Теория алгоритмов

- Первым дошедшим до нас алгоритмом считается предложенный Евклидом в III веке до нашей эры алгоритм нахождения наибольшего общего делителя двух чисел - алгоритм Евклида
- Начальной точкой отсчета современной теории алгоритмов можно считать работу немецкого математика Курта Гёделя 1931 год - теорема о неполноте символических логик
- Первые фундаментальные работы по теории алгоритмов были опубликованы независимо в 1936 году годы Аланом Тьюрингом, Алоизом Черчем и Эмилем Постом
- В 1950-е годы существенный вклад в теорию алгоритмов внесли работы Колмогорова и Маркова.

К 1960-70-ым годам оформились следующие направления в теории алгоритмов:

- Классическая теория алгоритмов
- Теория асимптотического анализа алгоритмов
- Теория практического анализа вычислительных алгоритмов

Цели и задачи теории алгоритмов

- формализация понятия «алгоритм» и исследование формальных алгоритмических систем;
- формальное доказательство алгоритмической неразрешимости ряда задач;
- классификация задач, определение и исследование сложностных классов;
- асимптотический анализ сложности алгоритмов;
- исследование и анализ рекурсивных алгоритмов;
- получение явных функций трудоемкости в целях сравнительного анализа алгоритмов;
- разработка критериев сравнительной оценки качества алгоритмов.

Практическое применение результатов теории алгоритмов

- Теоретический аспект

  - является ли задача в принципе алгоритмически разрешимой

- Практический аспект

  - рациональный выбор из известного множества алгоритмов решения

    данной задачи

  - получение временных оценок решения сложных задач

  - получение достоверных оценок невозможности решения некоторой

    задачи за определенное время

  - разработка и совершенствование эффективных алгоритмов

Понятие алгоритма

Определение 1.1: *Алгоритм* - это заданное на некотором языке конечное предписание, задающее конечную последовательность выполнимых элементарных операций для решения задачи, общее для класса возможных исходных данных.

Определение 1.2 (Колмогоров): *Алгоритм* – это всякая система вычислений, выполняемых по строго определенным правилам, которая после какого-либо числа шагов заведомо приводит к решению поставленной задачи.

Определение 1.3 (Марков): *Алгоритм* – это точное предписание, определяющее вычислительный процесс, идущий от варьируемых исходных данных к искомому результату.

Требования к алгоритму

- алгоритм должен содержать *конечное* количество элементарно выполнимых предписаний, т.е. удовлетворять требованию конечности записи;
- алгоритм должен выполнять конечное количество шагов при решении задачи, т.е. удовлетворять требованию конечности действий;
- алгоритм должен быть единым для всех допустимых исходных данных, т.е. удовлетворять требованию универсальности;
- алгоритм должен приводить к правильному по отношению к поставленной задаче решению, т.е. удовлетворять требованию правильности.

Одной из фундаментальных статей, результаты которой лежат в основе современной теории алгоритмов является статья Эмиля Поста (Emil Post), «Финитные комбинаторные процессы»

Пост рассматривает общую проблему, состоящую из множества конкретных проблем, при этом решение общей проблемы это такое решение, которое доставляет ответ для каждой конкретной проблемы.

Машина Поста

Одной из фундаментальных статей, результаты которой лежат в основе современной теории алгоритмов является статья Эмиля Поста (Emil Post), «Финитные комбинаторные процессы»

Пост рассматривает общую проблему, состоящую из множества конкретных проблем, при этом решение общей проблемы это такое решение, которое доставляет ответ для каждой конкретной проблемы.

Основные понятия алгоритмического формализма Поста

- пространство символов (язык L) в котором задаётся конкретная проблема и получается ответ
- набор инструкций, т.е. операций в пространстве символов, задающих как сами операции, так и порядок выполнения инструкций

Постовское пространство символов – это бесконечная лента ячеек.

Каждый ящик или ячейка могут быть помечены или не п

Конкретная проблема задается «внешней силой» пометкой конечного количества ячеек, при этом, очевидно, что любая конфигурация начинается и заканчивается помеченным ящиком.

Набор инструкций (элементарных операций) Поста

- 1.пометить ящик, если он пуст;
- 2.стереть метку, если она есть;
- 3.переместиться влево на 1 ящик;
- 4.переместиться вправо на 1 ящик;
- 5.определить помечен ящик или нет, и по результату перейти на одну из двух указанных инструкций;
- 6.остановиться.

Программа представляет собой нумерованную последовательность инструкций, причем переходы в инструкции 5 производятся на указанные в ней номера других инструкций.

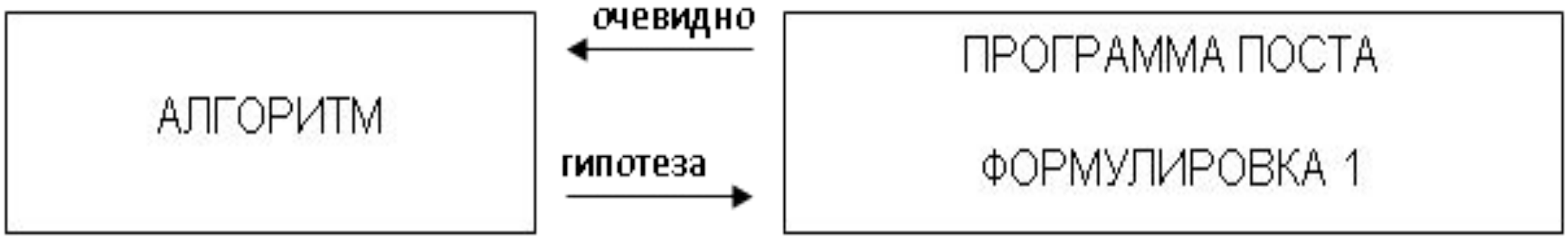
Пост формулирует требование универсальности и вводит следующие понятия

- набор инструкций *применим* к общей проблеме, если для каждой конкретной проблемы не возникает коллизий в инструкциях 1 и 2, т.е. никогда программа не стирает метку в пустом ящике и не устанавливает метку в помеченном ящике;
- набор инструкций *заканчивается* (за конечное количество инструкций), если выполняется инструкция (6);
- набор инструкций задаёт *финитный 1 – процесс*, если набор применим, и заканчивается для каждой конкретной проблемы;
- финитный 1 – процесс для общей проблемы есть 1 – решение, если ответ для каждой конкретной проблемы

Способ задания проблемы и формулировка 1

- Общая проблема называется по Посту *1-заданой,* если существует такой финитный 1 – процесс, что, будучи, применим к *n* ***є N*** в качестве исходной конфигурации ящиков, он задает *n-ую* конкретную проблему в виде набора помеченных ящиков.
- Если общая проблема 1-задана и 1-разрешима, то, соединяя наборы инструкций по заданию проблемы, и ее решению мы получаем ответ по номеру проблемы – это и есть в терминах статьи Поста *формулировка 1*.

Гипотеза Поста состоит в том, что любые более широкие формулировки в смысле алфавита символов ленты, набора инструкций, представления и интерпретации конкретных проблем сводимы к формулировке 1.



Следовательно, если гипотеза верна, то любые другие формальные определения, задающие некоторый класс алгоритмов, эквивалентны классу алгоритмов, заданных формулировкой 1 Эмиля Поста.

Машина Тьюринга

Машина Тьюринга является расширением модели конечного автомата, расширением, включающим потенциально бесконечную память с возможностью перехода (движения) от обозреваемой в данный момент ячейки к ее левому или правому соседу

Формально машина Тьюринга может быть описана следующим образом

Пусть заданы:

- конечное множество состояний – Q, в которых может находиться машина Тьюринга;
- конечное множество символов ленты – Γ;
- функция δ (функция переходов или программа), которая задается отображением пары из декартова произведения $Q \times \Gamma$ (машина находится в состоянии q_i и обозревает символ γ_i) в тройку декартова произведения $Q \times \Gamma \times \{L,R\}$ (машина переходит в состояние q_j, заменяет символ γ_i на символ γ_j и передвигается влево или вправо на один символ ленты) – $Q \times \Gamma \rightarrow Q \times \Gamma \times \{L,R\}$
- один символ из $\Gamma \rightarrow e$ (пустой);
- подмножество $\Sigma \in \Gamma \rightarrow$ определяется как подмножество входных символов ленты, причем $e \in (\Gamma - \Sigma)$;
- одно из состояний – $q_0 \in Q$ является начальным состоянием машины.

Решаемая проблема задается путем записи конечного количества символов из множества Σ є Г – Si є Σ на ленту

- после чего машина переводится в начальное состояние и головка устанавливается у самого левого непустого символа – $(q_0, \uparrow\omega)$, после чего в соответствии с указанной функцией переходов $(q_i, S_i) \rightarrow (q_j, S_k, L$ *или* $R)$ машина начинает заменять обозреваемые символы, передвигать головку вправо или влево и переходить в другие состояния, предписанные функций переходов.
- Остановка машины происходит в том случае, если для пары (q_i, S_i) функция перехода не определена.

Алгоритмически неразрешимые проблемы

Теорема

Не существует алгоритма (машины Тьюринга), позволяющего по описанию произвольного алгоритма и его исходных данных (и алгоритм и данные заданы символами на ленте машины Тьюринга) определить, останавливается ли этот алгоритм на этих данных или работает бесконечно.

- **Проблема 1**: Распределение девяток в записи числа *π* [10];
- **Проблема 2**: Вычисление совершенных чисел;
- **Проблема 3**: Десятая проблема Гильберта;
- **Проблема 4**: Позиционирование машины Поста на последний

  помеченный ящик [10];
- **Проблема 5**: Проблема «останова» ;
- **Проблема 6**: Проблема эквивалентности алгоритмов;
- **Проблема 7**: Проблема тотальности;

Введение в анализ алгоритмов

При использовании алгоритмов для решения практических задач мы сталкиваемся с проблемой рационального выбора алгоритма решения задачи. Решение проблемы выбора связано с построением системы сравнительных оценок, которая в свою очередь существенно опирается на формальную модель алгоритма.

Конкретная проблема задается N словами памяти, таким образом, на входе алгоритма – $N_\beta = N*\beta$ бит информации. Отметим, что в ряде случаев, особенно при рассмотрении матричных задач N является мерой длины входа алгоритма, отражающей линейную размерность.

Программа, реализующая алгоритм для решения общей проблемы состоит из M машинных инструкций по $\beta_м$ битов – $M_\beta = M*\beta_м$ бит информации.

Кроме того, алгоритм может требовать следующих дополнительных ресурсов абстрактной машины:

- S_d – память для хранения промежуточных результатов;
- S_r – память для организации вычислительного процесса

Трудоёмкость алгоритма.

Трудоёмкостью алгоритма для конкретного входа – $F_a(N)$, является количество «элементарных» операций совершаемых алгоритмом для решения конкретной проблемы в данной формальной системе.

$$\psi_A = c_1 * F_a(N) + c_2 * M + c_3 * S_d + c_4 * S_r,$$

где c_i – веса ресурсов.

Система обозначений в анализе алгоритмов

1. $F_a^{\wedge}(N)$ – *худший случай* – наибольшее количество операций, совершаемых алгоритмом *A* для решения конкретных проблем размерностью *N*:

$$F_a^{V}(N) = \min_{D \in D_N} \{F_a(D)\}$$ – лучший случай на D_N

2. $F_a^V(N)$ – *лучший случай* – наименьшее количество операций, совершаемых алгоритмом A для решения конкретных проблем размерностью *N*:

$$F_a^V(N) = \min_{D \in D_N} \{F_a(D)\}$$ – лучший случай на D_N

3. $\boldsymbol{F_a(N)}$ – *средний случай* – среднее количество операций, совершаемых алгоритмом A для решения конкретных проблем размерностью *N:*

$$F_a(N) = (1 / M_{DN}) * \sum_{D \in D_N} \{F_a(D)\}$$ – средний случай на D_N

Классификация алгоритмов по виду функции трудоёмкости

Количественно-зависимые по трудоемкости алгоритмы

Алгоритмы, функция трудоемкости которых зависит только от размерности конкретного входа, и не зависит от конкретных значений:

$$F_a(D) = F_a(|D|) = F_a(N)$$

Параметрически-зависимые по трудоемкости алгоритмы

Алгоритмы, трудоемкость которых определяется не размерностью входа, а конкретными значениями обрабатываемых слов памяти:

$$F_a(D) = F_a(d_1,\ldots,d_n) = F_a(P_1,\ldots,P_m),\ m \le n$$

Количественно-параметрические по трудоемкости алгоритмы

В большинстве практических случаев функция трудоемкости зависит как от количества данных на входе, так и от значений входных данных, в этом случае:

$$F_a(D) = F_a(||D||, P_1, \ldots, P_m) = F_a(N, P_1, \ldots, P_m)$$

Порядково-зависимые по трудоемкости алгоритмы

Пусть множество D состоит из элементов $(d_1,\ldots,d_n)$, и $||D||=N$,

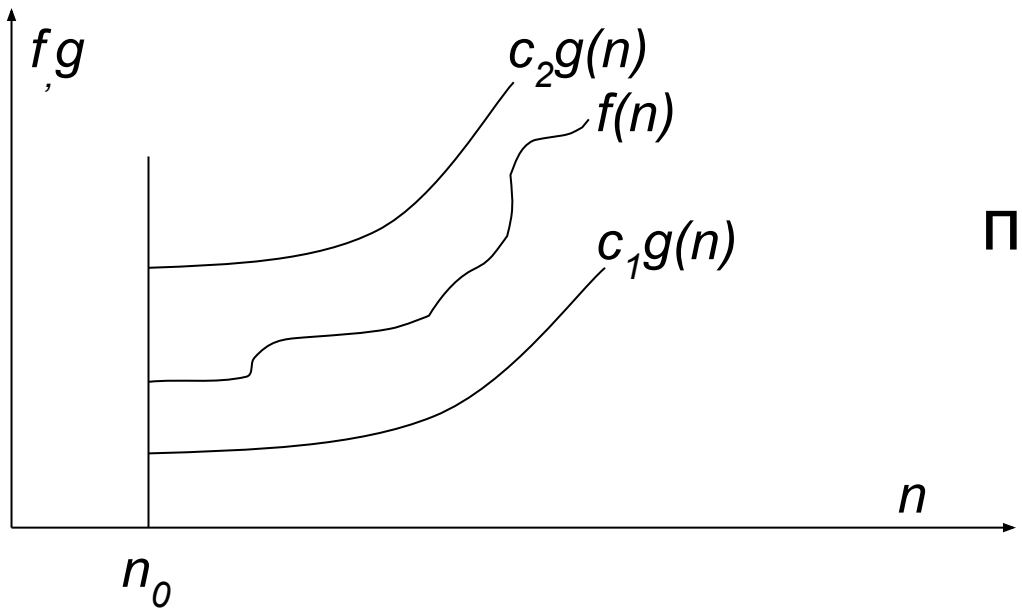
Определим $D_p = \{(d_1,\ldots,d_n)\}$-множество всех упорядоченных N-*ок* из $d_1,\ldots,d_n$, отметим, что $|D_p|=n!$.

Если $F_a({}_iD_p) \neq F_a({}_jD_p)$, где ${}_iD_p, {}_jD_p \in D_p$, то алгоритм будем называть порядково-зависимым по трудоемкости.

Асимптотический анализ функций

Оценка Θ (тетта)

Пусть *f(n)* и *g(n)* – положительные функции положительного аргумента, *n* ≥ *1* (количество объектов на входе и количество операций – положительные числа), тогда:

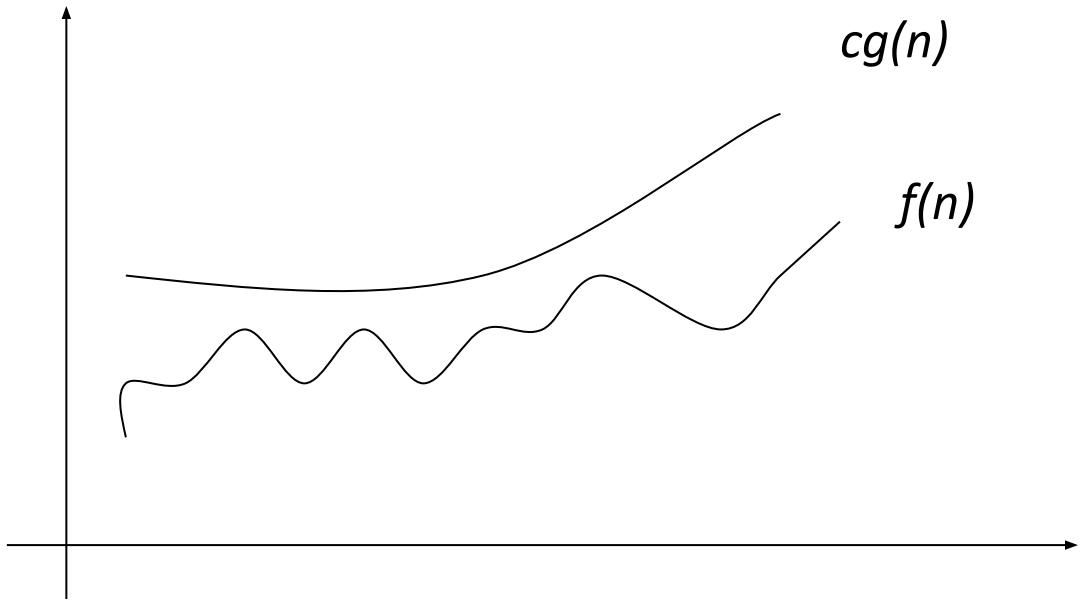


f(n) = Θ (g(n)), если существуют положительные c_1, c_2, n_0, такие, что:

$$c_1 * g(n) \le f(n) \le c_2 * g(n), \text{ при } n > n_0$$

Оценка *O* (*O* большое)

В отличие от оценки *Θ*, оценка *O* требует только, что бы функция *f(n)* не превышала *g(n)* начиная с *n > n0*, с точностью до постоянного множителя:

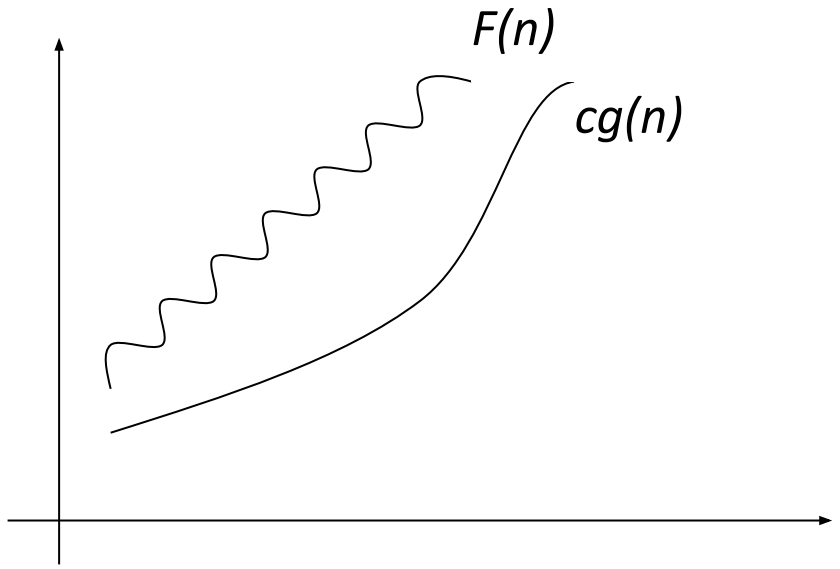


$$\exists\ c > 0,\ n0 > 0 :$$

$$0 \le f(n) \le c * g(n),\ \ \forall n > n0$$

Оценка Ω (Омега)

В отличие от оценки O, оценка Ω является оценкой снизу – т.е. определяет класс функций, которые растут не медленнее, чем $g(n)$ с точностью до постоянного множителя:



$$\exists\ c > 0,\ n0 > 0 :$$

$$0 \le c * g(n) \le f(n)$$

Временная оценка алгоритма

Проблемы построения временных оценок

- неадекватность формальной системы записи алгоритма и реальной системы команд процессора;
- наличие архитектурных особенностей существенно влияющих на наблюдаемое время выполнения программы, таких как конвейер, кеширование памяти, предвыборка команд и данных, и т.д.;
- различные времена выполнения реальных машинных команд;
- различие во времени выполнения одной команды, в зависимости от значений операндов;
- различные времена реального выполнения однородных команд в зависимости от типов данных;
- неоднозначности компиляции исходного текста, обусловленные как самим компилятором, так и его настройками.

Методики перехода к временным оценкам

Пооперационный анализ

Идея пооперационного анализа состоит в получении пооперационной функции трудоемкости для каждой из используемых алгоритмом элементарных операций с учетом типов данных.

Ожидаемое время выполнения рассчитывается как сумма произведений пооперационной трудоемкости на средние времена операций:

$$T_A(N) = \sum F_{aопi}(N) * t_{опi}$$

Метод Гиббсона

Метод предполагает проведение совокупного анализа по трудоемкости и переход к временным оценкам на основе принадлежности решаемой задачи к одному из типов

Далее на основе анализа множества реальных программ определяется частотная встречаемость операций

На основе полученной информации оценивается общее время работы алгоритма в виде:

$$T_A(N) = F_A(N) * \square t_{\text{тип задачи}}$$

Метод прямого определения среднего времени

В этом методе так же проводится совокупный анализ по трудоемкости – определяется $F_A(N)$, после чего на основе прямого эксперимента для различных значений $N_э$ определяется среднее время работы данной программы $T_э$ и на основе известной функции трудоемкости рассчитывается среднее время на обобщенную элементарную операцию, порождаемое данным алгоритмом, компилятором и компьютером – $\square t_a$.

$$\square t_a = T_э(N_э) / F_A(N_э),\ T(N) = \square t_a * F_A(N).$$

Классы сложности задач

Теоретический предел трудоемкости задачи

Рассматривая некоторую алгоритмически разрешимую задачу, и анализируя один из алгоритмов ее решения, мы можем получить оценку трудоемкости этого алгоритма в худшем случае –

$$f\hat{}_A(D_A)=O(g(D_A)).$$

Возникает вопрос: какова оценка сложности самого «быстрого» алгоритма решения данной задачи в худшем случае?

Определение понятия функционального теоретического нижнего предела трудоемкости задачи в худшем случае:

$$F_{thlim} = min \{ \Theta (F_a\hat{} (D)) \}$$

Если мы можем на основе теоретических рассуждений доказать существование и получить оценивающую функцию, то можно утверждать, что любой алгоритм, решающий данную задачу работает не быстрее, чем с оценкой F_{thlim} в худшем случае;

Класс P (задачи с полиномиальной сложностью)

Задача называется полиномиальной, т.е. относится к классу *P*, если существует константа *k* и алгоритм, решающий задачу с $F_a(n)=O(n^k)$, где *n* - длина входа алгоритма в битах $n = |D|$ [6].

Преимущества алгоритмов из этого класса:

- для большинства задач из класса *P* константа *k* меньше 6;
- класс *P* инвариантен по модели вычислений (для широкого класса моделей);
- класс *P* обладает свойством естественной замкнутости (сумма или произведение полиномов есть полином).

Класс NP (полиномиально проверяемые задачи)

Дано N чисел – $A = (a_1, \ldots a_n)$ и число V.

Задача: Найти вектор (массив) $X=(x_1, \ldots, x_n)$, $x_i \in \{0,1\}$, такой, что $\sum a_i x_i = V$.

Содержательно: может ли быть представлено число V в виде суммы каких либо элементов массива A.

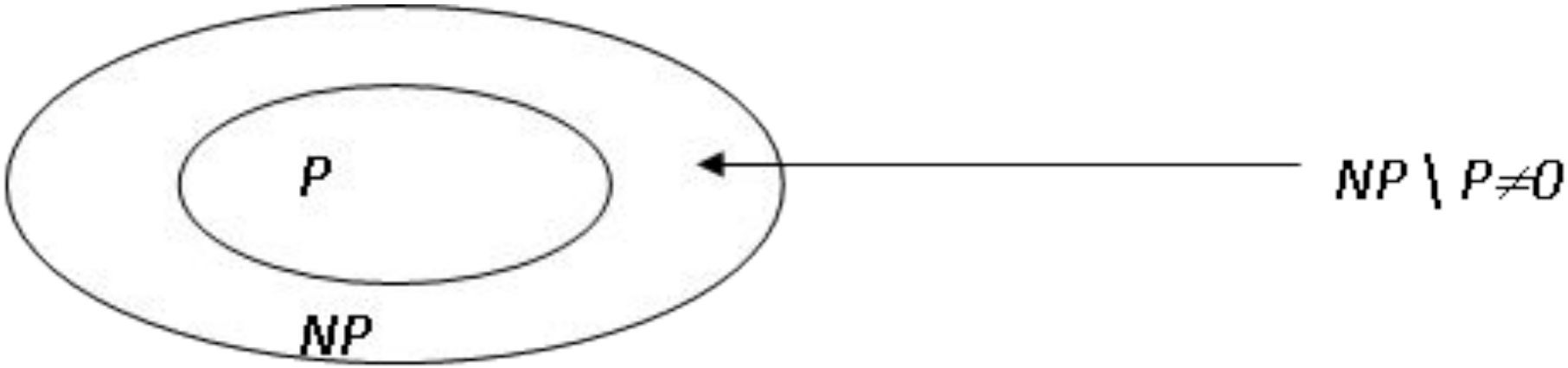
Если какой-то алгоритм выдает результат – массив X, то проверка правильности этого результата может быть выполнена с полиномиальной сложностью: проверка $\sum a_i x_i = V$ требует не более $\Theta(N)$ операций.

Формально: $\forall D \in D_A$, $|D|=n$ поставим в соответствие сертификат $S \in S_A$, такой что $|S|=O(n^l)$ и алгоритм $A_s = A_s(D,S)$, такой, что он выдает «1», если решение правильно, и «0», если решение неверно. Тогда задача принадлежит классу NP, если $F(A_s)=O(n^m)$ [6].

