 072 | 0.55 | 0.81 | 0.91 | 1.94 | 3.59 |
| 262 144 | 1.42 | 1.76 | 1.93 | 4.36 | 7.51 |
| 524 288 | 6.79 | 6.29 | 5.82 | 11.44 | 18.49 |
| 1 048 576 | 16.82 | 16.69 | 14.43 | 29.45 | 42.55 |
| 2 097 152 | 35.96 | 38.10 | 31.71 | 66.14 | … |

С непривычки табл. 1.5 выглядит как стена чисел. Если запускать наши функции на другом компьютере — с меньшей памятью или менее мощным процессором, — это будут другие числа, но некоторые закономерности можно проследить безотносительно к быстродействию компьютера. В первую очередь рост чисел в каждом столбце: при удвоении размера входного набора время выполнения увеличивается тоже примерно вдвое.

В таблице можно заметить кое-что неожиданное: к примеру, `double_two()` поначалу ведет себя как самое быстрое решение, но с ростом $N$ (после $N > 262.144$) уступает пальму первенства `largest_two()`. А хитрая наша `tournament_two()` оказалась ужасно медленной — она тратит много времени на создание и обработку вспомогательных списков, размер которых даже больше, чем объем входных данных. Она настолько медленно работает, что на самых больших наборах даже не проверяется — это было бы слишком долго.

Чтобы получить представление обо всех этих числах, посмотрим на рис. 1.9, где в виде графиков представлена зависимость производительности от роста объема данных.

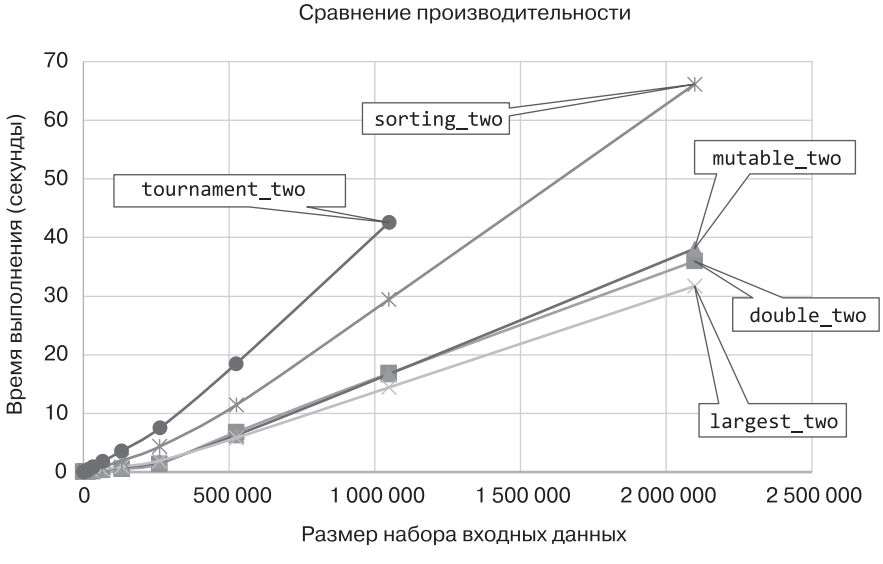


**Рис. 1.9.** Сравнение тестов производительности

Графики показывают особенности работы всех пяти алгоритмов.

- Видно, что производительность `mutable_two()`, `double_two()` и `largest_two()` примерно одинакова, но явно отличается от производительности двух других функций. Можно сказать, что эти три функции образуют «семейство»:

графики их производительности — прямые линии, предсказать продолжение которых, по-видимому, просто.

- Функция `tournament_two()` — самая медленная, и ее график заметно отличается от прочих. Измерений сделано немного, поэтому до конца не ясно, будет ли график продолжать расти вверх или тоже в конце концов превратится в прямую.
- Функция `sorting_two()` явно получше, чем `tournament_two()`, однако медленнее остальных трех. А ее график? Будет ли он и дальше изгибаться кверху или стремиться к прямой?

Чтобы понять, отчего графики имеют такой вид, надо обратиться к двум важным понятиям, которые определяют сложность, присущую алгоритмам.

## Сложность по времени и сложность по памяти

Сложно посчитать, сколько конкретных машинных инструкций (например, сложений, присваиваний или ветвлений) выполнила программа: это зависит от языка программирования, тем более что некоторые из них, например Python или Java, требуют для работы интерпретатор. Но если общее число выполненных инструкций посчитать *можно*, то можно исследовать, как оно зависит от объема входных данных. Задача оценки *сложности по времени* — получить некоторую формулу $C(N)$, которая вычисляла бы количество инструкций, выполненных некоторым алгоритмом, как функцию от $N$ — размера набора входных данных.

Предположим, выполнение одной машинной инструкции центральным процессором некоторого компьютера требует фиксированного времени $t$. Тогда время $T$, затраченное на работу алгоритма на этом компьютере, можно выразить как $T(N) = t \times C(N)$. Пример 1.7 подтверждает догадку о том, что самое важное — это *структура* программы. Можно в точности посчитать, сколько потребуется сложений вида `ct = ct + 1` для работы функций `f0()`, `f1()`, `f2()` и `f3()` на входных данных размером $N$.

**Пример 1.7.** Четыре различные функции с разной вычислительной сложностью

```
def f0(N):
  ct = 0
  ct = ct + 1
  ct = ct + 1
  return ct
```

```
def f1(N):
  ct = 0
  for i in range(N):
    ct = ct + 1
  return ct
```

```
def f2(N):
  ct = 0
  for i in range(N):
    ct = ct + 1
    ct = ct + 1
    ct = ct + 1
    ct = ct + 1
    ct = ct + 1
    ct = ct + 1
    ct = ct + 1
  return ct
```

```
def f3(N):
  ct = 0
  for i in range(N):
    for j in range(N):
      ct = ct + 1
  return ct
```