Содержательно задача относится к классу NP, если ее решение некоторым алгоритмом может быть быстро (полиномиально) проверено.

Проблема *P = NP*

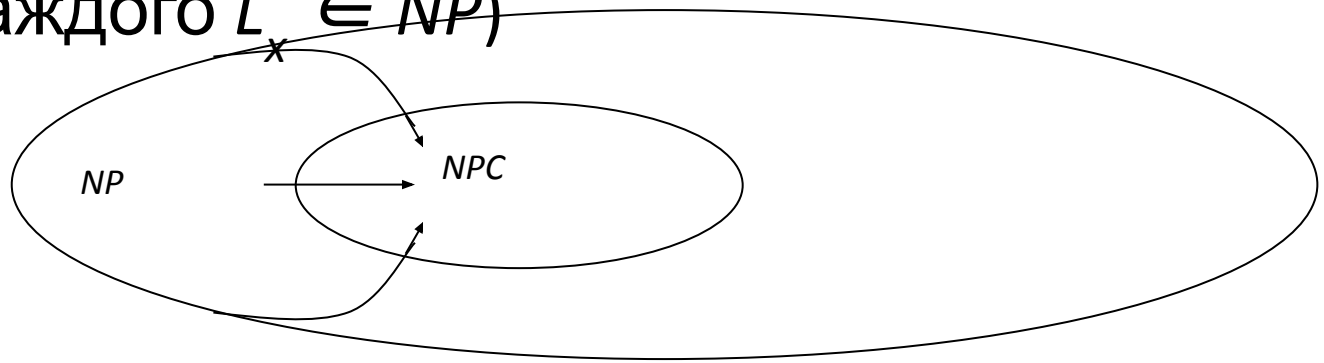
После введения в теорию алгоритмов понятий сложностных классов Эдмондсом была поставлена основная проблема теории сложности – *P = NP*



На сегодня отсутствуют теоретические доказательства как совпадения классов (*P=NP*), так и их несовпадения. Предположение состоит в том, что класс P является собственным подмножеством класса *NP*, т.е. *NP* \ *P* не пусто

Класс *NPC* (*NP* – полные задачи)

NPC (*NP*-complete) или клас *NP*-полных задач требует выполнения следующих двух условий: во-первых, задача должна принадлежать классу *NP* ($L \in NP$), и, во-вторых, к ней полиномиально должны сводиться все задачи из класса *NP* ($L_x \leq_P L$, для каждого $L_x \in NP$)



Для класса *NPC* доказана следующая теорема: Если существует задача, принадлежащая классу *NPC*, для которой существует полиномиальный алгоритм решения ($F = O(n^k)$), то класс *P* совпадает с классом *NP*, т.е. *P=NP* [6].

Алгоритм полного перебора

Полный перебор — метод решения математических задач. Относится к классу методов поиска решения исчерпыванием всевозможных вариантов.

Любая задача из класса NP может быть решена полным перебором. При этом, даже если вычисление целевой функции от каждого конкретного возможного решения задачи может быть осуществлено за полиномиальное время, в зависимости от количества всех возможных решений полный перебор может потребовать экспоненциального времени работы.

Полный перебор в большинстве прикладных задач на практике не применяется, есть ряд исключений. В частности, когда полный перебор всё же оказывается оптимальным, либо представляет собой *начальный этап* в разработке алгоритма, его использование оправдано.

Пример использования полного перебора

- Данный алгоритм используется для решения классической задачи динамического программирования — определения приоритетов вычислений матричных произведений следующего вида:$A_1 A_2 A_3 \ldots A_n$.

Исходная задача заключается в вычислении данной цепочки (матричного произведения) за наименьшее время. Поскольку матричное произведение является ассоциативной операцией, можно вычислить цепочечное произведение, произвольно выбирая пару элементов цепочки $(A_i A_{i+1}), \mathrm{i} = 1 \ldots n-1$ и заменяя её результирующей матрицей $A_i^1 : A_i^1 = (A_i A_{i+1})$. Если повторять описанную процедуру n-1 раз, то оставшаяся результирующая матрица A_k^{n-1} и будет ответом:

$A_k^{n-1} = \left(A_k^{n-2} A_{k+1}^{n-2}\right) = \cdots = A_1 A_2 A_3 \ldots A_n, k = 1 \ldots n-1.$

Эта формула может быть проиллюстрирована следующим образом.

Рассмотрим матричную цепочку: (A_1, A_2, A_3, A_4).

Существуют следующие 5 способов вычислить соответствующее этой цепочке произведение A_1, A_2, A_3, A_4:

$(A_1(A_2(A_3A_4)))$,

$(A_1((A_2A_3)A_4))$,

$((A_1A_2)(A_3A_4))$,

$((A_1(A_2A_3))A_4)$,

$(((A_1A_2)A_3)A_4)$.

Выбрав правильный порядок вычислений, можно добиться значительного ускорения вычислений. Чтобы убедиться в этом, рассмотрим простой пример цепочки из 3-х матриц. Положим, что их размеры равны соответственно 10×100,100×5,5×50.

- Стандартный алгоритм перемножения двух матриц размерами p×q,q×r требует время вычисления, пропорциональное числе pqr (число вычисляемых скалярных произведений). Следовательно, вычисляя цепочку в порядке$((A_1A_2)A_3)$, получаем 10*100*5=5000 скалярных произведений для вычисления (A_1A_2), плюс дополнительно 10*5*50=2500 скалярных произведений, чтобы вычислить второе матричное произведение. Общее число скалярных произведений: 7500. При ином выборе порядка вычислений получаем 100*5*50=25000+10*100*50=50000 скалярных произведений, то есть 75000 скалярных произведений.

Таким образом, решение данной задачи может существенно сократить временные затраты на вычисление матричной цепочки. Это решение может быть получено полным перебором: необходимо рассмотреть все возможные последовательности вычислений и выбрать из них ту, которая при вычислении цепочки занимает наименьшее число скалярных произведений. Однако надо учитывать, что этот алгоритм сам по себе требует экспоненциальное время вычисления, так что для длинных матричных цепочек выигрыш от вычисления цепочки самым эффективным образом (оптимальная стратегия) может быть полностью потерян временем нахождения этой стратегии.

Метод “разделяй и властвуй”

Описание метода

Задача решается в три стадии:

- задача разбивается на несколько более простых подзадач (как правило, независимых);
- каждая подзадача решается рекурсивно;
- исходя из ответов для подзадач, строится ответ для исходной задачи

Простейший пример — сортировка слиянием

Умножение чисел

- Необходимо вычислить произведение двух n-битных чисел x и y. Это несложно сделать за время $O(n^2)$ умножением в столбик. Методом “разделяй и властвуй” можно построить алгоритм быстрее. Разобьем числа x и y на две примерно одинаковые по длине части:

$$x = 2^{n/2}x_L + x_R, \mathrm{y} = 2^{n/2}\mathrm{y}_L + \mathrm{y}_R.$$

Тогда

$$xy = \left(2^{n/2}x_L + x_R\right)\left(2^{n/2}y_L + y_R\right) = 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R$$

- Сложение чисел и домножение на степень двойки производятся за линейное время.

Обозначив через T(n) время умножения двух n-битных чисел, получаем следующее рекуррентное соотношение

$$T(n) = 4T(n/2) + O(n).$$

Решив, получим, что T(n) = $O(n^2)$.Таким образом, рассмотренный рекурсивный алгоритм не лучше алгоритма умножения в столбик.

Более эффективный алгоритм

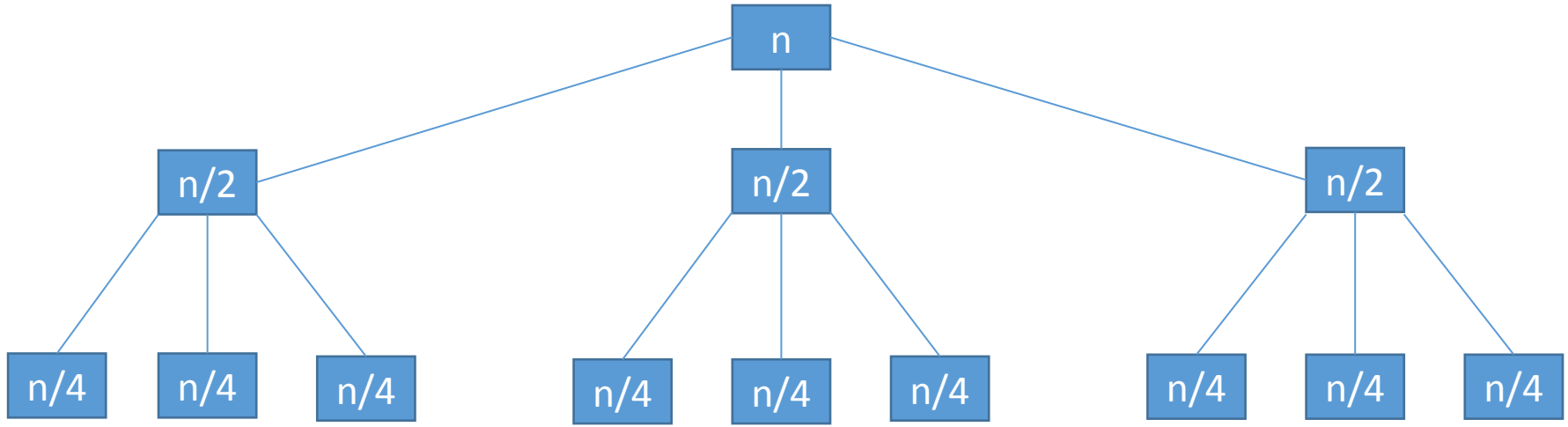
Заметим, однако, что для вычисления $x_L y_L, x_L y_R + x_R y_L, x_R y_R$ достаточно и трёх умножений: $x_L y_L$, $x_R y_R$, $(x_L + x_R)(y_L + y_R)$

поскольку

$$x_L y_R + x_R y_L = (x_L + x_R)\,(y_L + y_R) - x_L y_L - x_R y_R.$$

Время работы соответствующего алгоритма удовлетворяет соотношению T(n) = 3T(n/2) + O(n), а значит, T(n) = $O(n^{1.59})$

Дерево работы алгоритма



- Высота дерева равна log2 n
- На k-м уровне дерева всего 3^k подзадач. Для определения следующих подзадач и соединения ответов требуется линейное время.
- Таким образом, на k-м уровне дерева делается

$$3^k \times O(\frac{n}{2^k}) = \left(\frac{3}{2}\right)^k \times O(n) \text{операций.}$$

- Количество выполняемых операций на уровне с ростом k растет геометрически с коэффициентом 3/2, значит, с точностью до константы общее количество операций равно

$$\left(\frac{3}{z}\right)^{\log_2 n} \times O(n) = O\left(3^{\log_2 n}\right) = O\left(3^{\log_2 3}\right)$$

Жадные алгоритмы

Общая идея

- На каждом шаге жадный алгоритм делает локально оптимальный выбор.
- Решение, найденное таким образом, не всегда оказывается оптимальным, но в ряде случаев все-таки оказывается.
- Общеизвестный пример — алгоритм для построения минимального покрывающего дерева: на каждом шаге берется самое легкое из возможных ребер.

Задача о выборе заявок

Даны n пар чисел (si, fi), где si < fi обозначает время начала занятия, а fi — конец.

Говорим, что заявки (si, fi) и (sj , fj) совместны, если интервалы [si, fi) и [sj , fj) не пересекаются: fi ≤ sj или fj ≤ si.

Задача о выборе заявок (activity-selection problem) заключается в выборе максимального количества совместных друг с другом заявок

Замечание

Будем считать, что заявки отсортированы в порядке возрастания времени окончания: f1 ≤ f2 ≤ · · · ≤ fn.

Алгоритм

Greedy-Activity-Selector(s, f)

1 n ← length[s]

2 A ← {1}

3 j ← 1

4 for i ← 2 to n

5 do if si ≥ fj

6 then A ← A ∪ {i}

7 j ← i

8 return A

Анализ

- Время работы есть $\Theta(n)$ (при условии, что заявки отсортированы!).
- Существует оптимальное решение, содержащее заявку 1: если в некотором оптимальном множестве заявка 1 не содержится, то первую заявку (то есть заявку с самым ранним временем окончания) этого множества можно заменить на заявку 1. При такой операции совместность не нарушится, а количество заявок останется прежним.
- Итак, мы ищем решение, содержащее заявку 1. Значит, можно выкинуть все заявки, несовместные с ней.
- Получаем подзадачу с меньшим количеством заявок.

Две отличительные особенности жадных алгоритмов

- Принцип жадного выбора: последовательность локально оптимальных (жадных) выборов дает глобально оптимальное решение. Для установления этого свойства, как правило, нужно показать, что жадный выбор согласован с некоторым оптимальным решением и что после выбора получается аналогичная подзадача.
- Свойство оптимальности для подзадач: оптимальное решение для задачи содержит оптимальные решения для подзадач.

Жадный алгоритм или динамическое программирование?

В непрерывной задаче о рюкзаке (fractional knapsack problem), в отличие от общей задачи о рюкзаке, в рюкзак разрешать класть части предметов

- Нетрудно видеть, что жадный алгоритм находит оптимальное решение для непрерывной задачи о рюкзаке: на каждом шаге добавляем максимальное количество предмета максимальной удельной стоимости (стоимость/объём).
- Нетрудно также убедиться в том, что аналогичный алгоритм для общей задачи может и не найти оптимального решения: сразу положив в рюкзак самый дорогой предмет, мы можем потерять возможность полностью заполнить рюкзак.

Жадный алгоритм или динамическое программирование?

- Итак, в первом случае выполняется принцип жадного выбора, во втором — нет. Поэтому непрерывная задача решается жадным алгоритмом, общая — динамическим программированием.
- Слегка модифицировав жадный алгоритм для общей задачи о рюкзаке, можно получить 2-приближенный алгоритм.

Алгоритм перебора с возвратом

Обзор метода

Решение задачи *методом перебора* с возвратом строится конструктивно последовательным расширением частичного решения. Если на конкретном шаге такое расширение провести не удается, то происходит возврат к более короткому частичному решению, и попытки его расширить продолжаются.

При использовании метода решета вместо конструктивного построения решений задачи из *множества* возможных вариантов исключаются все элементы, не являющиеся решениями.

Вычислительная схема перебора с возвратом

- Пусть M_0, M_1, ...,M_{n-1} - n конечных линейно упорядоченных множеств и G - совокупность ограничений (условий), ставящих в соответствие векторам вида

$$v = \left(v_{\nabla_1} v_{1,\ldots}, v\right)^T \left(v_J \in M_J; j = 0,1, \ldots, k; k \Leftarrow n-1\right),$$

булево *значение* $G(v)\epsilon$ { *истина,ложь* } .Векторы $v=(v_0,v_1,\ldots,v_k)^T$, для которых G(v) = *истина*, назовем *частичными решениями*.

Пусть, далее, существует конкретное правило P, в соответствии с которым некоторые из частичных решений могут объявляться полными решениями. Тогда возможна постановка следующих поисковых задач.

- Найти все полные решения или установить отсутствие таковых.
- Найти хотя бы одно полное решение или установить его отсутствие.

Общий *метод решения* приведенных задач состоит в последовательном покомпонентном наращивании вектора v слева направо, начиная с v_0, и последующих проверках его ограничениями G и правилом P.

Задача о расстановке ферзей на шахматной доске.

Составьте *рекурсивную функцию,* находящую возможную расстановку n ферзей на шахматной доске размером nxn так, чтобы они не били друг друга (n – *натуральное число*).

В соответствии с общей схемой алгоритма перебора с возвратом предложенную задачу можно решать по алгоритму "Все расстановки", но завершить выполнение алгоритма при нахождении первой требуемой расстановки или при получении вывода о невозможности получить нужную комбинацию. Согласно алгоритму ферзи расставляются последовательно на вертикалях с номерами от нуля и далее. В процессе выполнения предписания возможны снятия ферзей с доски (возвраты).

Алгоритм "Все расстановки"

- Шаг 1. Полагаем $D = \emptyset, j = 0$(D – множество решений, j – текущий столбец для очередного ферзя).

Шаг 2. Пытаемся в столбце j продвинуть вниз по вертикали или новый (если столбец j пустой), или уже имеющийся там ферзь на ближайшую допустимую строку. Если это сделать не удалось, то переходим к шагу 4.

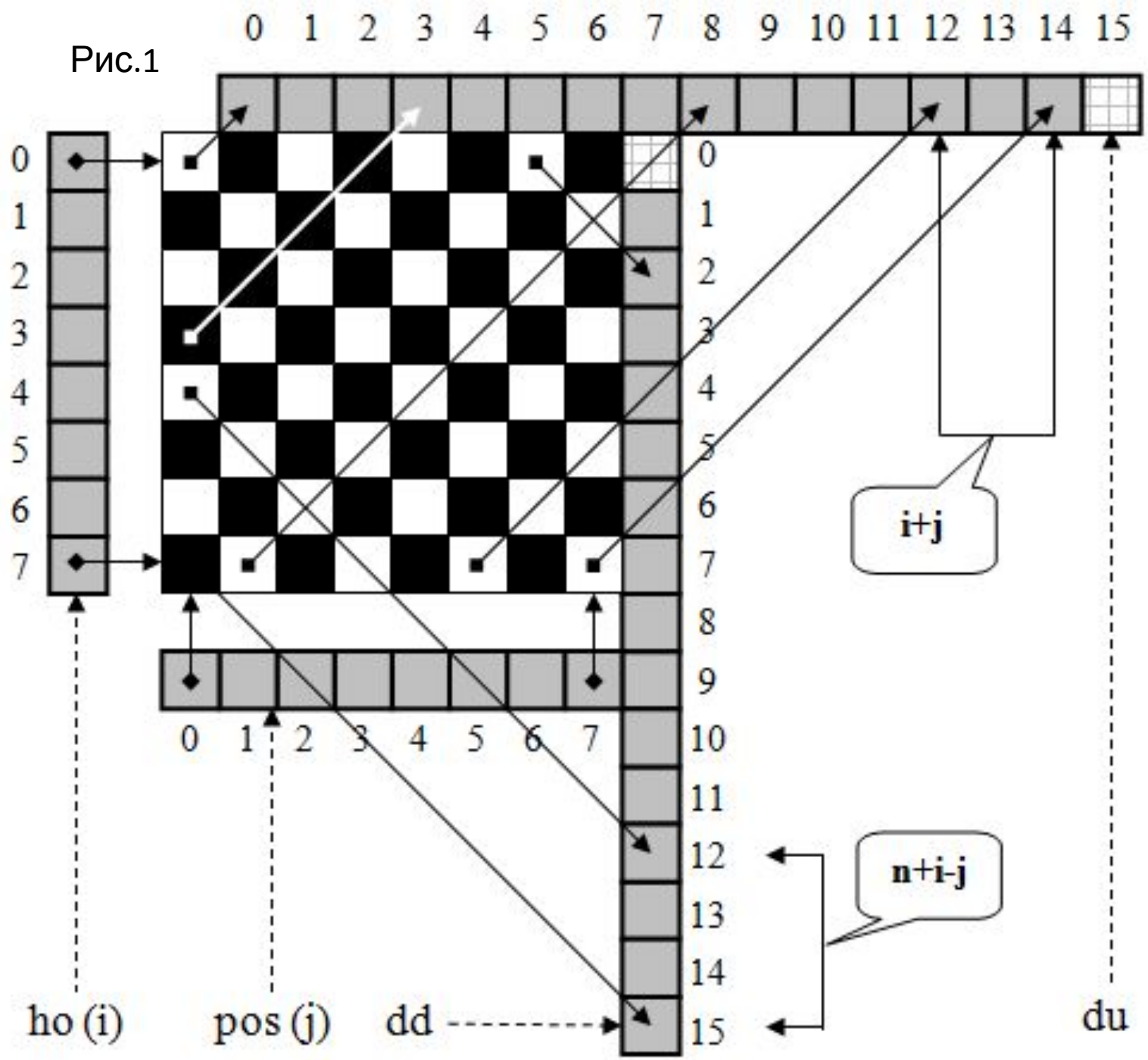
Шаг 3. Увеличиваем j на 1, то есть переходим к следующему столбцу. Если j<n-1, то переходим к шагу 2. В противном случае j=n-1, то есть все вертикали уже заняты. Найденное частичное решение запоминаем в множестве D и переходим к шагу 2.

Шаг 4. Уменьшаем j на 1, то есть снимаем ферзь со столбца j и переходим к предыдущему столбцу. Если j>0, то выполняем шаг 2. Иначе вычисления прекращаем. Решения задачи находятся в множестве D, которое, может быть и пустым.

Проведем *параметризацию* задачи. Введем четыре вспомогательных вектора: *pos*, ho, dd и du с длинами n, n, 2n-1 и 2n-1 соответственно. Использовать их будем следующим образом рис.1

- ho_i=1, если на горизонтали с номером i (i=0,1,...n-1) имеется ферзь, и ho_i=0 - в противном случае;
- du_s=1, если на диагонали с номером s (s=0,1,...2n-2), идущей слева направо и снизу вверх, имеется ферзь, и du_s=0 - в противном случае;
- dd_s=1, если на диагонали с номером s (s=0,1,...2n-1), идущей слева направо и сверху вниз, имеется ферзь, и dd_{s+1}=0 - в противном случае;
- pos_j=i, если в позиции (i,j) (i,j=0,1,...n-1) стоит ферзь.

Рис.1


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
0 1 2 3 4 5 6 7
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
i+j
n+i-j
ho (i)
pos (j)
dd
du

Использование этих соглашений позволяет получить такие утверждения:

- В позицию (i,j) можно поставить ферзь, если $ho_i+du_{i+j}+dd_{n+i-j}=0$.
- Поставить ферзь в позицию (i,j) равносильно присваиваниям: $ho_i=1$, $du_{i+j}=1$, $dd_{n+i-j}=1$.
- Убрать ферзь из позиции (i,j) равносильно присваиваниям: $ho_i=0$, $du_{i+j}=0$, $dd_{n+i-j}=0$.

Данное описание алгоритма является моделью решения общей задачи о нахождении всех вариантов расстановок.

Алгоритм перебора с возвратом

Обзор метода

Решение задачи *методом перебора* с возвратом строится конструктивно последовательным расширением частичного решения. Если на конкретном шаге такое расширение провести не удается, то происходит возврат к более короткому частичному решению, и попытки его расширить продолжаются.

При использовании метода решета вместо конструктивного построения решений задачи из *множества* возможных вариантов исключаются все элементы, не являющиеся решениями.

Вычислительная схема перебора с возвратом

- Пусть M_0, M_1, ...,M_{n-1} - n конечных линейно упорядоченных множеств и G - совокупность ограничений (условий), ставящих в соответствие векторам вида

$$v = \left(v_{\nabla_1} v_{1,\ldots}, v\right)^T \left(v_J \in M_J; j = 0,1, \ldots, k; k \Leftarrow n-1\right),$$

булево *значение* $G(v)\epsilon$ { *истина,ложь* } .Векторы $v=(v_0,v_1,\ldots,v_k)^T$, для которых G(v) = *истина*, назовем *частичными решениями*.

Пусть, далее, существует конкретное правило P, в соответствии с которым некоторые из частичных решений могут объявляться полными решениями. Тогда возможна постановка следующих поисковых задач.

- Найти все полные решения или установить отсутствие таковых.
- Найти хотя бы одно полное решение или установить его отсутствие.

Общий *метод решения* приведенных задач состоит в последовательном покомпонентном наращивании вектора v слева направо, начиная с v_0, и последующих проверках его ограничениями G и правилом P.

Задача о расстановке ферзей на шахматной доске.

Составьте *рекурсивную функцию,* находящую возможную расстановку n ферзей на шахматной доске размером nxn так, чтобы они не били друг друга (n – *натуральное число*).

В соответствии с общей схемой алгоритма перебора с возвратом предложенную задачу можно решать по алгоритму "Все расстановки", но завершить выполнение алгоритма при нахождении первой требуемой расстановки или при получении вывода о невозможности получить нужную комбинацию. Согласно алгоритму ферзи расставляются последовательно на вертикалях с номерами от нуля и далее. В процессе выполнения предписания возможны снятия ферзей с доски (возвраты).

Алгоритм "Все расстановки"

- Шаг 1. Полагаем $D = \emptyset, j = 0$(D – множество решений, j – текущий столбец для очередного ферзя).

Шаг 2. Пытаемся в столбце j продвинуть вниз по вертикали или новый (если столбец j пустой), или уже имеющийся там ферзь на ближайшую допустимую строку. Если это сделать не удалось, то переходим к шагу 4.

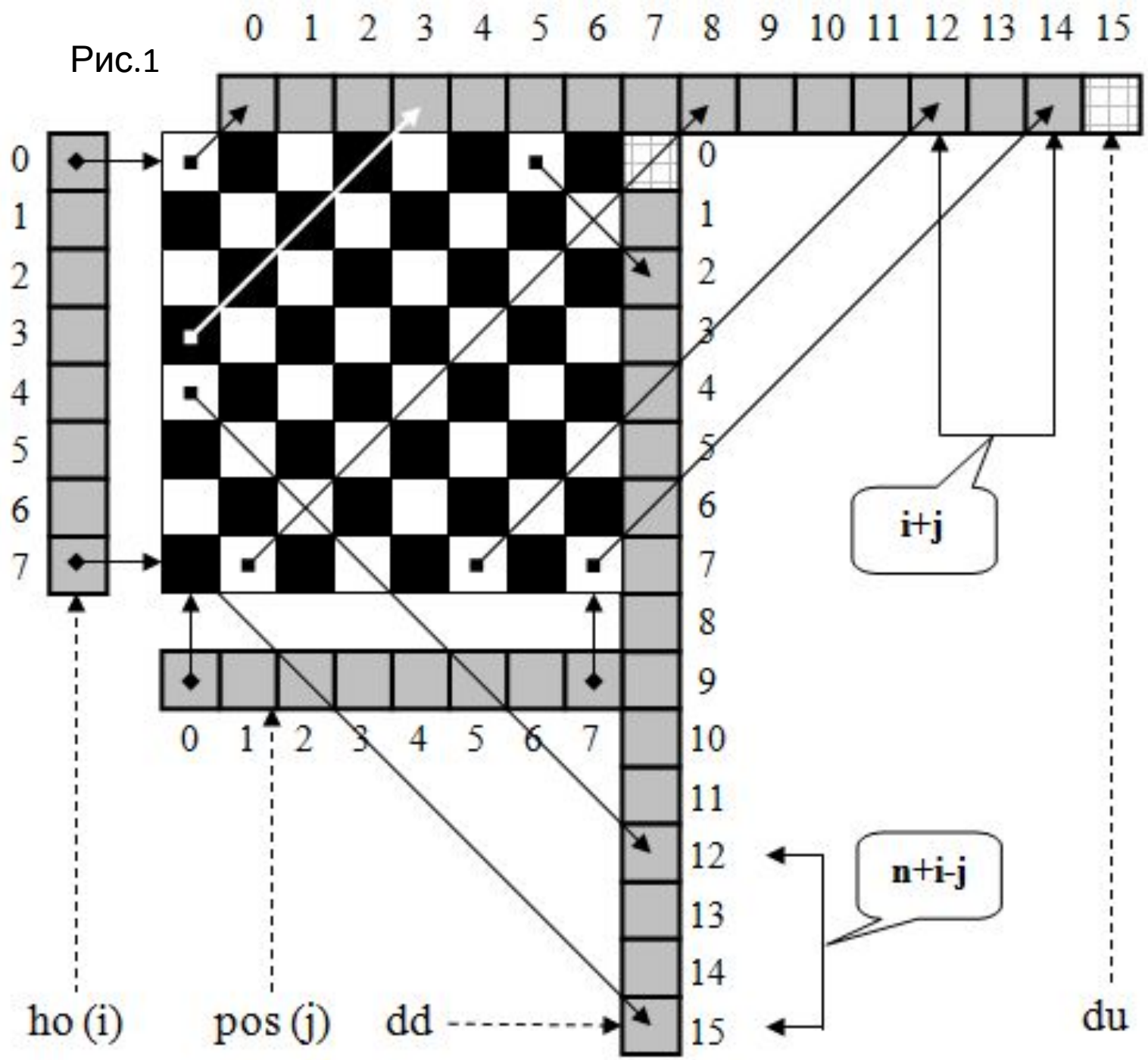
Шаг 3. Увеличиваем j на 1, то есть переходим к следующему столбцу. Если j<n-1, то переходим к шагу 2. В противном случае j=n-1, то есть все вертикали уже заняты. Найденное частичное решение запоминаем в множестве D и переходим к шагу 2.

Шаг 4. Уменьшаем j на 1, то есть снимаем ферзь со столбца j и переходим к предыдущему столбцу. Если j>0, то выполняем шаг 2. Иначе вычисления прекращаем. Решения задачи находятся в множестве D, которое, может быть и пустым.

Проведем *параметризацию* задачи. Введем четыре вспомогательных вектора: *pos*, ho, dd и du с длинами n, n, 2n-1 и 2n-1 соответственно. Использовать их будем следующим образом рис.1

- ho_i=1, если на горизонтали с номером i (i=0,1,...n-1) имеется ферзь, и ho_i=0 - в противном случае;
- du_s=1, если на диагонали с номером s (s=0,1,...2n-2), идущей слева направо и снизу вверх, имеется ферзь, и du_s=0 - в противном случае;
- dd_s=1, если на диагонали с номером s (s=0,1,...2n-1), идущей слева направо и сверху вниз, имеется ферзь, и dd_{s+1}=0 - в противном случае;
- pos_j=i, если в позиции (i,j) (i,j=0,1,...n-1) стоит ферзь.

Рис.1


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
0 1 2 3 4 5 6 7
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
i+j
n+i-j
ho (i)
pos (j)
dd
du

Использование этих соглашений позволяет получить такие утверждения:

- В позицию (i,j) можно поставить ферзь, если $ho_i+du_{i+j}+dd_{n+i-j}=0$.
- Поставить ферзь в позицию (i,j) равносильно присваиваниям: $ho_i=1$, $du_{i+j}=1$, $dd_{n+i-j}=1$.
- Убрать ферзь из позиции (i,j) равносильно присваиваниям: $ho_i=0$, $du_{i+j}=0$, $dd_{n+i-j}=0$.

Данное описание алгоритма является моделью решения общей задачи о нахождении всех вариантов расстановок.

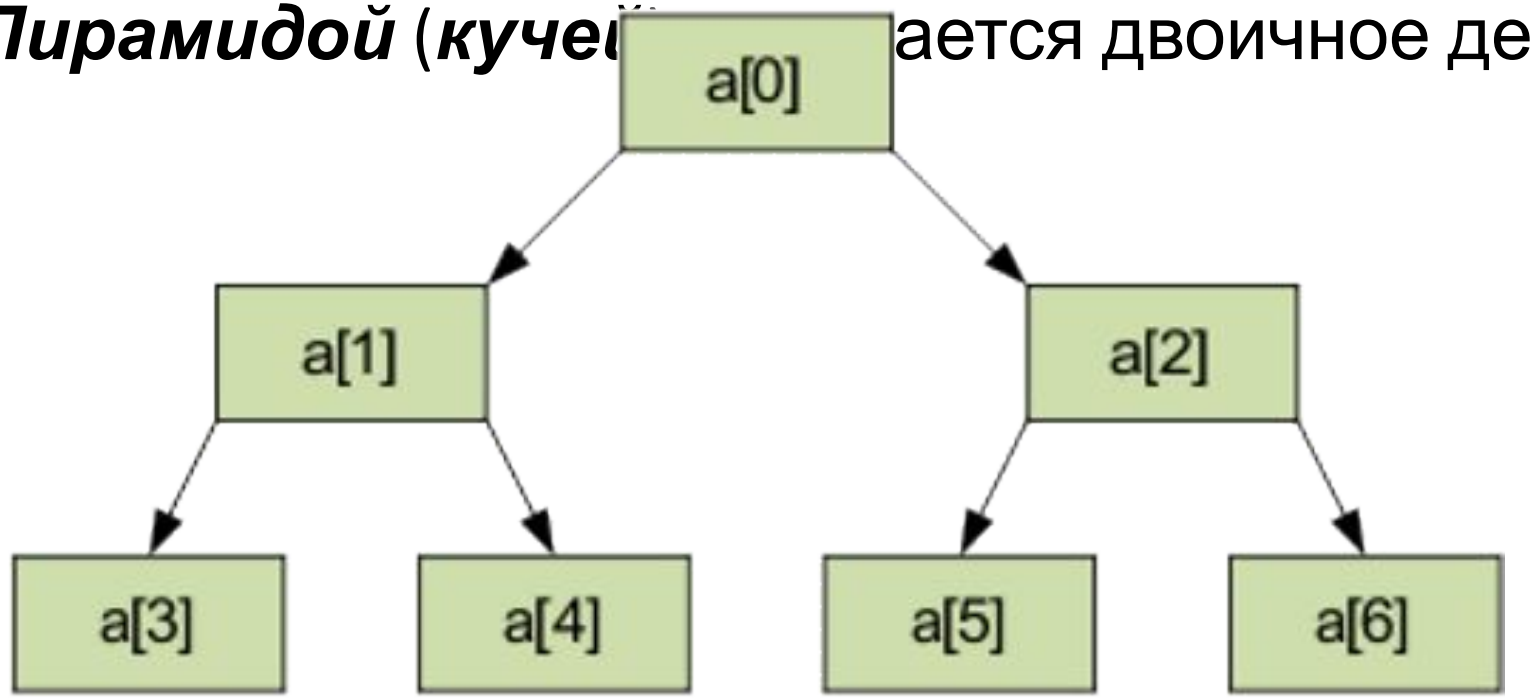
Алгоритм пирамидальной сортировки

Метод пирамидальной сортировки, изобретенный Д. Уилльямсом, является улучшением традиционных сортировок с помощью дерева.

Пирамидой (***кучей***) называется двоичное дерево такое, что

a[i] ≤ a[2i+1];

a[i] ≤ a[2i+2].



a[0] — минимальный элемент пирамиды.

Общая идея алгоритма

Общая идея пирамидальной сортировки заключается в том, что сначала строится пирамида из элементов исходного массива, а затем осуществляется сортировка элементов.

Выполнение алгоритма разбивается на два этапа.

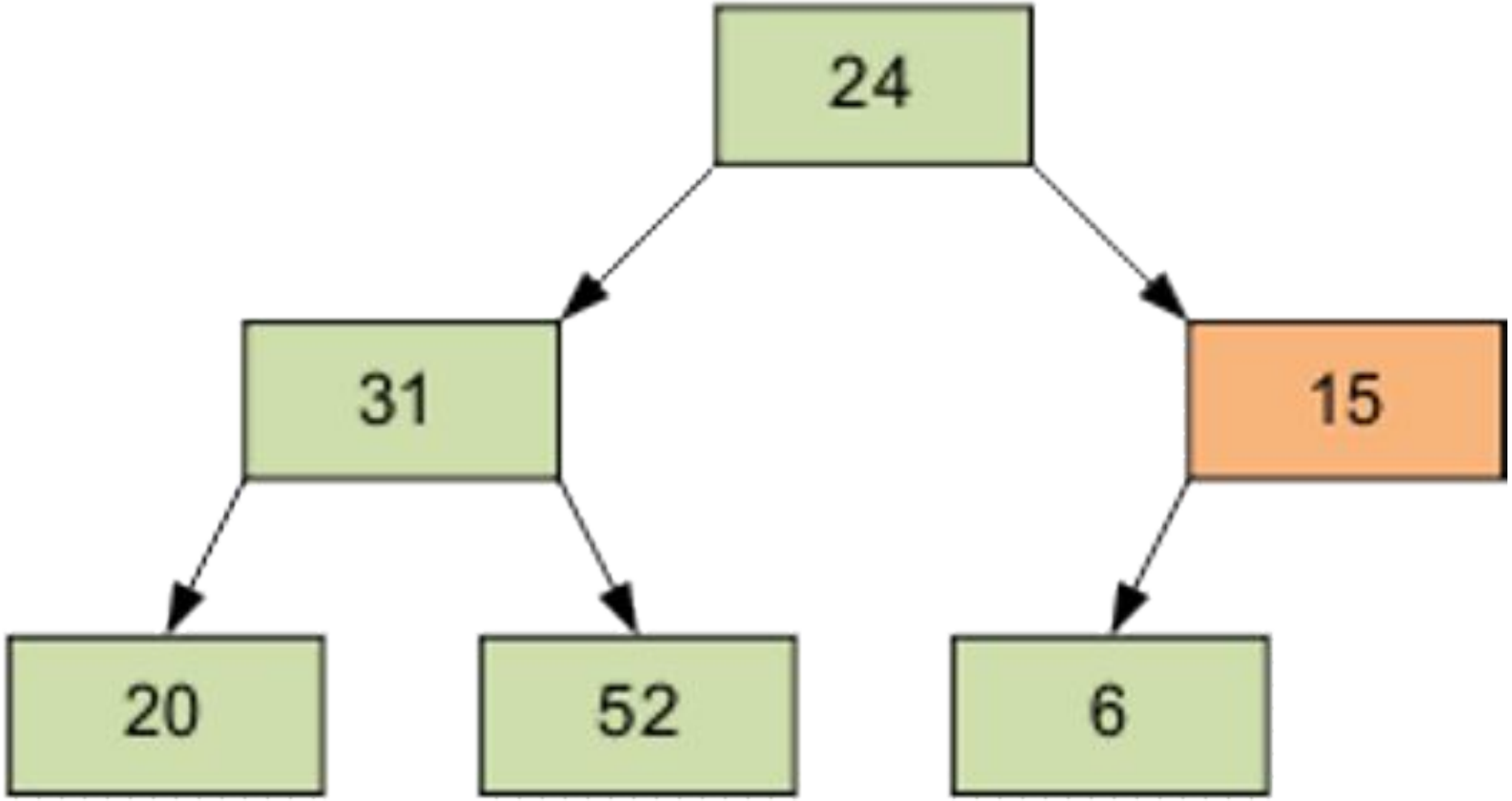
1 этап

Построение пирамиды. Определяем правую часть дерева, начиная с **n/2-1** (нижний уровень дерева). Берем элемент левее этой части массива и просеиваем его сквозь пирамиду по пути, где находятся меньшие его элементы, которые одновременно поднимаются вверх; из двух возможных путей выбираете путь через меньший элемент.

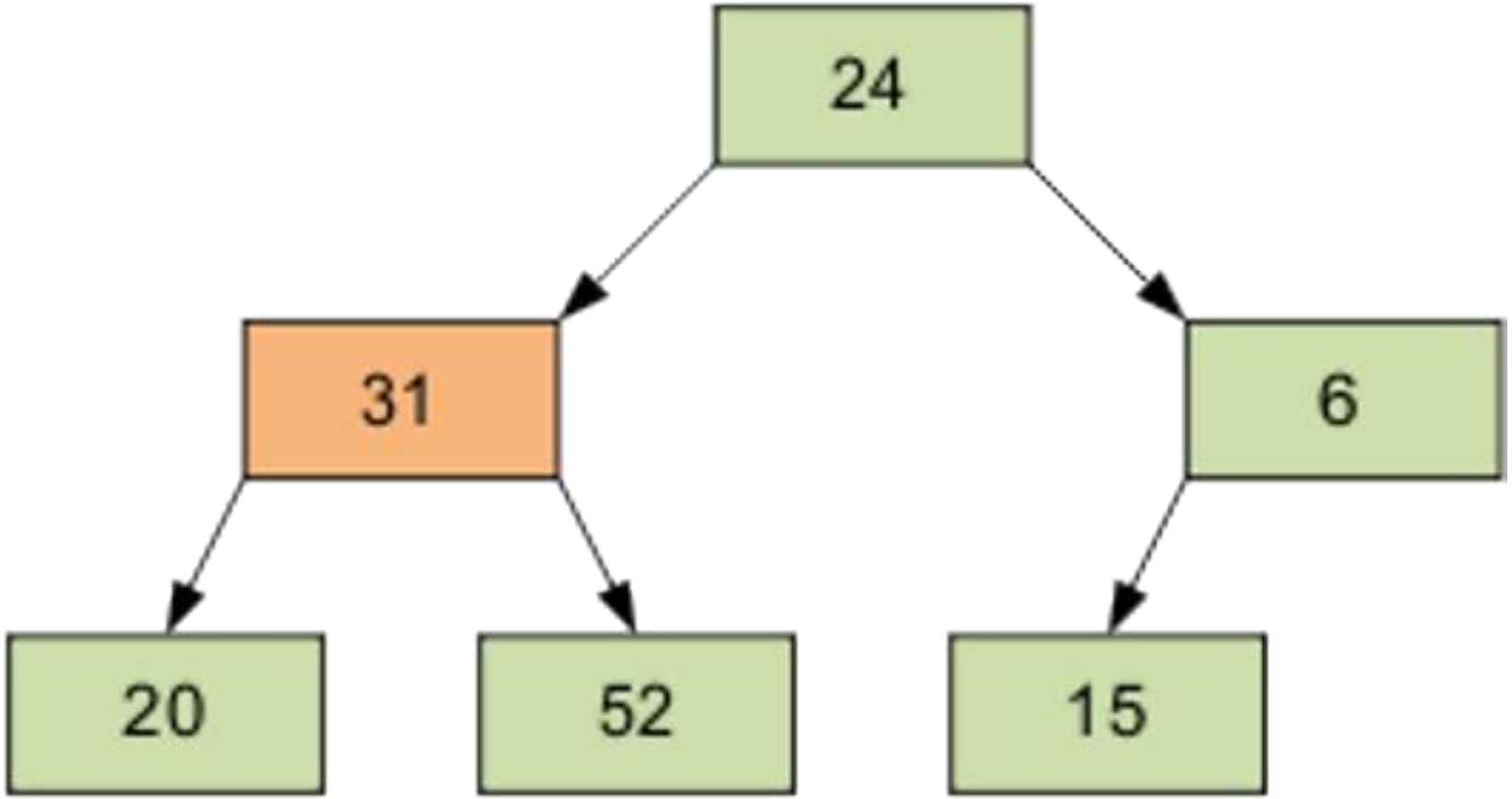
Например, массив для сортировки:

24, 31, 15, 20, 52, 6

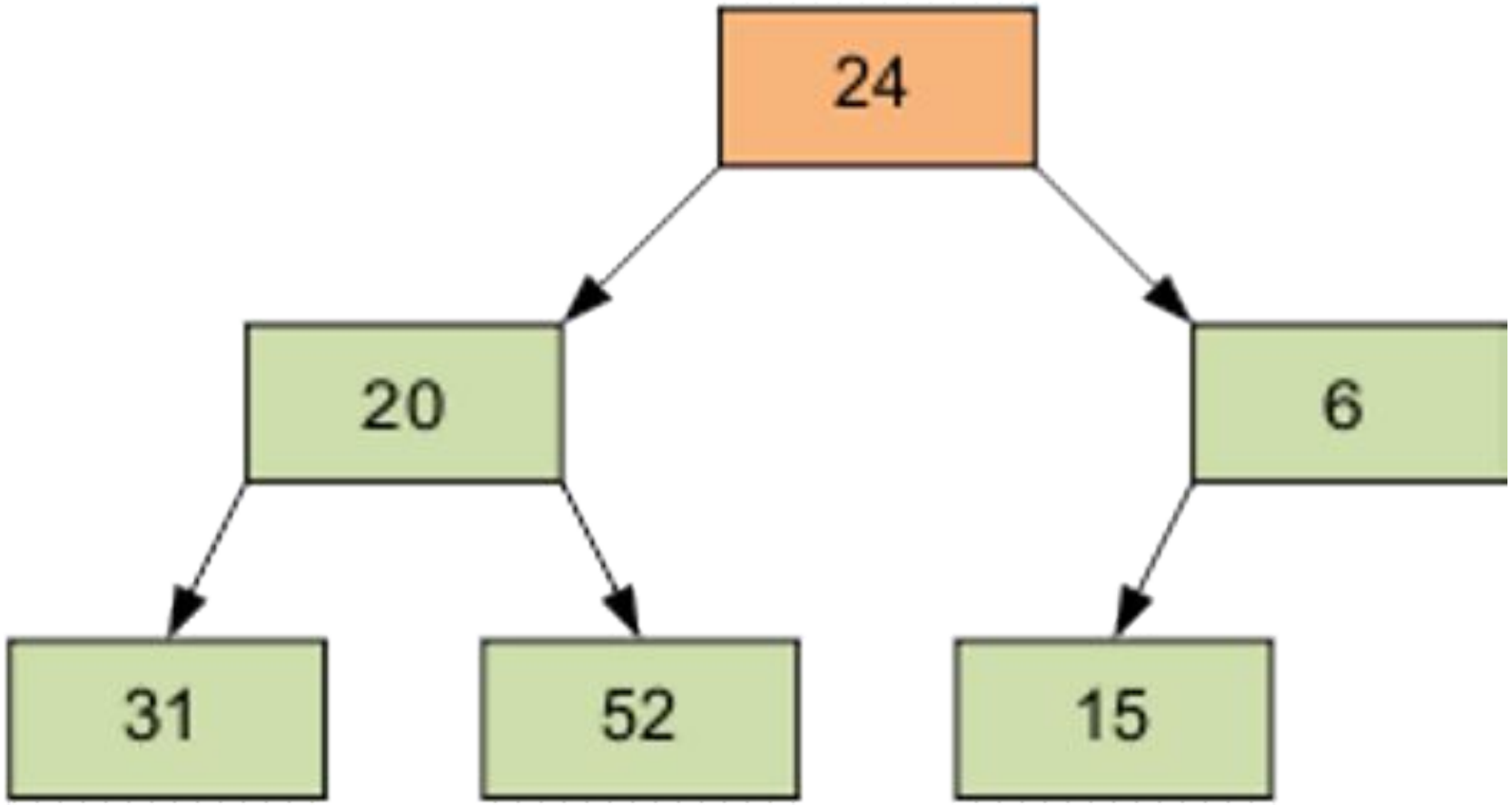
Расположим элементы в виде исходной пирамиды; номер элемента правой части (6/2-1)=2 - это элемент 15.



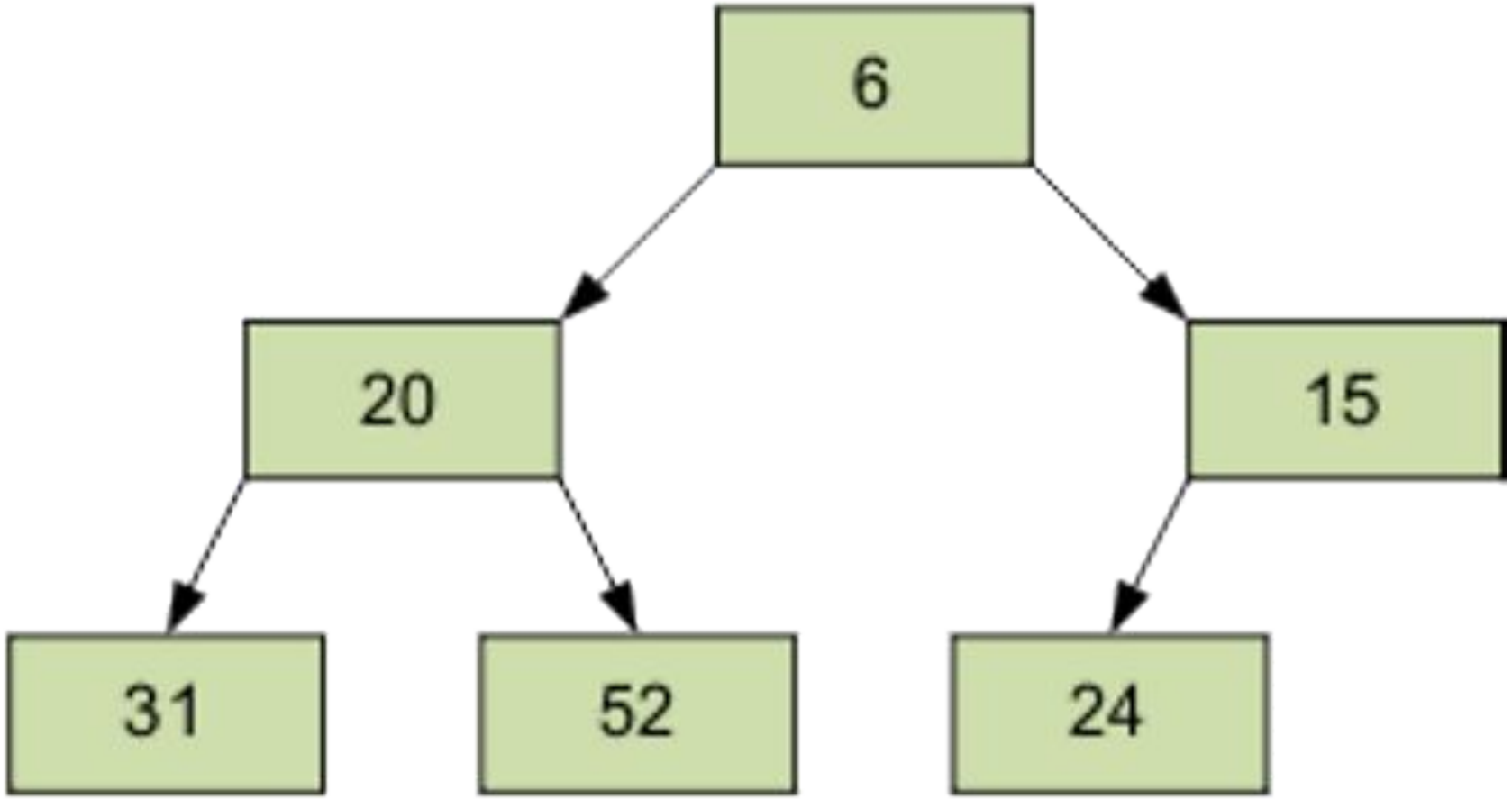
Результат просеивания элемента 15 через пирамиду.



Следующий просеиваемый элемент – 1, равный 31.



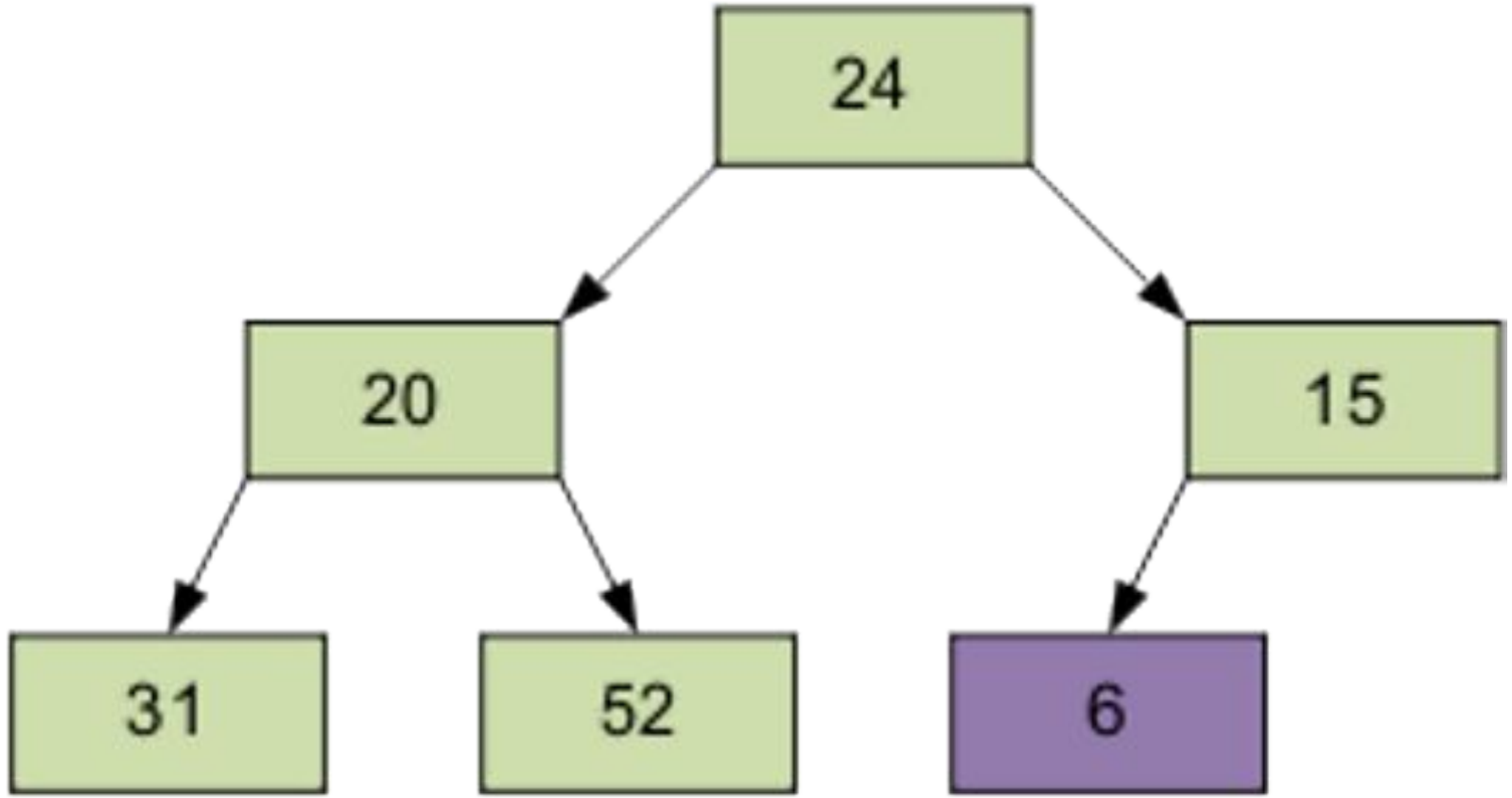
Затем – элемент 0, равный 24.

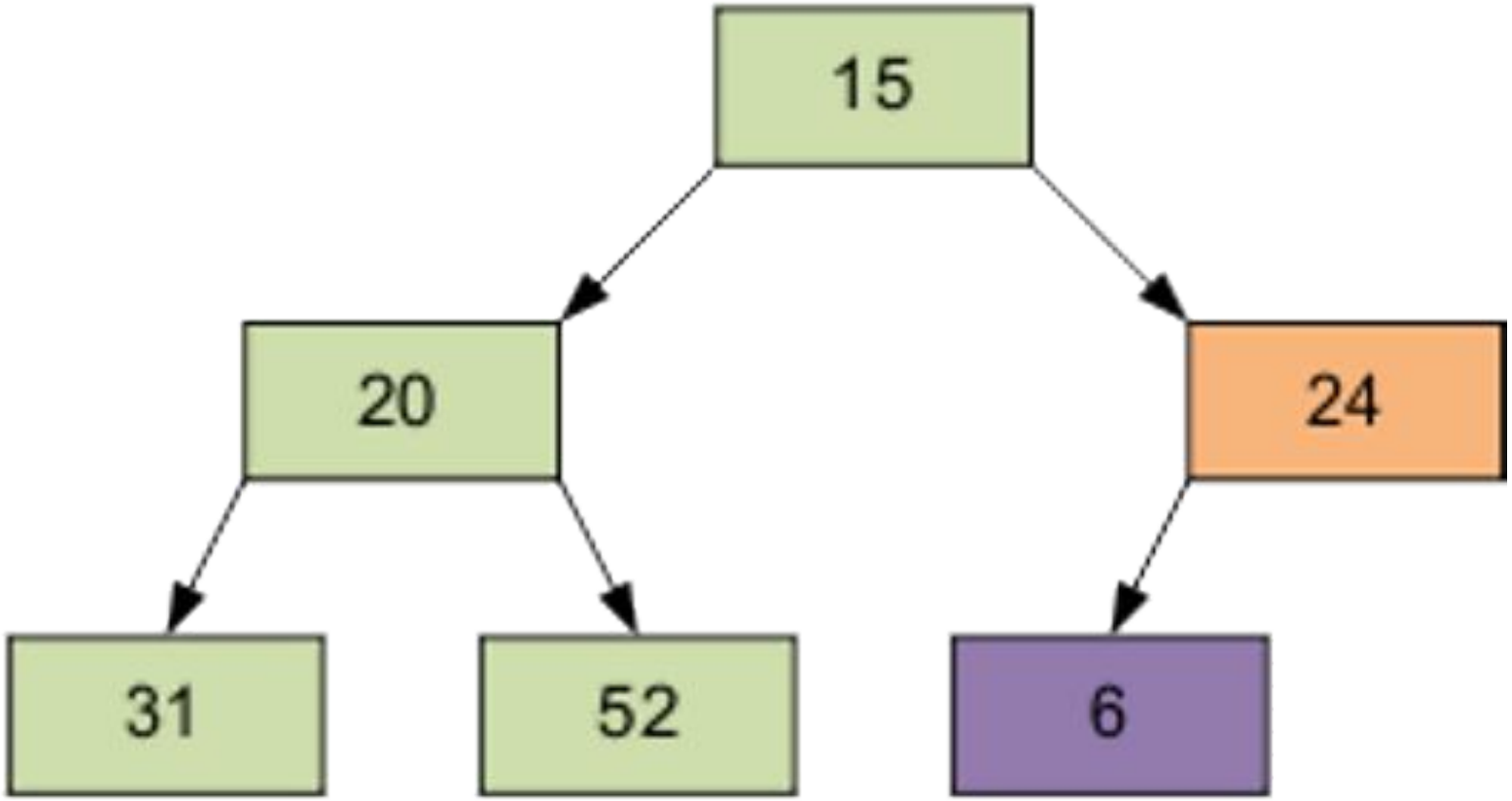


Разумеется, полученный массив еще не упорядочен. Однако процедура просеивания является основой для пирамидальной сортировки. В итоге просеивания наименьший элемент оказывается на вершине пирамиды.

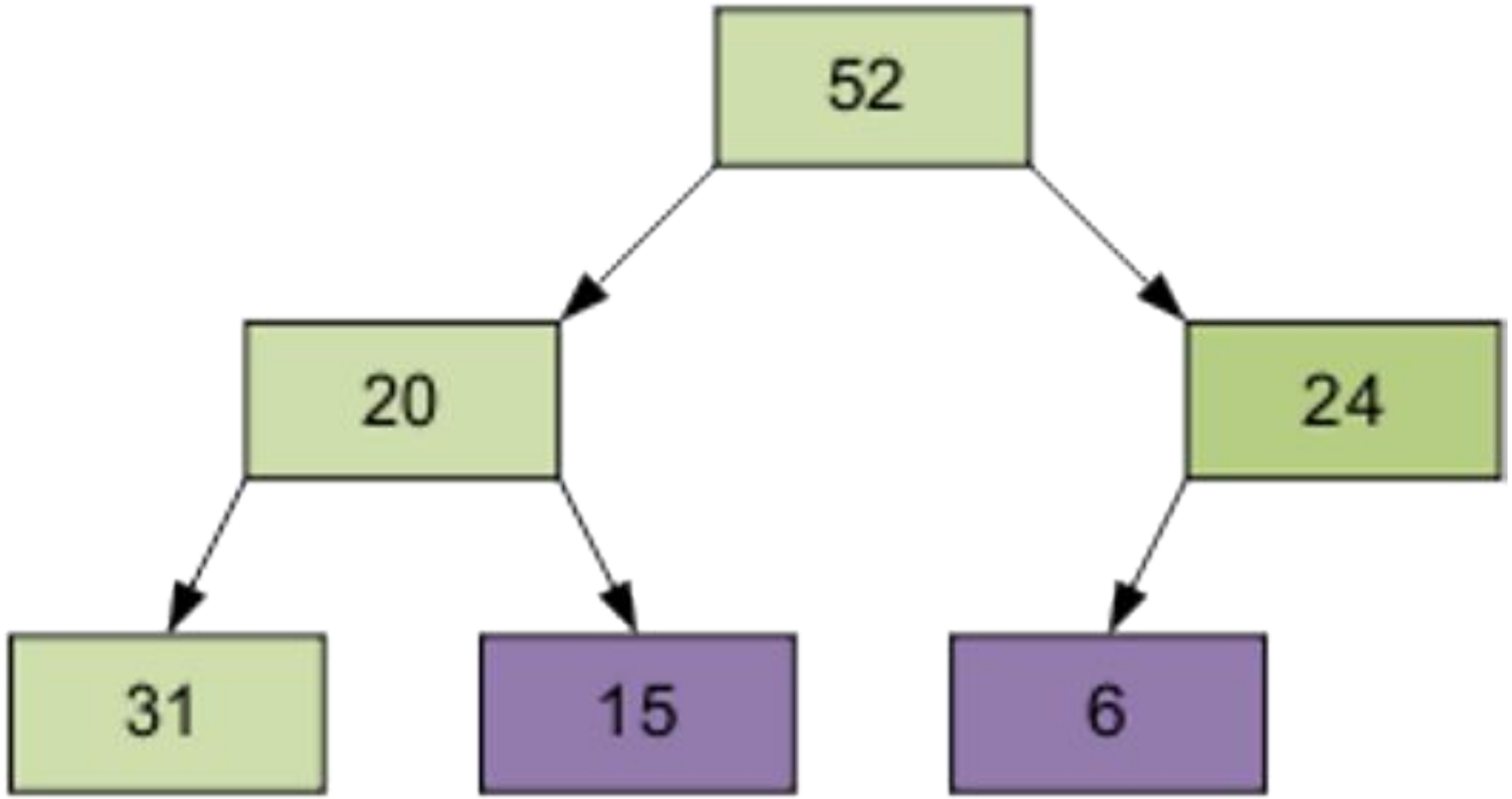
2 этап

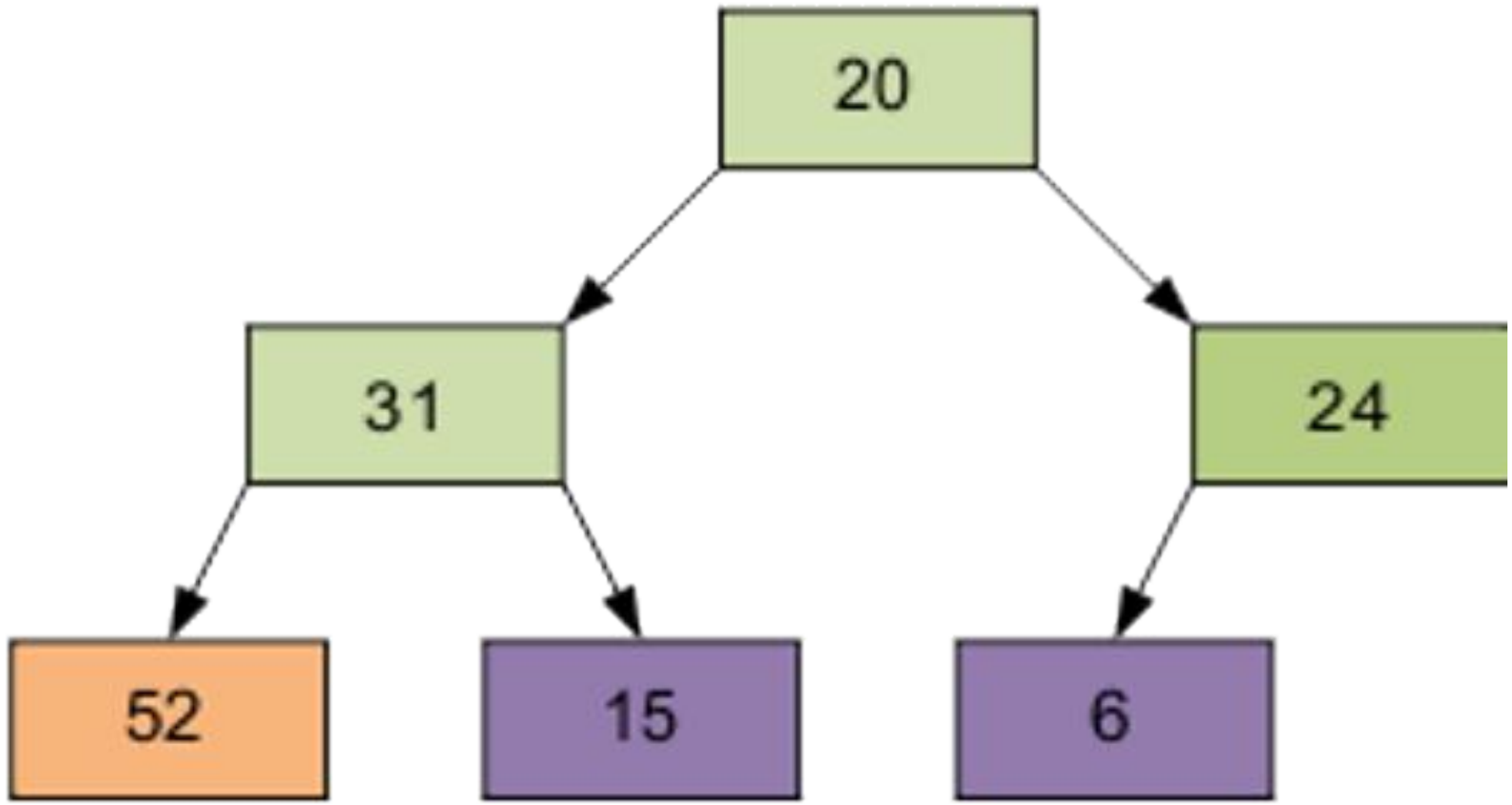
Сортировка на построенной пирамиде. Берем последний элемент массива в качестве текущего. Меняем верхний (наименьший) элемент массива и текущий местами. Текущий элемент (он теперь верхний) просеиваем сквозь n-1 элементную пирамиду. Затем берем предпоследний элемент и т.д.

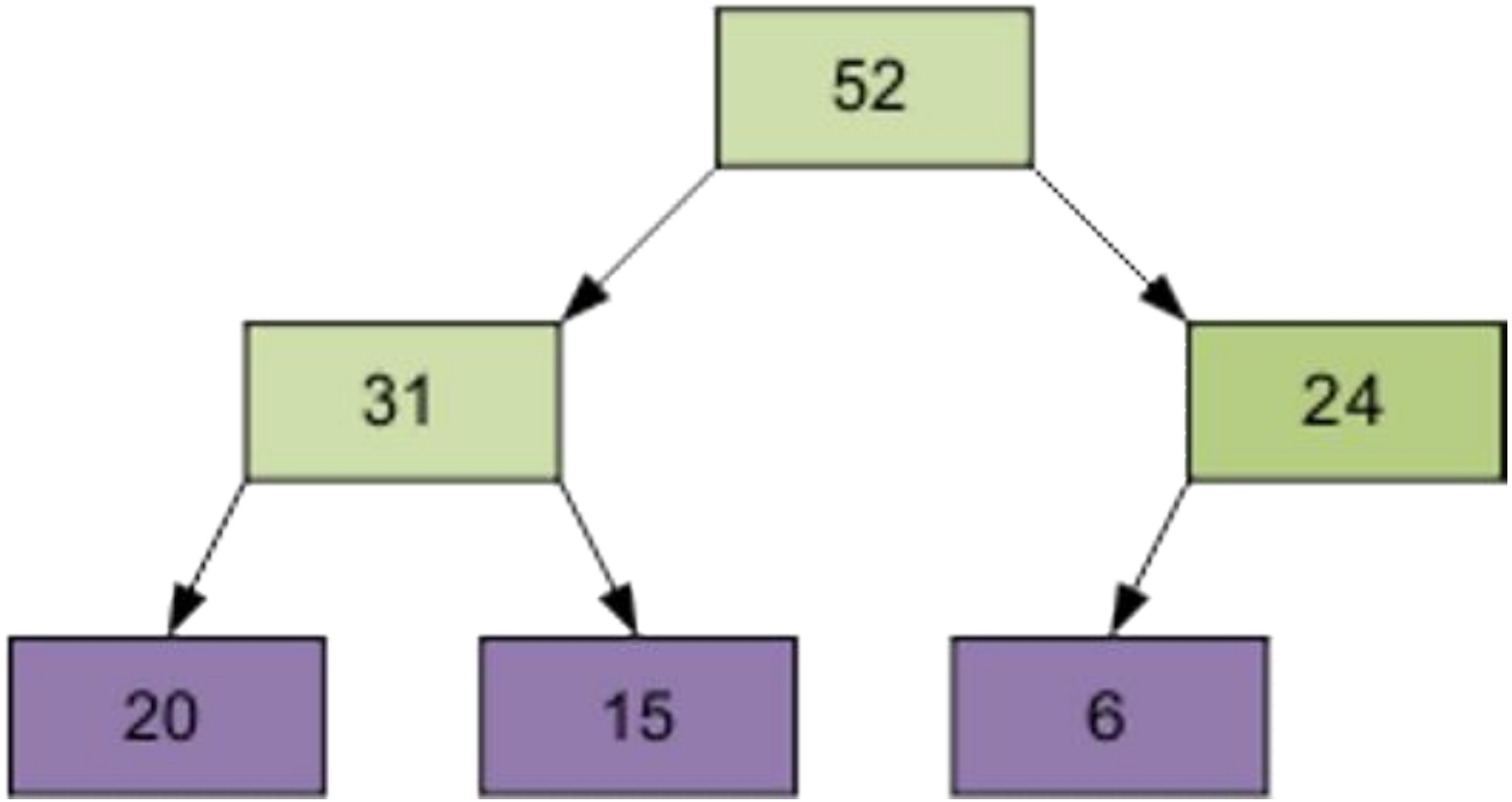

24
20
15
31
52
6

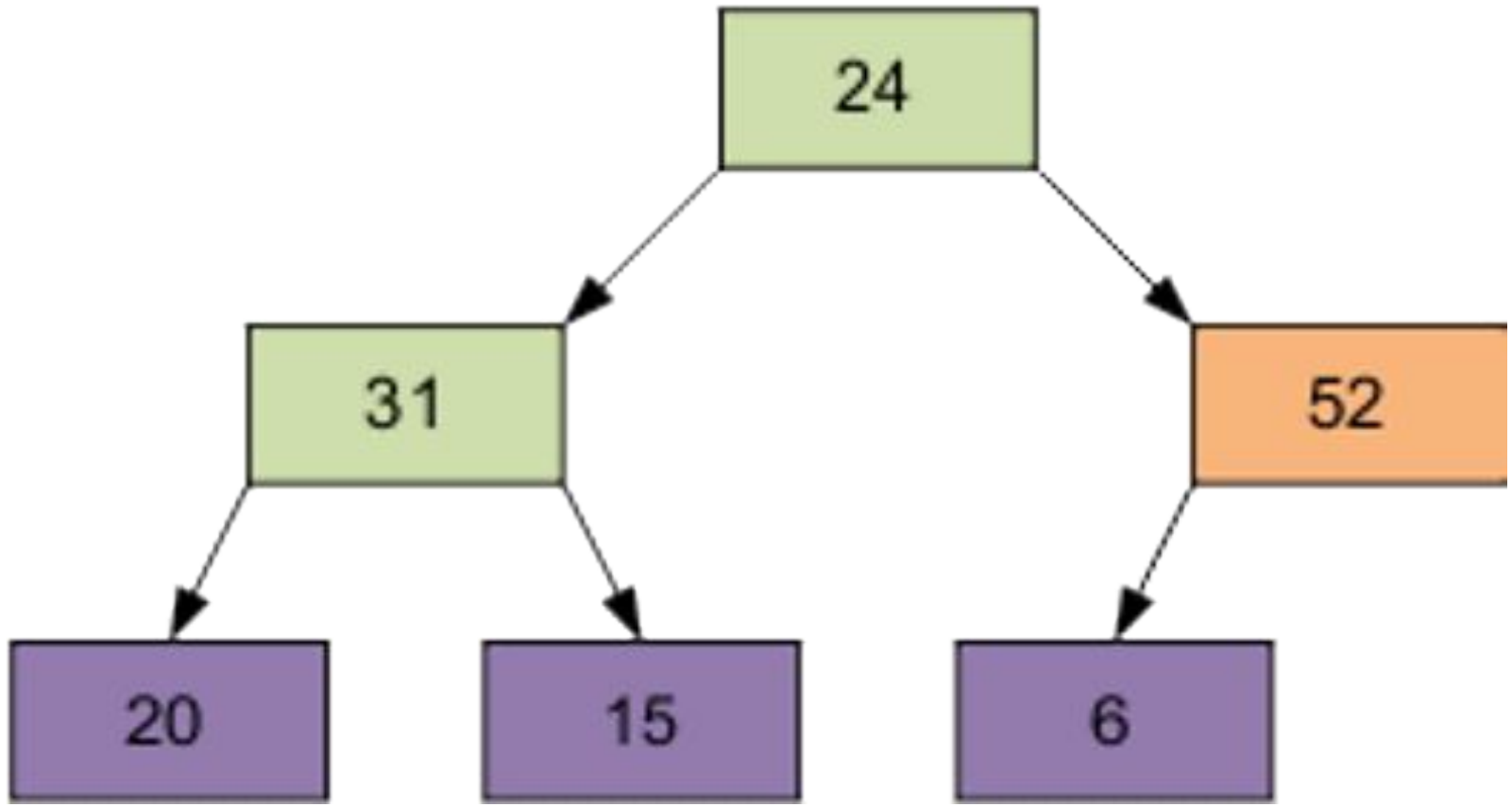


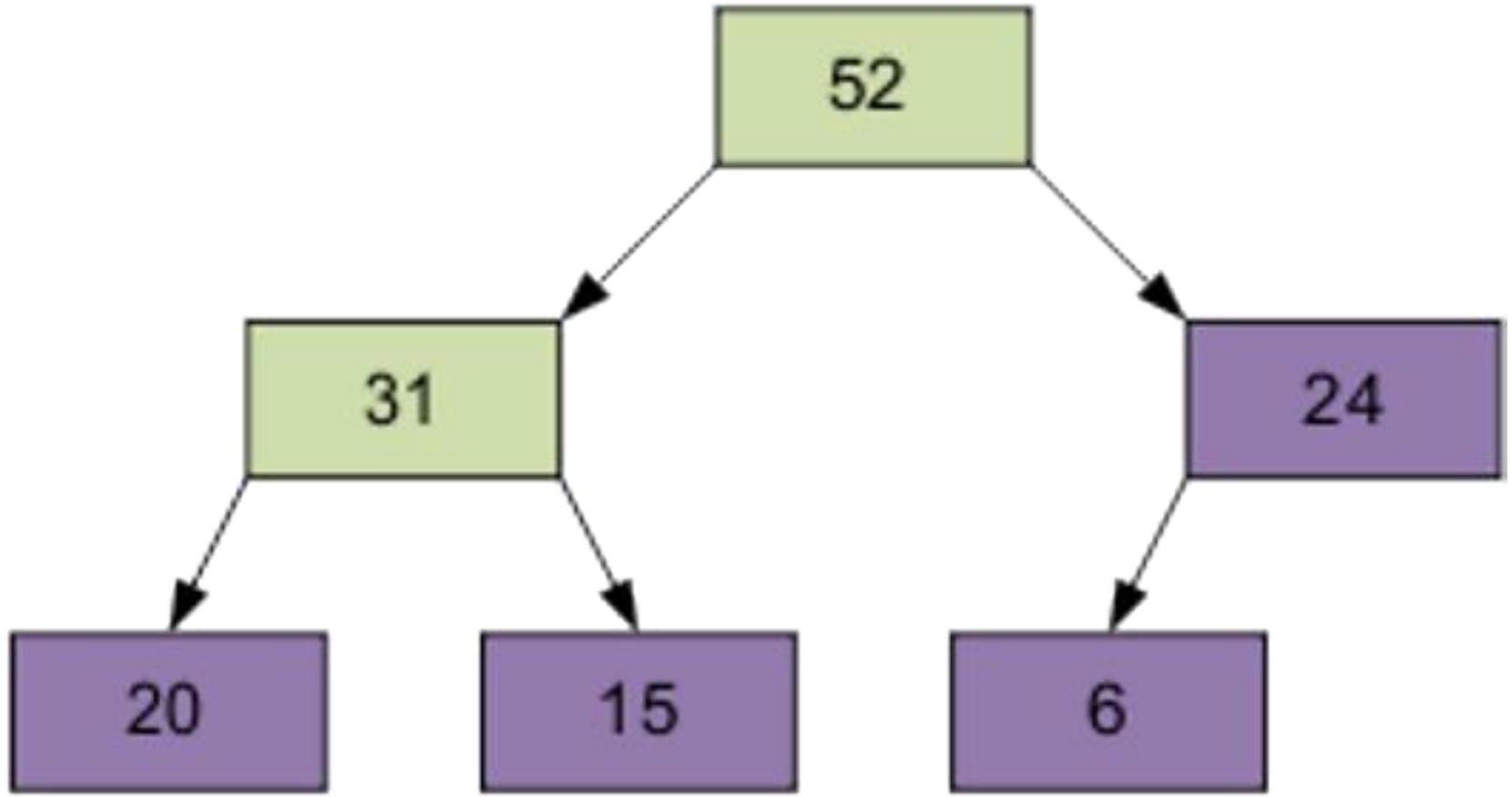
Продолжим процесс. В итоге массив будет отсортирован по убыванию.

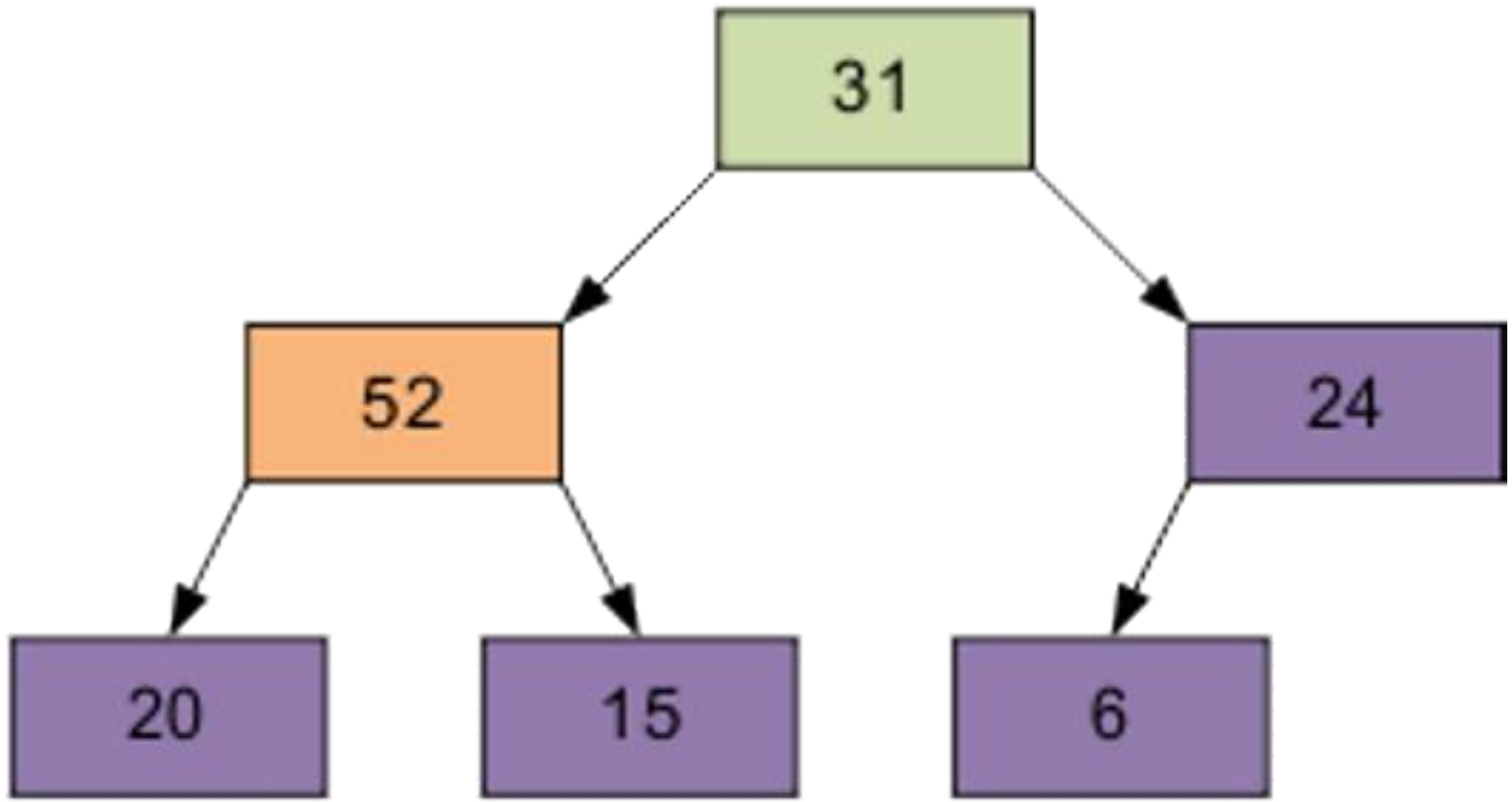

52
20
24
31
15
6

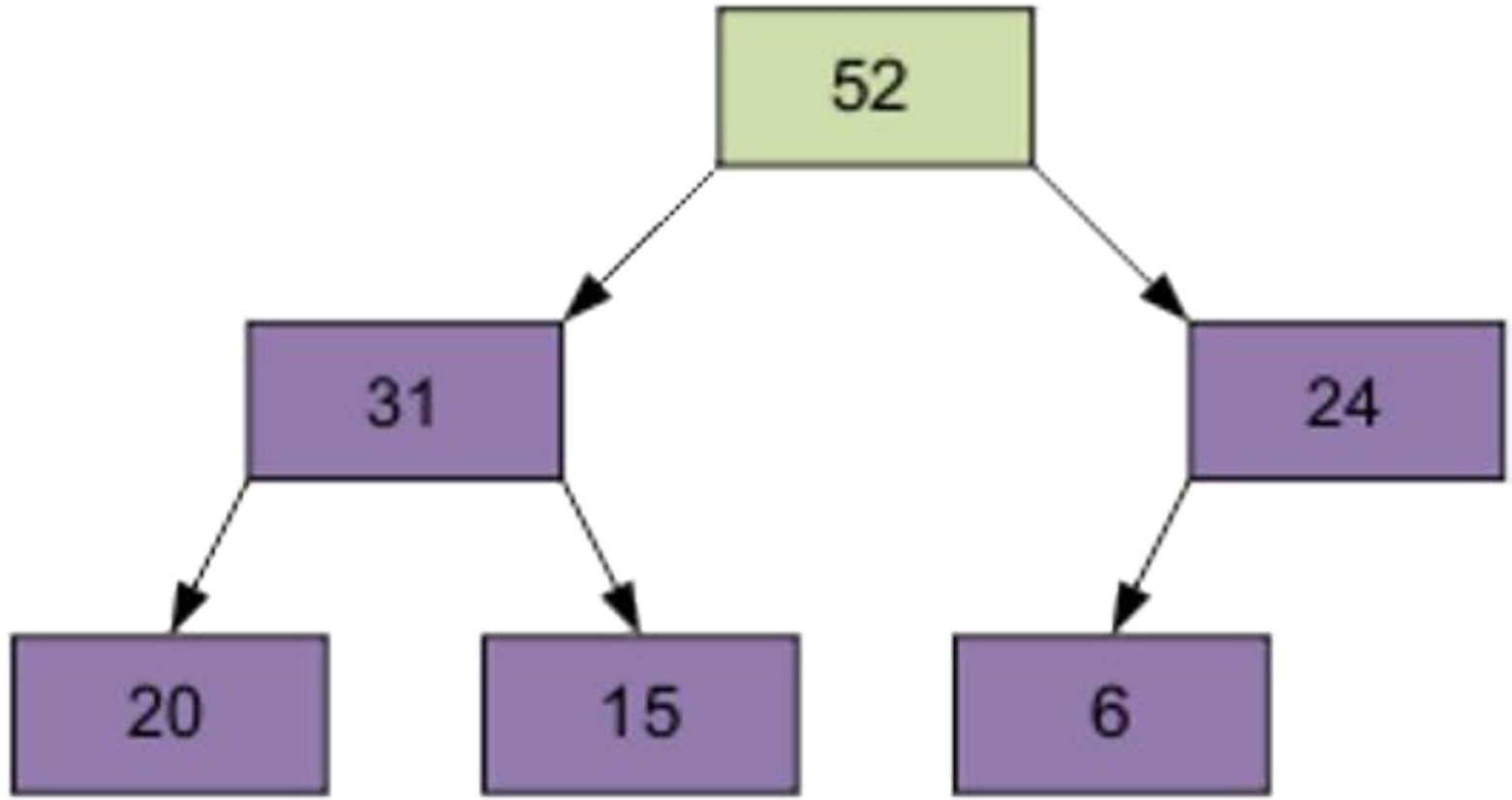

20
31
24
52
15
6

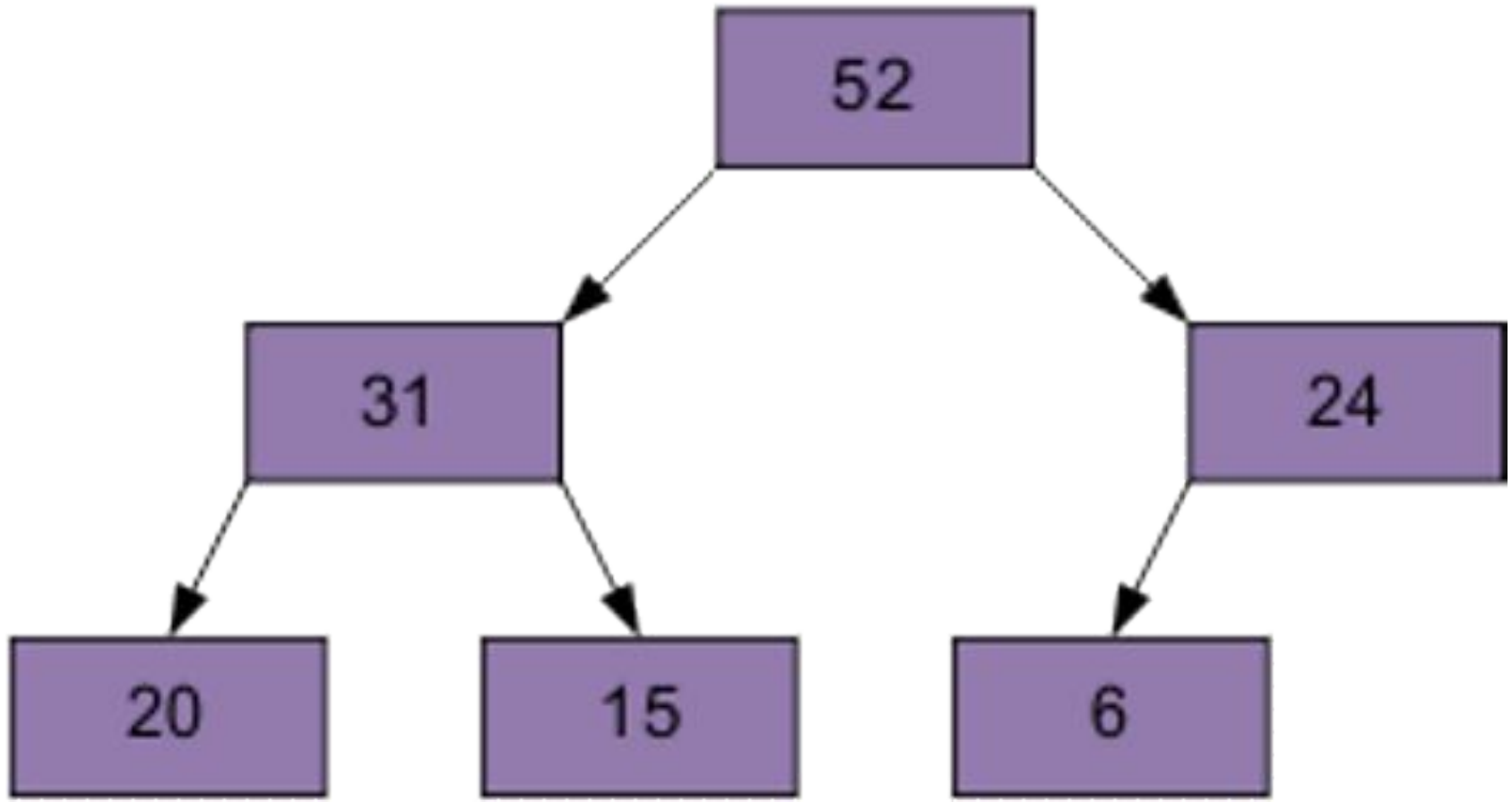

52
31
24
20
15
6


24
31
52
20
15
6


52
31
24
20
15
6


31
52
24
20
15
6


52
31
24
20
15
6

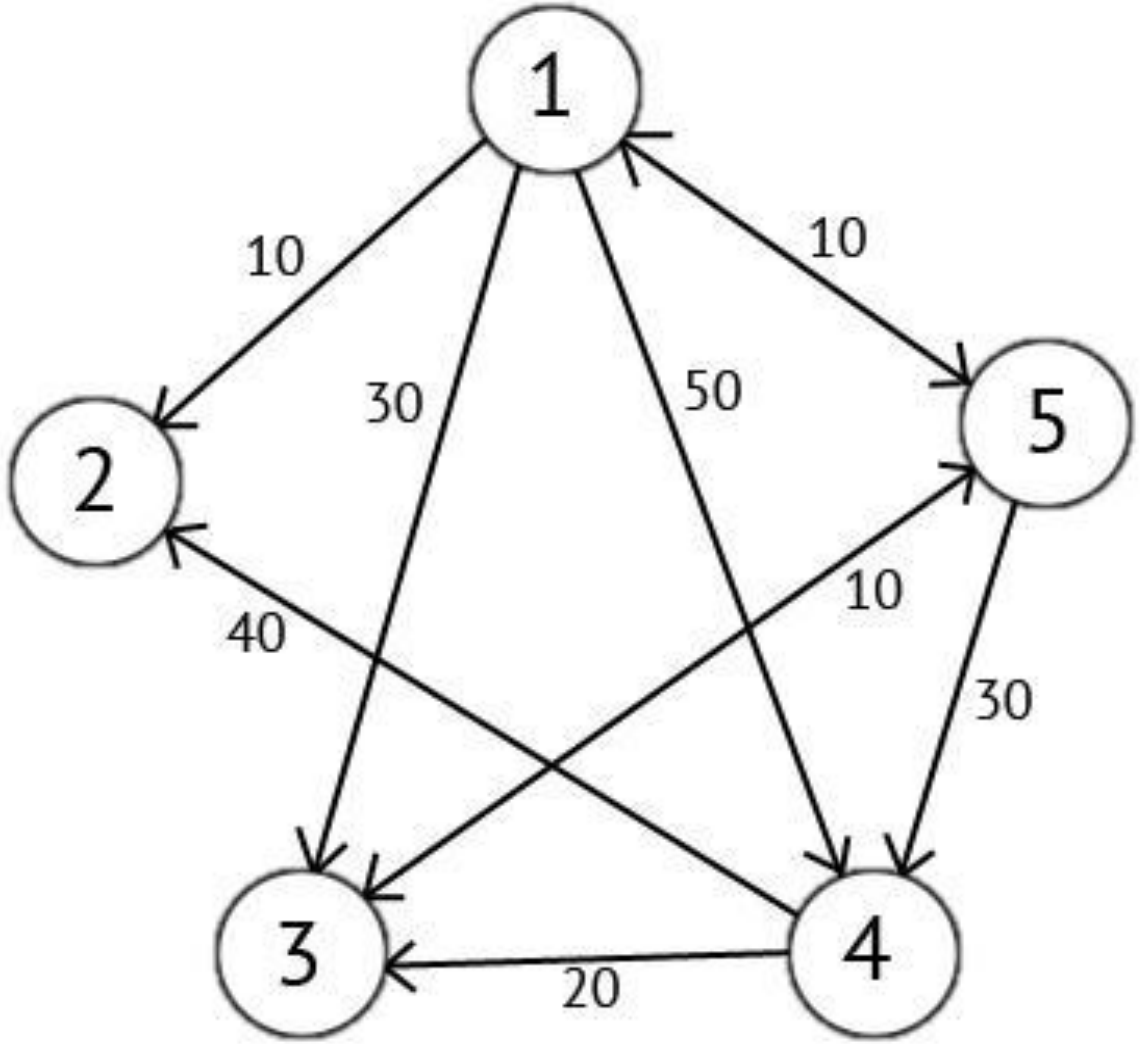

52
31
24
20
15
6

Алгоритмы поиска кратчайших путей на графах

Алгоритм Дейкстры

Алгоритм на графах, изобретённый нидерландским учёным Эдсгером Дейкстрой в 1959 году. Находит кратчайшие пути от одной из вершин графа до всех остальных. Алгоритм работает только для графов без рёбер отрицательного веса.

Для примера возьмем такой ориентированный граф G:

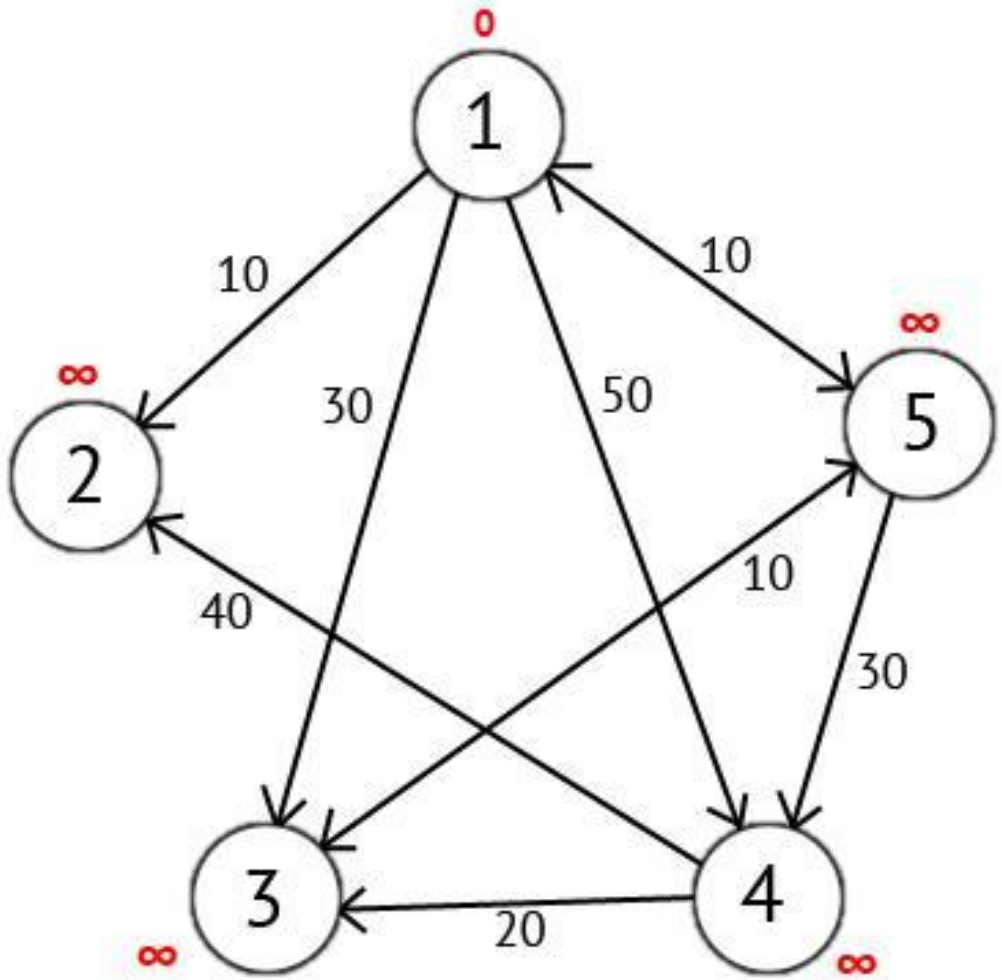


Этот граф мы можем представить в виде матрицы C

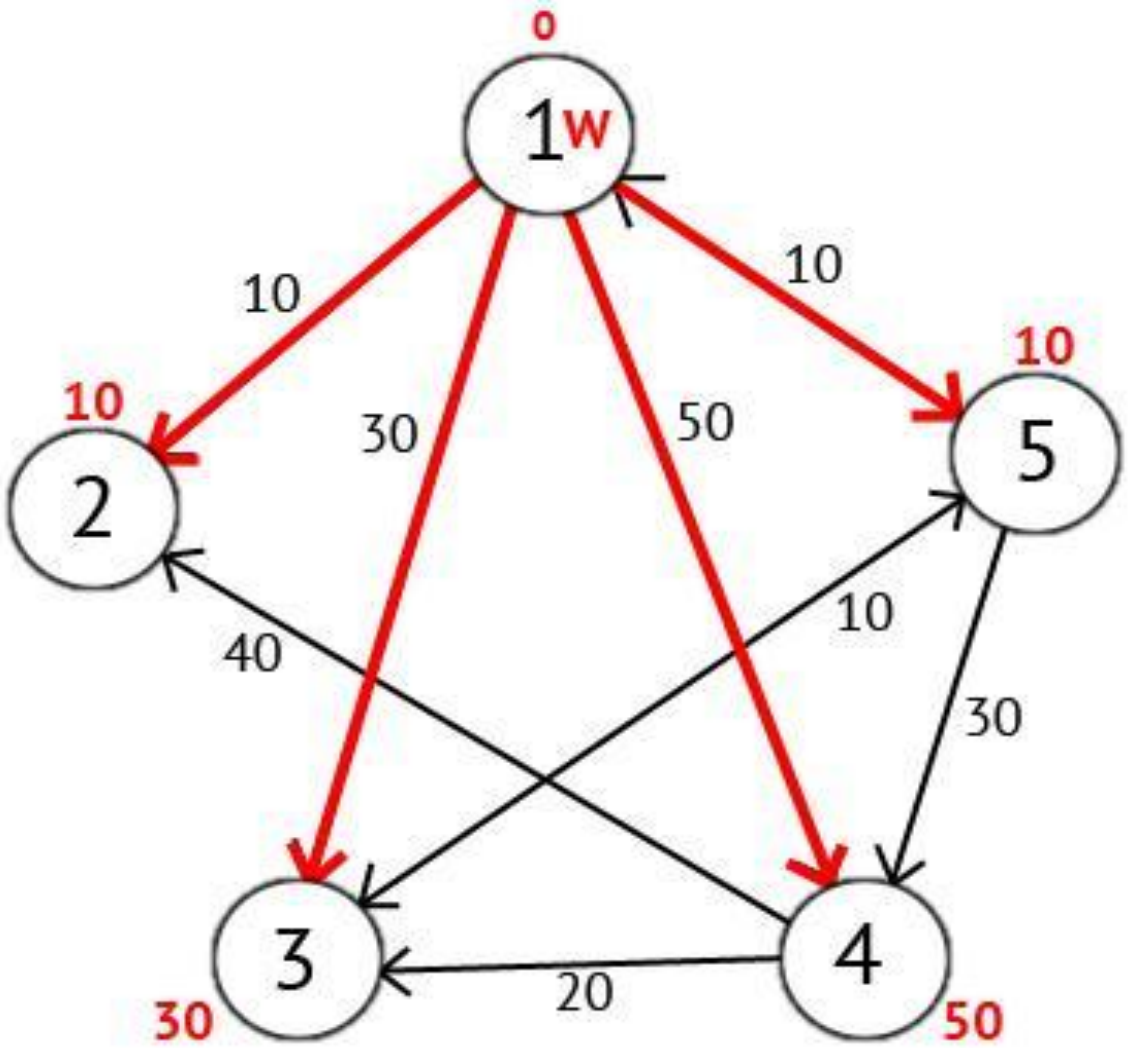
	1	2	3	4	5
1		10	30	50	10
2					
3					10
4		40	20		
5	10		10	30	

Возьмем в качестве источника вершину 1. Это значит что мы будем искать кратчайшие маршруты из вершины 1 в вершины 2, 3, 4 и 5.
Данный алгоритм пошагово перебирает все вершины графа и назначает им метки, которые являются известным минимальным расстоянием от вершины источника до конкретной вершины. Рассмотрим этот алгоритм на примере.

Присвоим 1-й вершине метку равную 0, потому как эта вершина — источник. Остальным вершинам присвоим метки равные бесконечности.

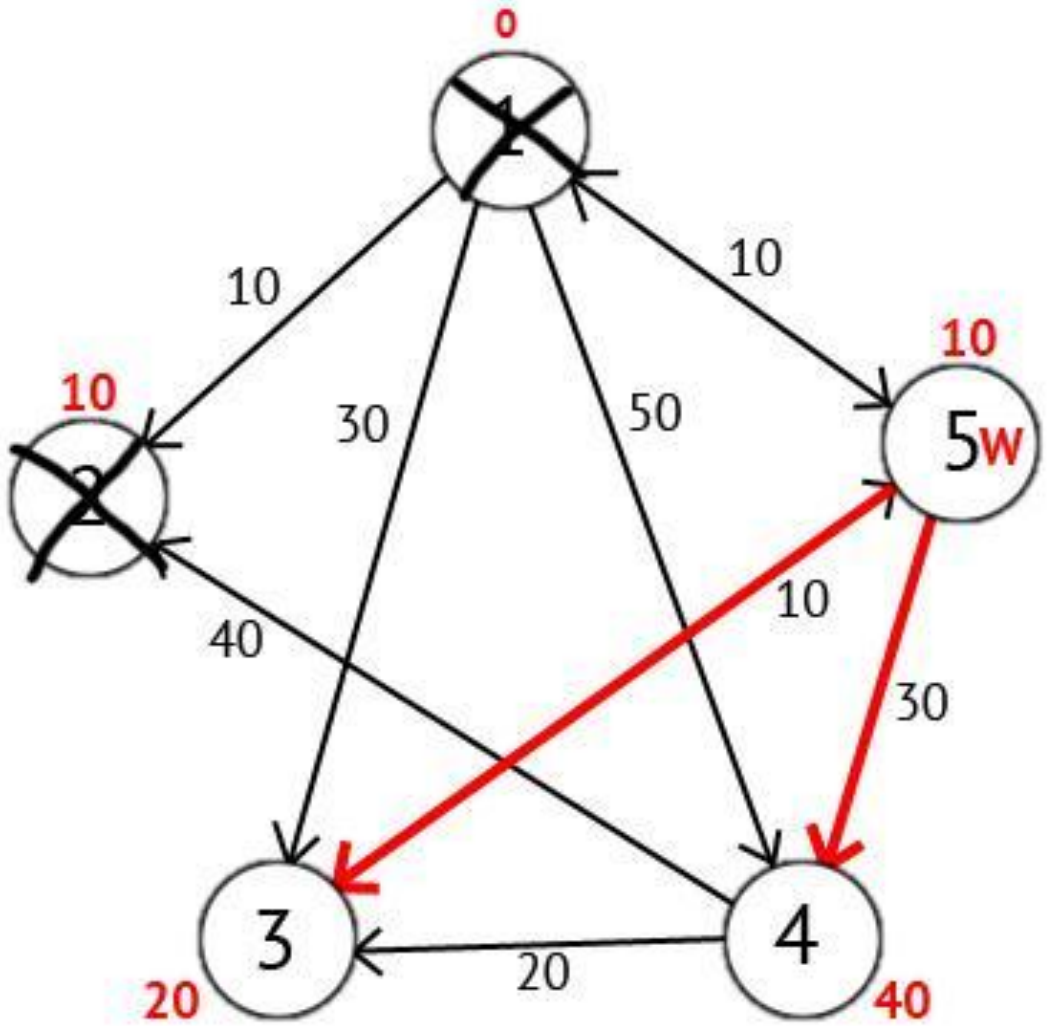


Далее выберем такую вершину W, которая имеет минимальную метку (сейчас это вершина 1) и рассмотрим все вершины в которые из вершины W есть путь, не содержащий вершин посредников. Каждой из рассмотренных вершин назначим метку равную сумме метки W и длинны пути из W в рассматриваемую вершину, но только в том случае, если полученная сумма будет меньше предыдущего значения метки. Если же сумма не будет меньше, то оставляем предыдущую метку без изменений.

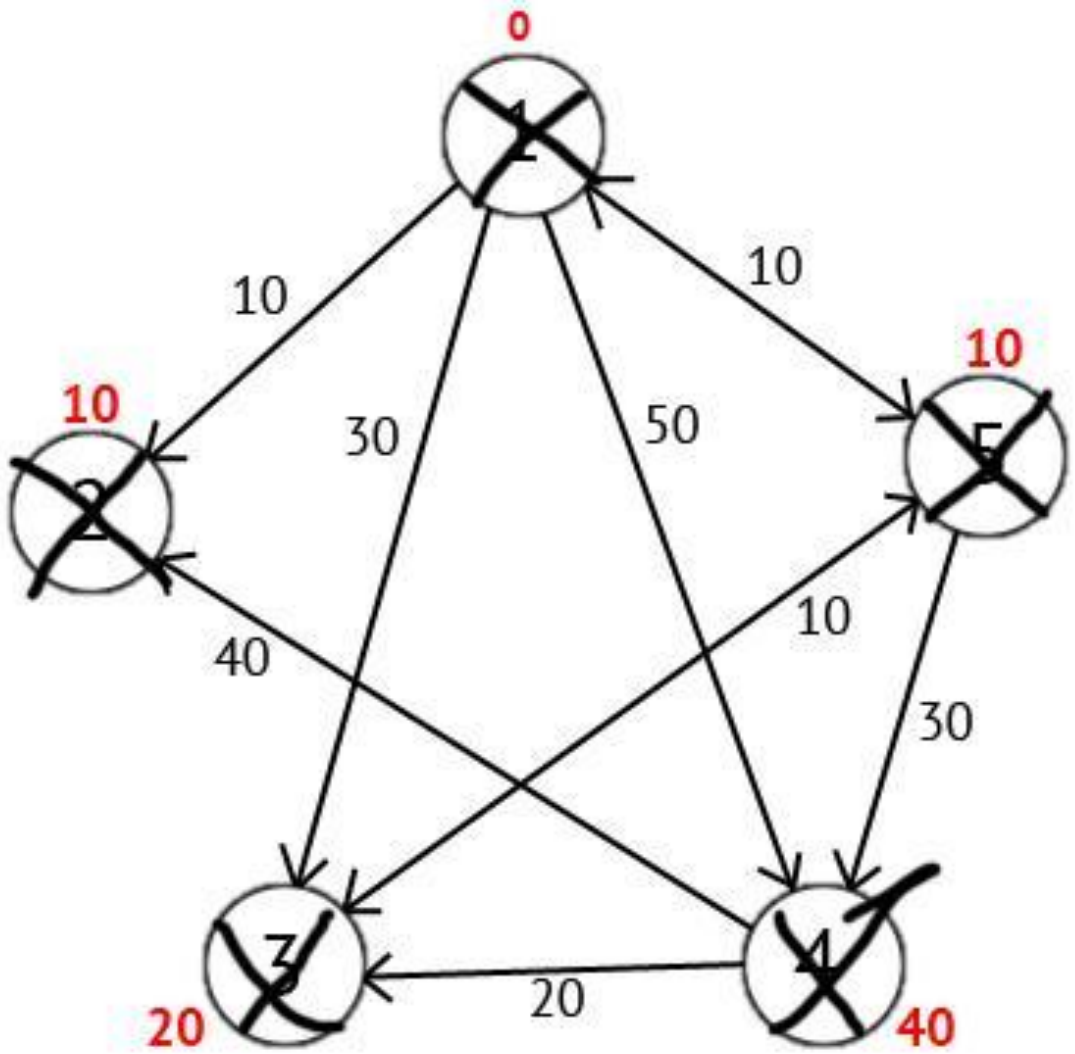


После того как мы рассмотрели все вершины, в которые есть прямой путь из W, вершину W мы отмечаем как посещённую, и выбираем из ещё не посещенных такую, которая имеет минимальное значение метки, она и будет следующей вершиной W. В данном случае это вершина 2 или 5. Если есть несколько вершин с одинаковыми метками, то не имеет значения какую из них мы выберем как W.

Мы выберем вершину 2. Но из нее нет ни одного исходящего пути, поэтому мы сразу отмечаем эту вершину как посещенную и переходим к следующей вершине с минимальной меткой. На этот раз только вершина 5 имеет минимальную метку. Рассмотрим все вершины в которые есть прямые пути из 5, но которые ещё не помечены как посещенные. Снова находим сумму метки вершины W и веса ребра из W в текущую вершину, и если эта сумма будет меньше предыдущей метки, то заменяем



Исходя из картинки мы можем увидеть, что метки 3-ей и 4-ой вершин стали меньше, тоесть был найден более короткий маршрут в эти вершины из вершины источника. Далее отмечаем 5-ю вершину как посещенную и выбираем следующую вершину, которая имеет минимальную метку. Повторяем все перечисленные выше действия до тех пор, пока есть непосещенные вершины.



Выполнив все действия получим такой результат

Также есть вектор P, исходя из которого можно построить кратчайшие маршруты. По количеству элементов этот вектор равен количеству вершин в графе, Каждый элемент содержит последнюю промежуточную вершину на кратчайшем пути между вершиной-источником и конечной вершиной. В начале алгоритма все элементы вектора P равны вершине источнику (в нашем случае P = {1, 1, 1, 1, 1}).

Далее на этапе пересчета значения метки для рассматриваемой вершины, в случае если метка рассматриваемой вершины меняется на меньшую, в массив P мы записываем значение текущей вершины W. Например: у 3-ей вершины была метка со значением «30», при W=1. Далее при W=5, метка 3-ей вершины изменилась на «20», следовательно мы запишем значение в вектор P — P[3]=5. Также при W=5 изменилось значение метки у 4-й вершины (было «50», стало «40»), значит нужно присвоить 4-му элементу вектора P значение W — P[4]=5. В результате получим вектор P = {1, 1, 5, 5, 1}.

Зная что в каждом элементе вектора P записана последняя промежуточная вершина на пути между источником и конечной вершиной, мы можем получить и сам кратчайший маршрут.

Алгоритмы поиска кратчайших путей на графах

Алгоритм Флойда

Алгоритм нахождения длин кратчайших путей между всеми парами вершин во взвешенном ориентированном графе. Работает корректно, если в графе нет циклов отрицательной величины, а в случае, когда такой цикл есть, позволяет найти хотя бы один такой цикл.

Алгоритм

- Пусть вершины графа $\mathrm{G} = (v, E), |v| = n$ пронумерованы от 1 до n и введено обозначение $d_{\mathrm{i}j}^{k} (\mathrm{i} < j < k)$ для длины кратчайшего пути от I до j проходит только через вершины 1…k. Очевидно, что $d_{\mathrm{i}j}^{0}$ длина (вес) ребра (I,j), если таковое существует (в противном случае его длина может быть обозначена как ∞).

Существует два варианта значения $d_{ij}^{k}, k \in (1, \dots, n)$:

1.Кратчайший путь между I,j не проходит через вершину k,

тогда $d_{ij}^{k}=d_{ij}^{k-1}$

2. Существует более короткий путь между I,j проходящий через k,

тогда он сначала идет от I до k, а потом от k до j В этом случае,

очевидно $d_{ij}^{k}=d_{ik}^{k-1}+d_{kj}^{k-1}$

Таким образом, для нахождения значения функции достаточно выбрать минимум из двух обозначенных значений.

Тогда рекуррентная формула для d_{ij}^{k} имеет вид:

d_{ij}^{0} - длина ребра (I,j);

$d_{ij}^{k} = \min(d_{ij}^{k-1}, d_{ik}^{k-1}+d_{kj}^{k-1})$.

Алгоритм Флойда-Уоршелла последовательно вычисляет все значения d_{ij}^{k}, $\forall i, j$ для k от 1 до n. Полученные значения d_{ij}^{n} являются длинами кратчайших путей между вершинами i, j .

Псевдокод

- На каждом шаге алгоритм генерирует матрицу W, $w_{ij} = d_{ij}^{n}$. Матрица W содержит длины кратчайших путей между всеми вершинами графа. Перед работой алгоритма матрица W заполняется длинами ребер графа (или запредельно большим *M*, если ребра нет).

```
for k = 1 to n
    for i = 1 to n
        for j = 1 to n
            W[i][j] = min(W[i][j] , W[i][k]  + W[k][j])
```

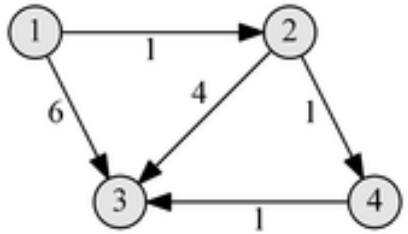
Сложность алгоритма

- Три вложенных цикла содержат операцию, исполняемую за константное время $\sum_{n,n,n} \mathrm{O}(1) = O(n^3)$, то есть алгоритм имеет кубическую сложность, при этом простым расширением можно получить также информацию о кратчайших путях — помимо расстояния между двумя узлами записывать в матрицу идентификатор первого узла в пути.

Но существует решение и за $\sum_{n,n,n} \mathrm{O}(1) = O(n^2)$ где хранятся значения не для всех вершин, а только значения для предыдущей вершины, так как следующая получается рекурсивно.

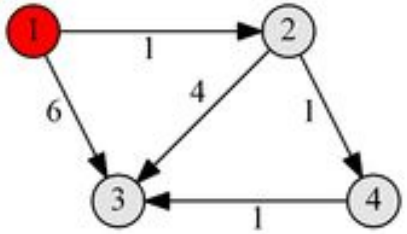
Пример работы

i=0



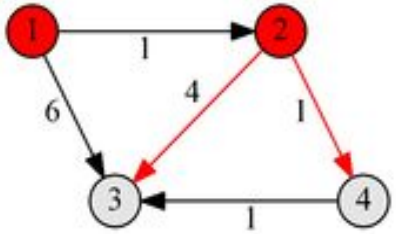
$$\begin{pmatrix} \times & 1 & 6 & \infty \\ \infty & \times & 4 & 1 \\ \infty & \infty & \times & \infty \\ \infty & \infty & 1 & \times \end{pmatrix}$$

i=1



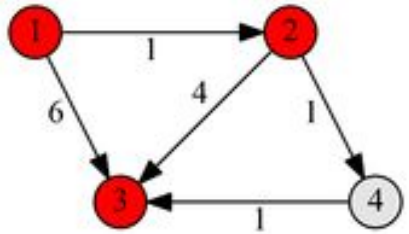
$$\begin{pmatrix} \times & 1 & 6 & \infty \\ \infty & \times & 4 & 1 \\ \infty & \infty & \times & \infty \\ \infty & \infty & 1 & \times \end{pmatrix}$$

i=2



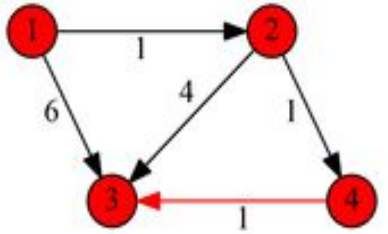
$$\begin{pmatrix} \times & 1 & \mathbf{5} & \mathbf{2} \\ \infty & \times & 4 & 1 \\ \infty & \infty & \times & \infty \\ \infty & \infty & 1 & \times \end{pmatrix}$$

i=3



$$\begin{pmatrix} \times & 1 & 5 & 2 \\ \infty & \times & 4 & 1 \\ \infty & \infty & \times & \infty \\ \infty & \infty & 1 & \times \end{pmatrix}$$

i=4



$$\begin{pmatrix} \times & 1 & \mathbf{3} & 2 \\ \infty & \times & \mathbf{2} & 1 \\ \infty & \infty & \times & \infty \\ \infty & \infty & 1 & \times \end{pmatrix}$$

Алгоритм Краскала

Алгоритм Краскала - алгоритм поиска минимального остовного дерева (англ. *minimum spanning tree*, *MST*) во взвешенном неориентированном связном графе.

Идея

- Будем последовательно строить подграф F графа G ("растущий лес"), пытаясь на каждом шаге достроить F до некоторого MST. Включим в F все вершины графа G. Теперь будем обходить множество E (G) в порядке неубывания весов ребер. Если очередное ребро ϵ соединяет вершины одной компоненты связности F , то добавление его в остов приведет к возникновению цикла в этой компоненте связности. В таком случае, очевидно, ϵ не может быть включено в F. Иначе ϵ соединяет разные компоненты связности F, тогда существует (S,T) разрез такой, что одна из компонент связности составляет одну его часть, а оставшаяся часть графа — вторую. Тогда ϵ — минимальное ребро, пересекающее этот разрез. Значит, из леммы о безопасном ребре следует, что ϵ является безопасным, поэтому добавим это ребро в F. На последнем шаге ребро соединит две оставшиеся компоненты связности, полученный подграф будет минимальным островным деревом графа G.

Реализация

//G – исходный граф

//F – минимальный остров

Function kruskalFindMST():

$F \leftarrow V(G)$

sort(E(G))

for $vu \in E(G)$

if v и u в разных компонентах связанности F

$F = F \cup vu$

return F

Задача о максимальном ребре минимального веса

Легко показать, что максимальное ребро в MST минимально. Обратное в общем случае неверно. Но MST из-за сортировки строится за $O(E \log E)$. Однако из-за того, что необходимо минимизировать только максимальное ребро, а не сумму всех рёбер, можно предъявить алгоритм, решающий задачу за линейное время.

С помощью алгоритма поиска k-ой порядковой статистики найдем ребро-медиану за O(E) и разделим множество ребер на два равных по мощности так, чтобы ребра в первом не превосходили по весу ребер во втором. Проверим образуют ли ребра из первого подмножества остов графа, запустив обход в глубину.

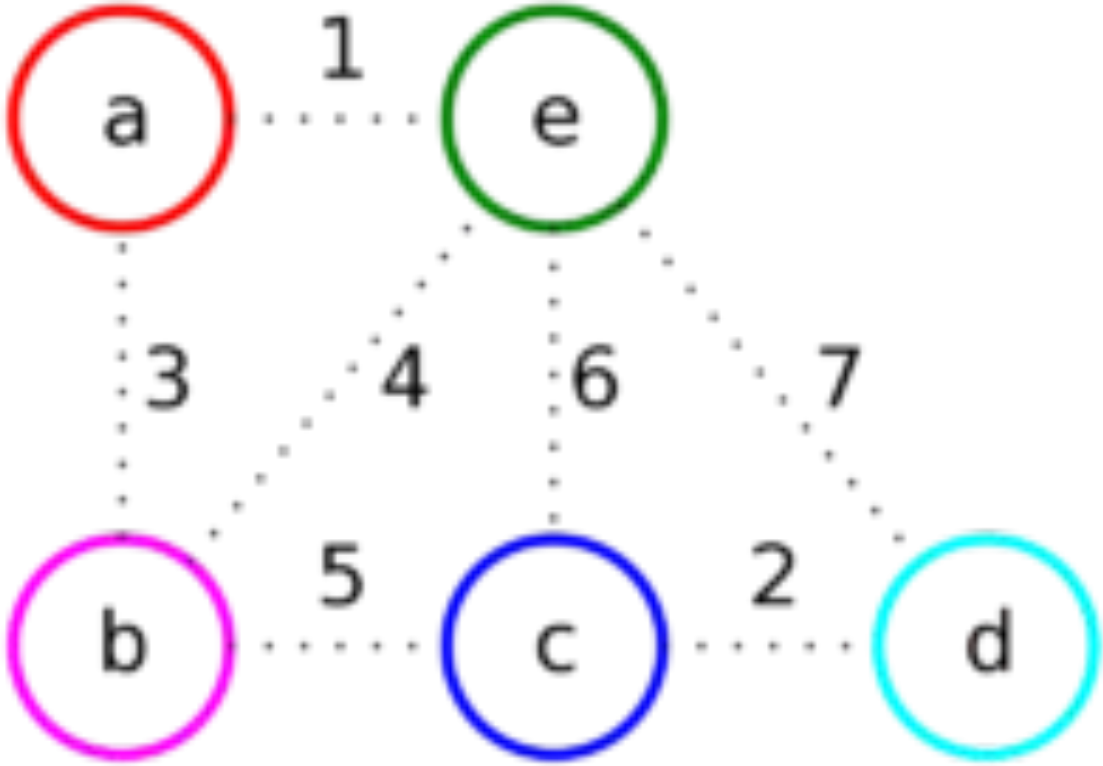
- Если да, то рекурсивно запустим алгоритм от него.
- В противном случае сконденсируем получившиеся несвязные омпоненты в супервершины и рассмотрим граф с этими вершинами и ребрами из второго подмножества.

На последнем шаге останутся две компоненты связности и одно ребро в первом подмножестве — это максимальное ребро минимального веса.

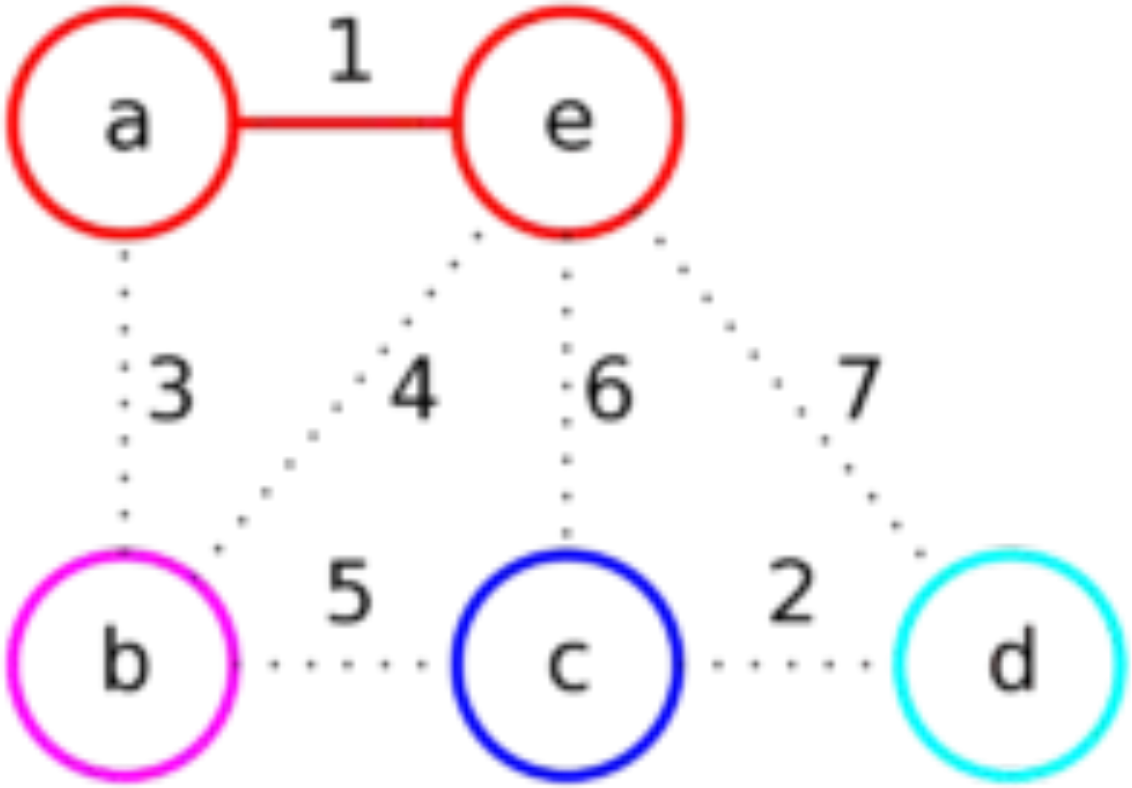
На каждом шаге ребер становится в два раза меньше, а все операции выполняются за время пропорциональное количеству ребер на текущем шаге, тогда время работы алгоритма $O\left(E + \frac{E}{2} + \frac{E}{4} + \cdots + 1\right) = O(E)$.

Пример

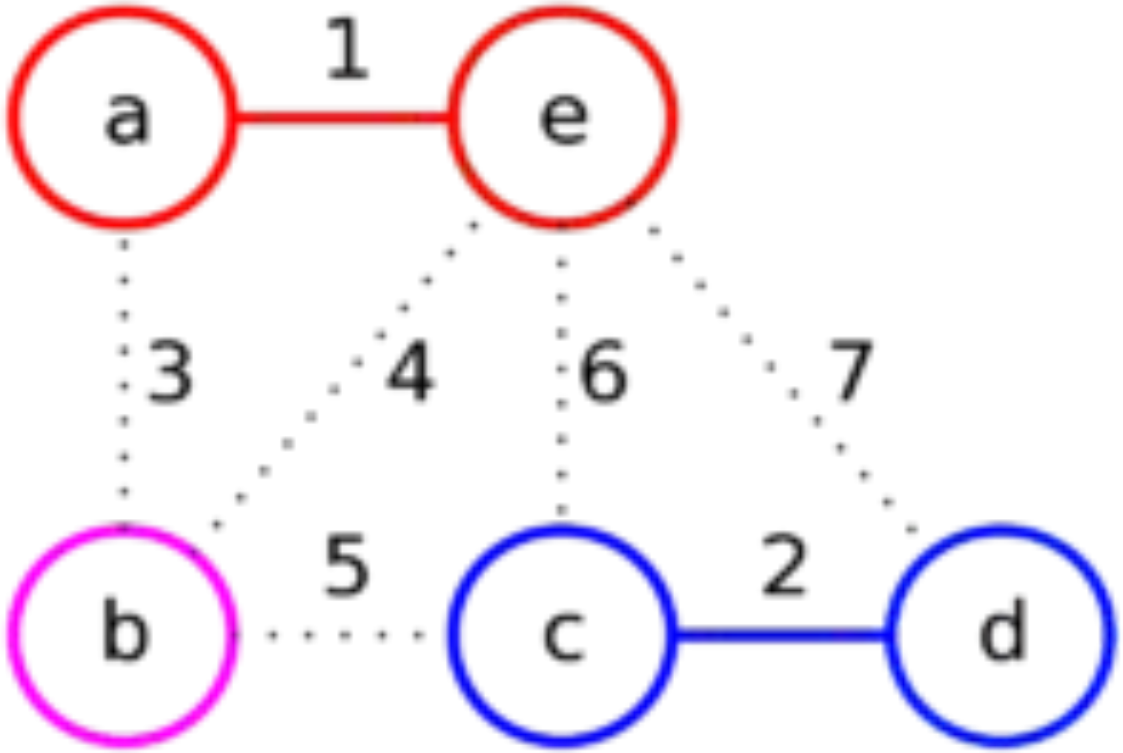
Рёбра *(в порядке их просмотра)*	ae	cd	ab	be	bc	ec	ed
Веса рёбер	1	2	3	4	5	6	7



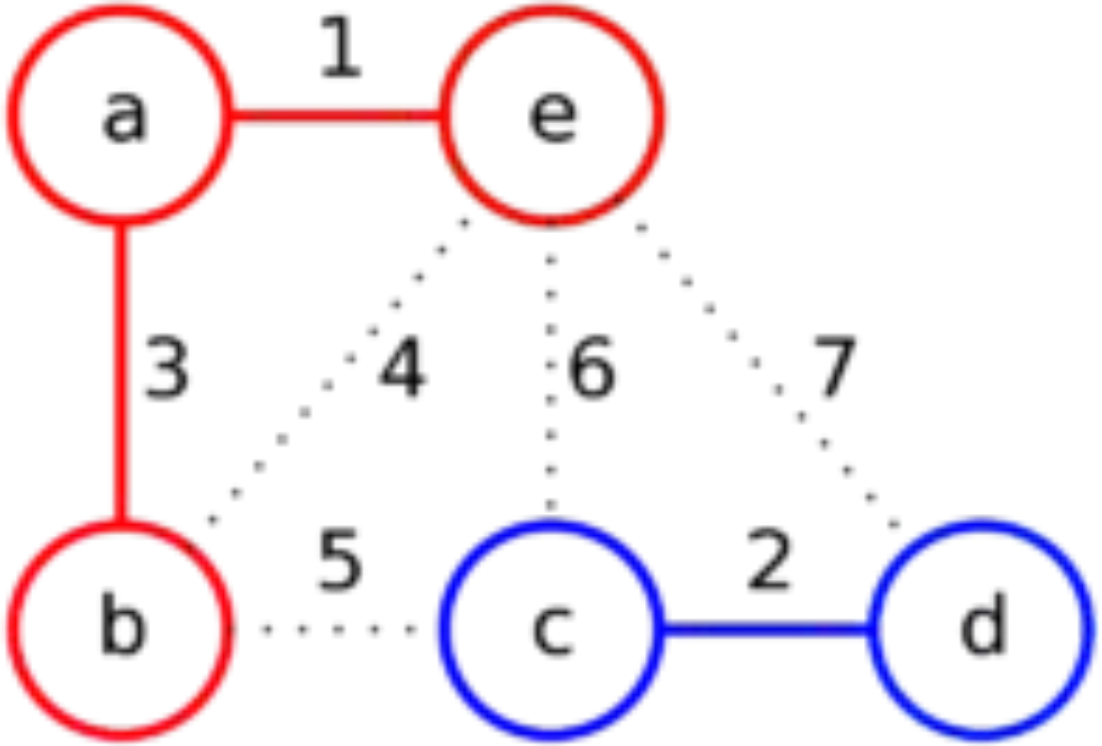
Первое ребро, которое будет рассмотрено — **ae**, так как его вес минимальный.
Добавим его к ответу, так как его концы соединяют вершины из разных множеств (**a**— красное и **e** — зелёное).
Объединим красное и зелёное множество в одно (красное), так как теперь они соединены ребром.



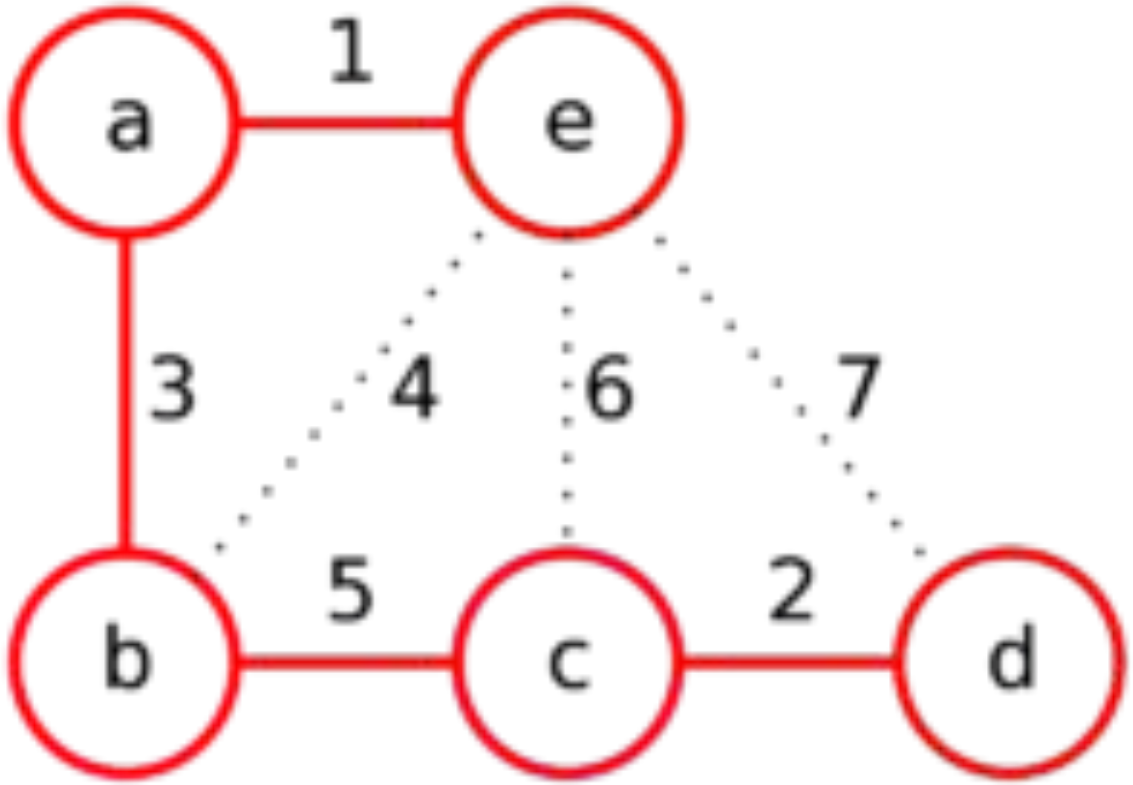
Рассмотрим следующие ребро — **cd**.
Добавим его к ответу, так как его концы соединяют вершины из разных множеств (**c**— синее и **d** — голубое).
Объединим синее и голубое множество в одно (синее), так как теперь они соединены ребром.



Дальше рассмотрим ребро **ab**.
Добавим его к ответу, так как его концы соединяют вершины из разных множеств (**a**— красное и **b** — розовое).
Объединим красное и розовое множество в одно (красное), так как теперь они соединены ребром.



Рассмотрим следующие ребро — **be**.
Оно соединяет вершины из одного множества, поэтому перейдём к следующему ребру **bc**
Добавим его к ответу, так как его концы соединяют вершины из разных множеств (**b**— красное и **c** — синее).
Объединим красное и синее множество в одно (красное), так как теперь они соединены ребром.



Рёбра **ec** и **ed** соединяют вершины из одного множества, поэтому после их просмотра они не будут добавлены в ответ
Всё рёбра были рассмотрены, поэтому алгоритм завершает работу.
Полученный граф — минимальное остовное дерево

Parallel Programming in OpenMP standard

Total amount of intelligence on the planet is a constant, in spite of the constant population growth.

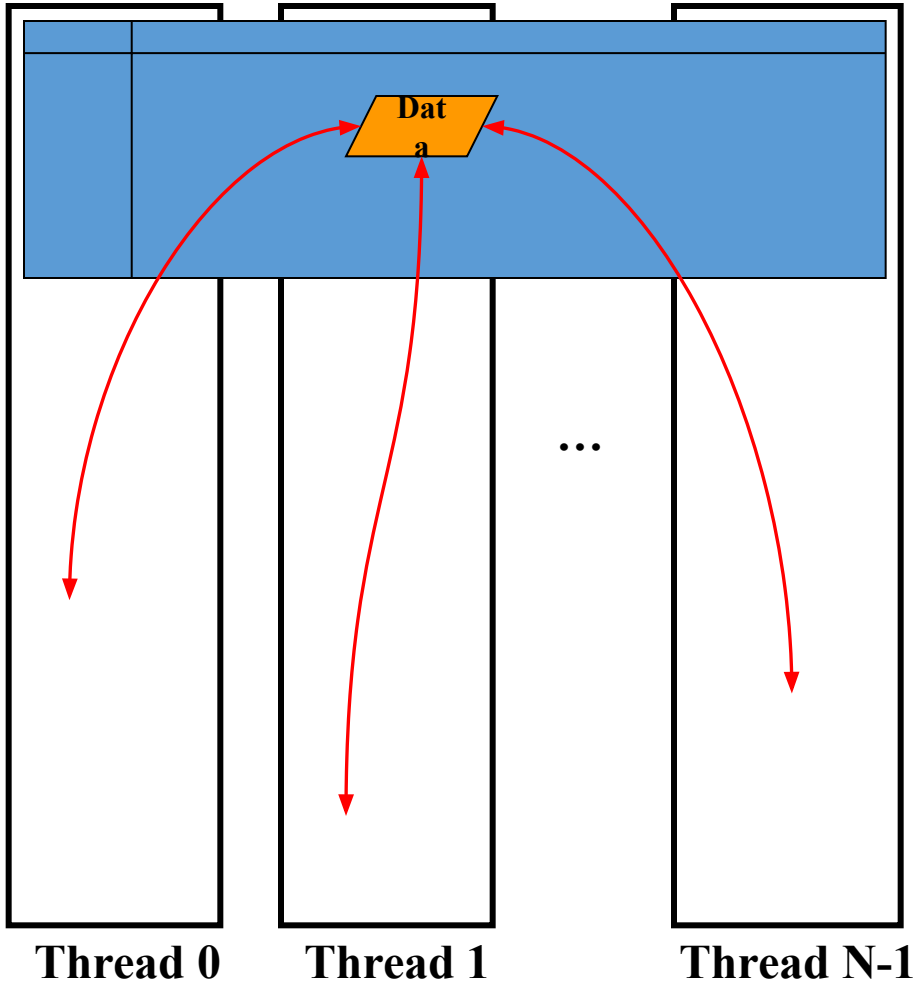
A. Bloch

"Parallel and distributed programming"

Content

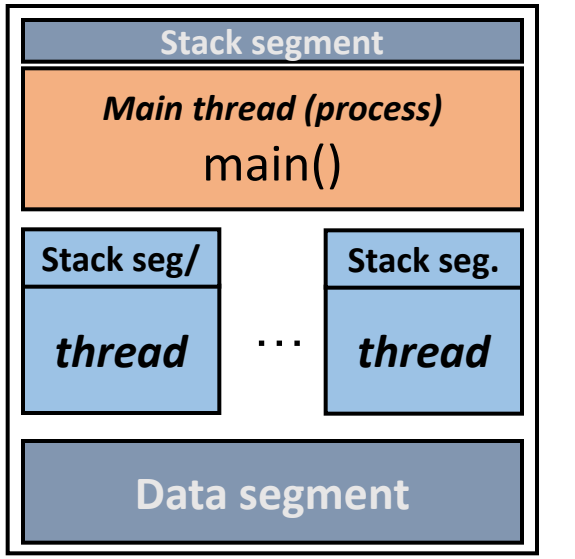
- Programming model in shared memory
- Model "pulsating" parallelism FORK-JOIN
- OpenMP standard
- Basic concepts and features OpenMP

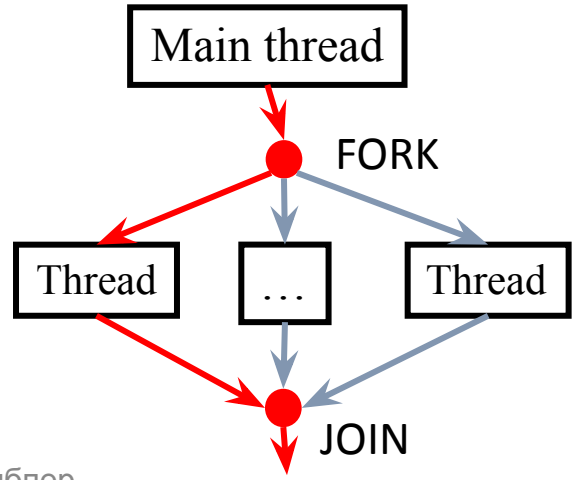
Programming shared memory



- *A parallel application* consists of multiple *threads* running simultaneously.
- The threads share a common memory.
- Exchanges between the filaments are made through the read/write shared memory.
- The threads running on different cores of one processor.

FORK-JOIN model





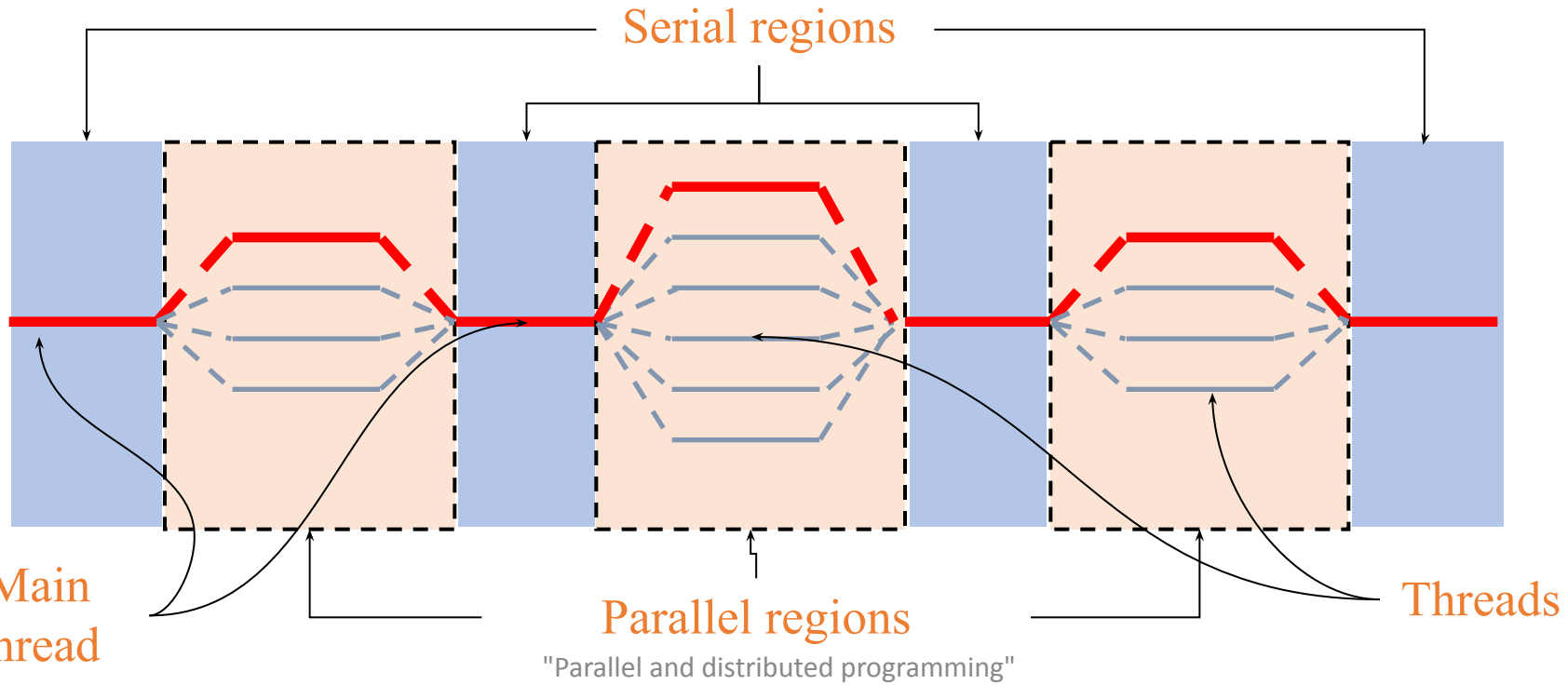
- Program - a full-weighted process.
- The process can run *lightweight processes (threads)* running in the background.
  - Application process - *the main thread*.
  - Thread can run other threads in the process. Each thread has its own stack segment.
  - All the threads of a process share the data segment of the process.

Standard OpenMP

- *OpenMP (Open Multi-Processing)* - a standard that provides a programming model in shared memory and Fork-Join.
- The standard includes a set of compiler directives and specifications routines in C, C + + and FORTRAN.
- Standard is implemented by developers of compilers for various hardware and software platforms (clusters, PCs, ..., Windows, Unix / Linux, ...).
- Standards developers - OpenMP Architecture Review Board(www.openmp.org).

OpenMP-program

- The main thread (the program) generates a family of child threads (as necessary). Duplication and Termination by using *compiler directives*.



Simple OpenMP-program

Serial code

```
void main()
{

printf("Hello!\n");
}
```

Result

```
Hello!
```

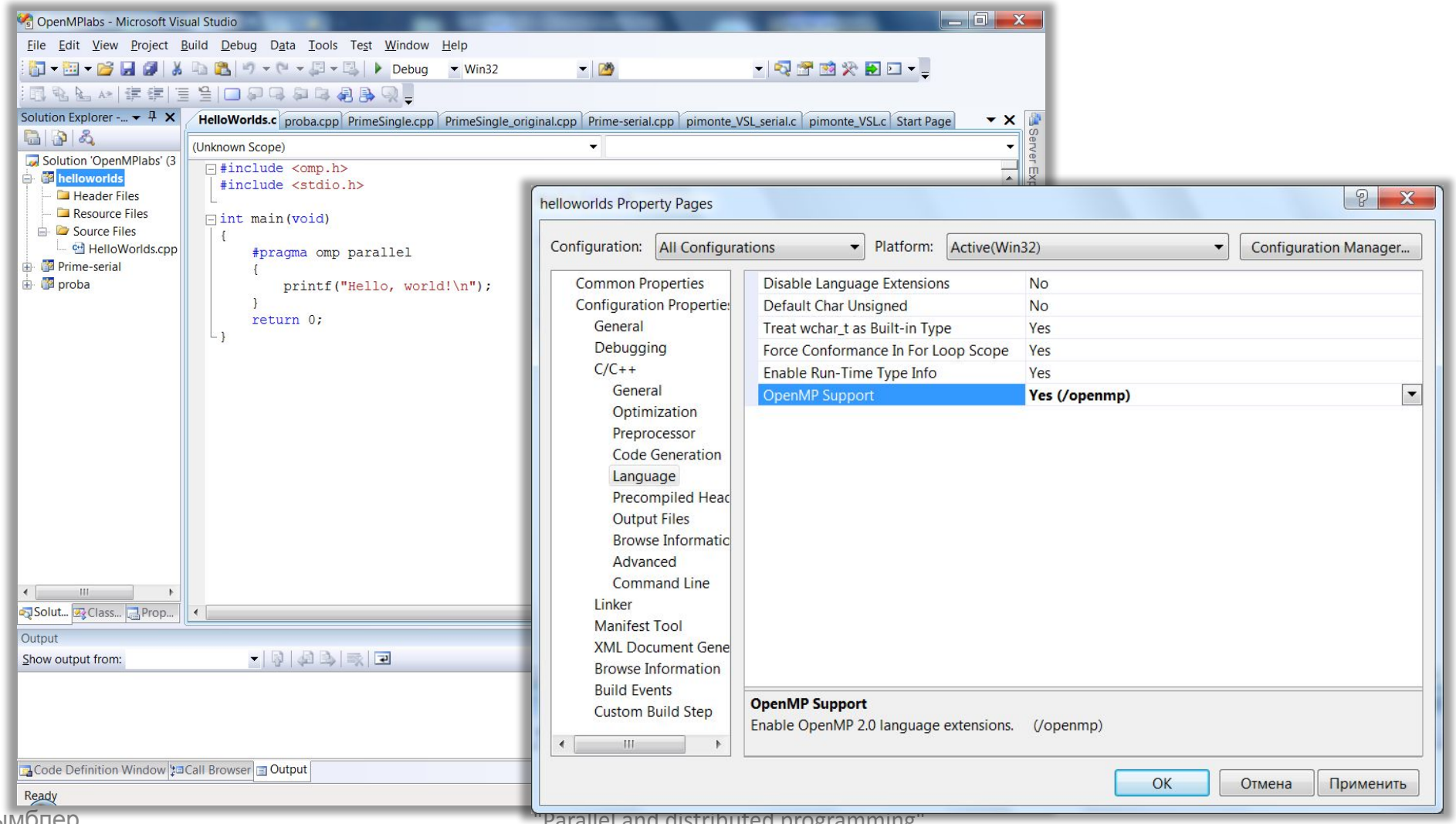
Parallel code

```
void main()
{
#pragma omp parallel
{
    printf("Hello!\n");
}
}
```

Result (for 2 threads)

```
Hello!
Hello!
```

Simple OpenMP-program



Advantages of OpenMP

- Gradual (incremental) paralleling
  - You can parallelize sequential programs in phases, without changing their structure.
- The uniqueness of the code
  - No need to support serial and parallel version of the program, as directives are ignored by conventional compilers.
- Code efficiency
  - Accounting for and use of the shared memory systems.
- Mobile code
  - Language support C / C + +, Fortran and OS Windows, Unix / Linux.

OpenMP directives

- Directives OpenMP - directive C / C + + compiler
  `# pragma`.
  - compile option `/ openmp`.
- The syntax of OpenMP directives
  - `# pragma omp direktive_name[options]`
- Examples:
  - `#pragma omp parallel`
  - `#pragma omp for private(i, j) reduction(+: sum)`

Functions of OpenMP library

- The assignment of the library:
  - control or view the OpenMP-programs
    - `omp_get_thread_num ()` returns the current thread
  - explicit synchronization of threads on the basis of "locks"
    - `omp_set_lock ()` sets the "lock"
- Connection library
  - `#include "omp.h"`

Environment variables OpenMP

- Environment variables control the behavior of the application.
  - `OMP_NUM_THREADS` - the number of threads in a parallel region
  - `OMP_DYNAMIC` - enable or disable the dynamic changes in the number of threads.
  - `OMP_NESTED` - enable or disable nested parallel regions.
  - `OMP_SCHEDULE` - the way the iterations in the loop.
- Function parameter assignment change values corresponding environment variables.
- Macro `_OPENMP` for conditional compilation of individual sections of the source code, specific to the parallel version of the program.

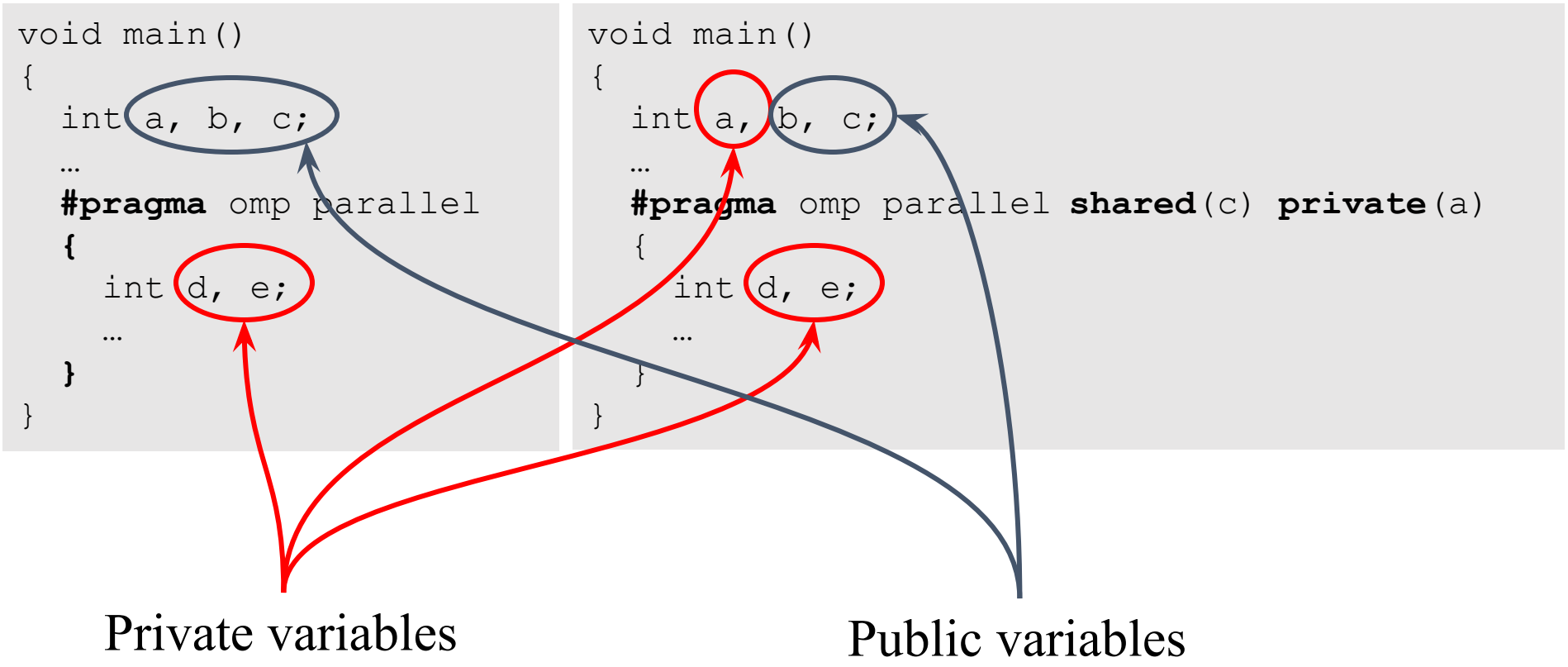
Variable Scope

- *Common variable (shared) -* global to the thread variable, is available for all versions threads.
- *Private variable (private) -* a local variable thread; accessible to only one modification (create it) thread only for the duration of this thread.
- A variable default
  - variables defined **outside** a parallel region - **general**;
  - variables defined **in** a parallel region - **private**.
- Clear indication of the scope - the parameters of directives:
  - `#pragma omp parallel shared(buf)`
  - `#pragma omp for private(i, j)`

Private and public variables

```
void main()
{
  int a, b, c;
  …
  #pragma omp parallel
  {
    int d, e;
    …
  }
}
```

```
void main()
{
  int a, b, c;
  …
  #pragma omp parallel shared(c) private(a)
  {
    int d, e;
    …
  }
}
```



Private and public variables

```
void main()
{
  int rank;
 #pragma omp parallel
 {
    rank = omp_get_thread_num();
 }
 printf("%d\n", rank);
}
```

One random number from a range of0..OMP_NUM_THREADS-1

```
void main()
{
  int rank;
 #pragma omp parallel
 {
   rank = omp_get_thread_num();
   printf("%d\n", rank);
 }
}
```

OMP_NUM_THREADS random numbers (possibly repeating) from a range of0..OMP_NUM_THREADS-1

Private and public variables

```
void main()
{
  int rank;
#pragma omp parallel shared (rank)
  {
    rank = omp_get_thread_num();
    printf("%d\n", rank);
  }
}
```

OMP_NUM_THREADS
random numbers
(with repetitions)
in the range
0..OMP_NUM_THREADS-1

```
void main()
{
  int rank;
 #pragma omp parallel private (rank)
  {
    rank = omp_get_thread_num();
    printf("%d\n", rank);
  }
}
```

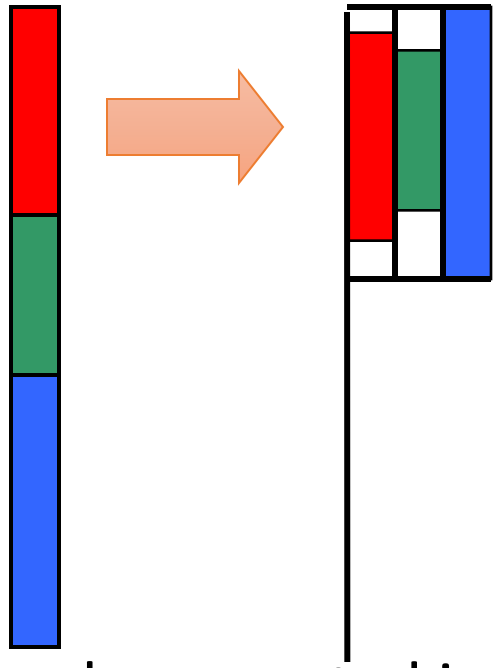
OMP_NUM_THREADS
numbers in the range
0..OMP_NUM_THREADS-1
(no repetitions,
in random order)

Distribution of computations

- Directive computation distribution between threads in a parallel region
  - `sections` - functional parallelism separate pieces of code
  - `single` and `master` - the directive to specify the code only one thread
  - `for` - parallelization of loops
- Beginning implementation of directives by default is not synchronized.
- Completion of directives by default is synchronous.

Directive `sections`

```
#pragma omp parallel sections
{
    #pragma omp section
    Job1();
    #pragma omp section
    Job2();
    #pragma omp section
    Job3();
}
```



- Explicit definition of blocks of code that can be executed in parallel.
  - each piece is performed once
  - different pieces are performed by different threads
  - end directive is synchronized.

Directive `single`

- Specifies a code that is only one (the first one who came to this point) thread.
  - The rest of the thread is passed the appropriate code and expect the end of its run.
  - If waiting for other threads may optionally be added parameter `nowait`.

```
#pragma omp parallel
{
#pragma omp single
  printf("Start Work #1.\n");
  Work1();
#pragma omp single
  printf("Stop Work #1.\n");
#pragma omp single nowait
  printf("Stop Work #1 and start Work #2.\n");
  Work2();
}
```

Directive `master`

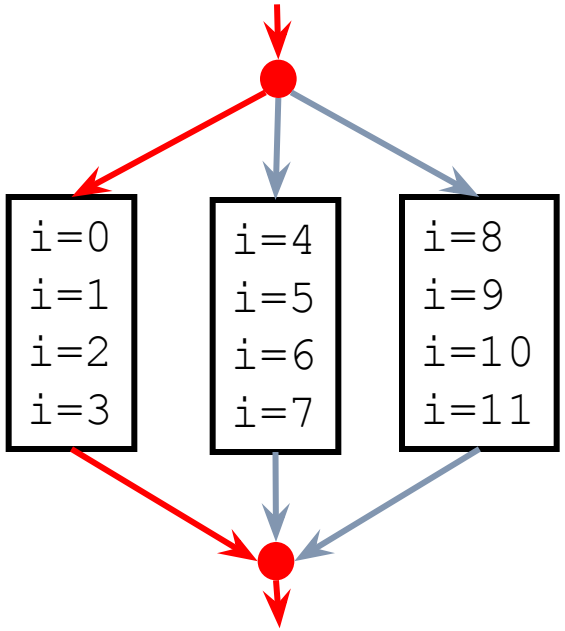
- Specifies a code that is only one main thread.
- The rest of the thread is passed the appropriate code without waiting for the end of its run.

```
#pragma omp parallel
{
#pragma omp master
  printf("Beginning Work1.\n");
  Work1();
#pragma omp master
  printf("Finishing Work1.\n");
#pragma omp master
  printf("Finished Work1 and beginning Work2.\n");
  Work2();
}
```

Parallelization of loops

```
#pragma omp parallel
{
  #pragma omp for
  for (i=0; i<N; i++) {
    res[i] = big_calc();
  }
}
```

```
#pragma omp parallel for
for (i=0; i<N; i++) {
  res[i] = big_calc();
}
```



- The loop counter is the default private variable.
- By default computations are distributed evenly between the filaments.

Parallelization of loops

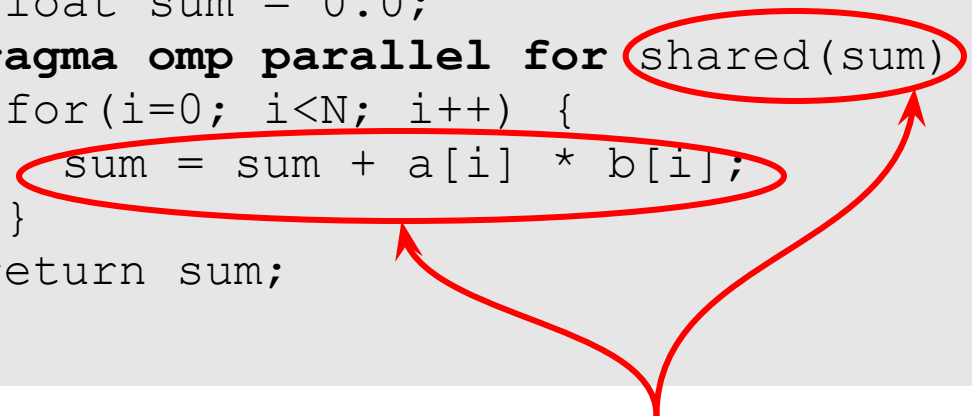
```
void work(float c[], int N)
{
  float x, y;
  int i;

#pragma omp parallel for private(x, y)
  for(i=0; i<N; i++) {
    x = a[i];
    y = b[i];
    c[i] = x + y;
  }
}
```

- You can explicitly define the private data of the cycle.

Parallelization of loops

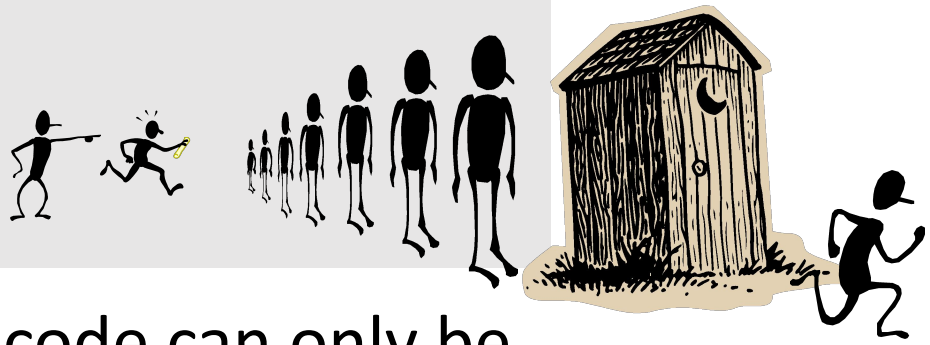
```
float scalar_product(float a[], float b[], int N)
{
  float sum = 0.0;
#pragma omp parallel for shared(sum)
   for(i=0; i<N; i++) {
     sum = sum + a[i] * b[i];
   }
  return sum;
}
```



- Uncontrolled change thread shared data leads to logical errors.

Critical section in cycles

```
float scalar_product(float a[], float b[], int N)
{
  float sum = 0.0;
#pragma omp parallel for shared(sum)
   for(i=0; i<N; i++) {
#pragma omp critical
     sum = sum + a[i] * b[i];
   }
  return sum;
}
```



- At any time, a critical section code can only be executed by one thread.

Reduction in a loop

- *Reduction* involves identifying for each thread private variable to calculate the "partial" results and automatic operation of "merging" of partial results.

```
float scalar_product(float a[], float b[], int N)
{
  float sum = 0.0;

#pragma omp parallel for reduction(+:sum)
  for(i=0; i<N; i++) {
    sum = sum + a[i] * b[i];
  }
  return sum;
}
```

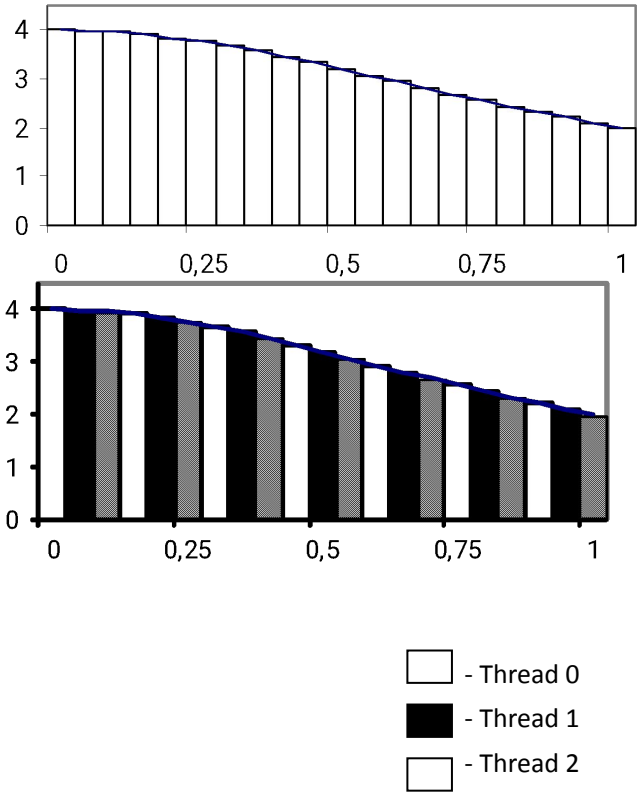
Operation	Initial value
+	0
*	1
-	0
^	0
&	~0
|	0
&&	1
||	0

Reduction in a loop

```
#include <stdio.h>
#include <time.h>
long long num_steps = 1000000000;
double step;
int main(int argc, char* argv[])
{
  clock_t start, stop;
  double x, pi, sum=0.0;
  int i;

  step = 1./(double)num_steps;
  start = clock();
#pragma omp parallel for private(x) reduction(+:sum)
  for (i=0; i<num_steps; i++) {
    x = (i + 0.5)*step;
    sum = sum + 4.0/(1.+ x*x);
  }
  pi = sum*step;
  stop = clock();
  printf("PI=%15.12f\n", pi);
  printf("Time=%f sec.\n",((double)(stop - start)/1000.0));
  return 0;
}
```

$$\pi = \int_0^1 \frac{4}{1+x^2}\,dx$$



Directive `for`

- Format directives
  - `#pragma omp parallel for [clause ...]`
    `for (…)`
- Types of parameter `clause`
  - `private(list_of_variables)`
  - `firstprivate(list_of_variables)`
  - `lastprivate(list_of_variables)`
  - `reduction(operator: variable)`
  - `ordered`
  - `nowait`
  - `schedule(form_of_the_distribution[, size])`

Parameter `firstprivate`

- Defines private loop variables `for`, that early in the cycle take the values of the sequential part of the program.

```
myrank = omp_get_thread_num();
#pragma omp parallel for firstprivate(myrank)
for (i=0; i<N-1; i++) {
  a[i] = b[i] + b[i+1] + myrank;
  myrank = myrank + N % (i+1);
}
```

Parameter `lastprivate`

- Defines the private variables that the end of the `for` loop of these values, as if the loop is executed sequentially.

```
#pragma omp parallel for lastprivate(i)
for (i=0; i<N-1; i++) {
  a[i] = b[i] + b[i+1];
}
// здесь i=N
```

Parameter `ordered`

- Defines the code in the loop `for`, performed in exactly the order in which he would perform sequential execution cycle.

```
#pragma omp for ordered schedule(dynamic)
for (i=start; i<stop; i+=step)
  Process(i);

void Process(int k)
{
#pragma omp ordered
  printf(" %d", k);
}
```

Parameter `nowait`

- Avoids the implicit barrier at the end of the directive for.

```
#pragma omp parallel
{
#pragma omp for nowait
  for (i=1; i<n; i++)
    b[i] = (a[i] + a[i-1]) / 2.0;
#pragma omp for nowait
  for (i=0; i<m; i++)
    y[i] = sqrt(z[i]);
}
```

Distribution of loop iterations

- The iterations in the directive `for` regulated parameter `schedule (form_of_the_distribution[size])`
  - `static` - iterations are divided into a series of iterations and the `size` of statically shared between threads, and if `size` parameter is not specified, the iterations are divided between flows evenly and continuously
  - `dynamic` - the distribution of iterative block is dynamic (default `size` = 1)
  - `guided` - the iteration block size decreases exponentially with each distribution, `size` determines the minimum block size (default `size` = 1)
  - `runtime` - usually determined by the variable distribution `OMP_SCHEDULE` (using `runtime` parameter `size` should not be set)

Example

```
// The amount of work in iterations is predictable and is
// about the same
#pragma omp parallel for schedule(static)
for(i=0; i<n; i++) {
  invariant_amount_of_work(i);
}
```

```
// The amount of work in iterations can vary significantly
// or unpredictable
#pragma omp parallel for schedule(dynamic)
for(i=0; i<n; i++) {
  unpredictable_amount_of_work(i);
}
```

Example

```
// The threads are suitable for distribution point iterations
// at different times, the amount of work in iterations
// predictable, and about the same
#pragma omp parallel
{
  #pragma omp sections nowait
  {
  ...
  }
  #pragma omp for schedule(guided)
  for(i=0; i<n; i++) {
    invariant_amount_of_work(i);
  }
}
```

Synchronization algorithms

- Directive explicit synchronization
  - `Critical`
  - `barrier`
  - `atomic`
- Directive implicit synchronization
  - `#pragma omp parallel`

Directive `critical`

- Defines *the critical section* - the section of code that is executed simultaneously by more than one thread.

```
#pragma omp parallel shared(x, y) private(x_next, y_next)
{
#pragma omp critical (Xaxis_critical_section)
  x_next = Queue_Remove(x);
  Process(x_next);
#pragma omp critical (Yaxis_critical_section)
  y_next = Queue_Remove(y);
  Process(y_next);
}
```

Directive `atomic`

- Defines the *critical section for a statement* of the form
  - `x++` and `++x`
  - `x--` and `--x`
  - `x+=выражение`, `x-=выражение` and other.

```
extern float a[], *p = a, b;

// Protection of racing data
// when updating multiple threads
#pragma omp atomic
a[index[i]] += b;

// Protection of racing data
// when updating multiple threads
#pragma omp atomic
p[i] -= 1.0f;
```

Directive `barrier`

- Determines the *barrier* - the point in the program, which should reach every thread to all the threads to continue the calculation.

```
#pragma omp parallel shared (A, TmpRes, FinalRes)
{
  DoSomeWork(A, TmpRes);
  printf("Processed A into TmpRes\n");
#pragma omp barrier
  DoSomeWork(TmpRes, FinalRes);
  printf("Processed B into C\n");
}
```

```
// Directive should be part of the structural unit
if (x!=0) {
  #pragma omp barrier
  ...
}
```

Directive `barrier`

```
int main()
{
sub1(2);
sub2(2);
sub3(2);
}
void sub1(int n)
{
int i;
#pragma omp parallel private(i) shared(n)
{
#pragma omp for
for (i=0; i<n; i++)
sub2(i);
}
}
void sub2(int k)
{
#pragma omp parallel shared(k)
sub3(k);
}
void sub3(int n)
{
work(n);
#pragma omp barrier
work(n);
}
```

Directives and parameters

Parameter	Directive					
	parallel	for	sections	single	parallel for	parallel sections
if	✔				✔	✔
private	✔	✔	✔	✔	✔	✔
shared	✔	✔			✔	✔
default	✔				✔	✔
firstprivate	✔	✔	✔	✔	✔	✔
lastprivate		✔	✔		✔	✔
reduction	✔	✔	✔		✔	✔
copyin	✔				✔	✔
schedule		✔			✔	
ordered		✔			✔	
nowait		✔	✔	✔		

Operation time

```
double start;
double end;

start = omp_get_wtime();
// Operations
end = omp_get_wtime();
printf(“Work took %f sec. time.\n”, end-start);
```

Threads count

```
// False
np = omp_get_num_threads(); // It was not yet performed
                // FORK
#pragma omp parallel for schedule(static)
for (i=0; i<np; i++)
  work(i);

// True
#pragma omp parallel private(i)
{
i = omp_get_thread_num();
work(i);
}
```

Synchronization functions

- As castles using shared variables of `omp_lock_t`. These variables should only be used as parameters of the synchronization primitives.
- Initializes the lock associated with the variable

`lock`

```
void omp_init_lock(omp_lock_t *lock)void
```

- Removes the lock associated with the variable `lock`

```
void omp_destroy_lock(omp_lock_t *lock)
```

Synchronization functions

- Causes the calling thread to wait for the release of the castle, and then captures it

```
void omp_set_lock(omp_lock_t *lock)
```

- Releases the lock, if he had been captured earlier thread

```
void omp_unset_lock(omp_lock_t *lock)
```

- Tries to lock the specified lock. If this is not possible, return `false`

```
void omp_test_lock(omp_lock_t *lock)
```

Example

```
#include <omp.h>
int main()
{
  omp_lock_t lck;
  int id;
  omp_init_lock(&lck);
  #pragma omp parallel shared(lck) private(id)
  {
  id = omp_get_thread_num();
  omp_set_lock(&lck);
  printf("My thread id is %d.\n", id);
 // only one thread at a time can execute this printf
  omp_unset_lock(&lck);
  while (! omp_test_lock(&lck)) {
   skip(id); /* we do not yet have the lock, so we must do something else */
  }
  work(id); /* we now have the lock and can do the work */
  omp_unset_lock(&lck);
  }
  omp_destroy_lock(&lck);
}
```

Conclusion

- Programming model in shared memory
- Model "pulsating" parallelism FORK-JOIN
- OpenMP standard
- Basic concepts and features OpenMP

Information Resources

- Introduction to OpenMP www.llnl.gov/computing/tutorials/workshops/workshop/openMP/MAIN.html
- Chandra R., Menon R., et al. Parallel Programming in OpenMP. Morgan Kaufmann, 2000.
- Quinn M.J. Parallel Programming in C with MPI and OpenMP. McGraw-Hill, 2004.
- www.openmp.org

Минимальный перебор в игровых деревьях. Альфа-бета отсечения. Построение игровых программ

Удалова Татьяна

85М21

Деревья решений

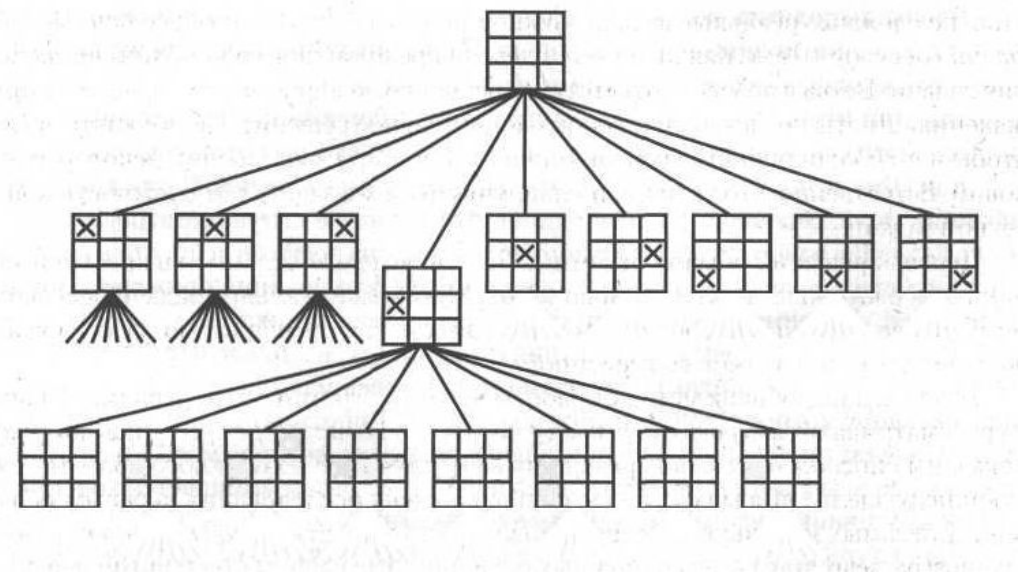
- Узел дерева – один шаг решения задачи
- Ветвь – решение, которое ведёт к более полному решению
- Листы – окончательное решение

Цель: Найти «наилучший» путь от корня до листа.

Проблема: Деревья решений обычно огромны

Игровые деревья

- Моделирование стратегических настольных игр (крестики нолики)
- Ветвь, выходящая из узла - ходы одного из игроков



- 362880 сценария развития игры [1]

Минимаксный перебор

- Минимизировать максимальное значение, которое может иметь позиция для противника после следующего хода

Т.Е

- Ищем наименьшие потери из тех, которые нельзя предотвратить принимающему решения субъекту в наихудших для него обстоятельствах.

Крестики-нолики(1)

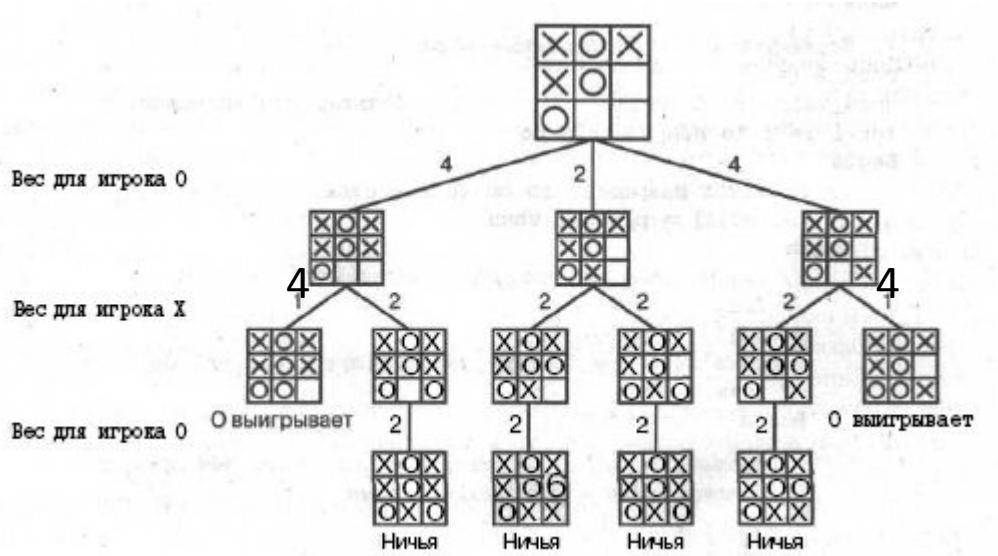
4 значения позиции поля:

- 4 –игрок выиграет
- 3 – ситуация не ясна
- 2 – ничья
- 1 – противник выиграет

На основании заданных значений реализуется функция оценки состояния игры.

Крестики-нолики(2)

- Дерево игры крестики-нолики в конце партии[1]



Игрок X минимизирует свой потерй

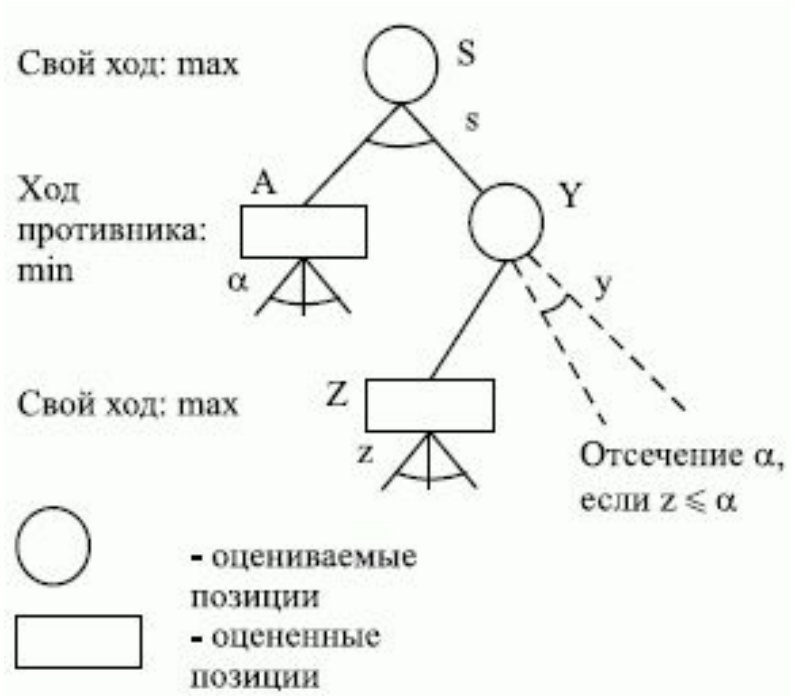
Игрок 0 максимизирует свой выигрыш

Альфа-бета отсечения(1)

Оптимизация минимаксного перебора

- Сравнение наилучших оценок, полученных для полностью изученных ветвей, с наилучшими предполагаемыми оценками для оставшихся

Альфа-бета отсечения(2)



Предположим, что z<=a. После анализа узла Z, когда справедливо соотношение y<=z<=a<=s, ветви дерева, выходящие из узла Y, могут быть отброшены (альфа-отсечение).[2]

Альфа-бета отсечения(4)

Правила вычисления альфа-бета:

- у MAX вершины значение a равно наибольшему в данный момент значению среди окончательных возвращенных значений для ее дочерних вершин
- у MIN вершины значение b равно наименьшему в данный момент значению среди окончательных возвращенных значений для ее дочерних вершин

Альфа-бета отсечения(5)

Правила прекращения поиска:

- можно не проводить поиска на поддереве, растущем из всякой MIN вершины, у которой значение b не превышает значения a всех ее родительских MAX вершин
- можно не проводить поиска на поддереве, растущем из всякой MAX вершины, у которой значение a не меньше значения b всех ее родительских MIN вершин

Альфа-бета отсечения(6)

[2]

Программная реализация

Игра крестики-нолики включающая в себя:

- Альфа-бета отсечения для расчёта следующего хода
- Возможность выбора глубины рекурсии при моделировании последующих ходов:
  - Глубина рекурсии менее 5 ходов – приложение сопротивляется противнику
  - Глубина более 5 ходов гарантирует конкурентоспособность приложения

Литература

1. Rod Stephens. Ready-to-run Delphi© Algorithms. Wiley Computer Publishing.
2. Интернет-Университет Информационных Технологий. Интеллектуальные робототехнические системы. Лекция: Методы поиска решений.
3. Википедия — свободная энциклопедия. Альфа-бета отсечение.
4. Donald E. Knuth and Ronald W. Moor. Анализ альфа-бета отсечений. Перевод: Павел Н. Дубнер. 1998.
5. Михаил Лопаткин. Минимаксный перебор в игровых деревьях. Зимняя студенческая школа-практикум «Высокопроизводительные вычисления» Нижегородский государственный университет, Intel. 2010.

Вопросы