МІРТ, ЗКШ, февраль 2015
Лекция про перебор и жадность

Копелиович Сергей Собрано 26 февраля 2015 г. в 15:31

Содержание

Жадность

Жадность – корень всех зол.

Джефри Чосер

1.1. Сортировка

Задача. Даны n работ, у каждой есть время выполнения t_i. Нужно к моменту времени D выполнить как можно больше работ.

Решение. Отсортируем работы в порядке возрастания t_i. Будем выполнять работы в таком порядке, пока не закончится время.

Задача. Даны n работ, у каждой есть время выполнения t_i и бонус, который мы получим, если её сделаем, p_i. Нужно к моменту времени D выполнить работ на макисмальную суммарную стоимость. Работу, конечно, можно делать только целиком.

Решение. Это рюкзак... Жадно не решится. Если бы работы можно было выполнять кусками (непрерывная версия задачи о рюкзаке), достаточно отсортировать по убыванию $\frac{p_i}{t_i}$.

Задача. Даны n работ, у каждой есть время выполнения t_i и дедлайн d_i – момент времени, когда работа должна быть выполнена. Нужно выполнить все работы вовремя.

Решение. Отсортируем работы в порядке возрастания d_i. Будем выполнять работы в таком порядке.

Задача про коробки. Нужно построить башню из всех данных коробок. У каждой есть масса m_i и прочность s_i (какую суммарную массу можно поставить сверху).

Решение. Вниз можно поставить коробку с максимальным $s_i + m_i$ т.к. для того, что i-ю коробку можно было поставить в самый низ, достаточно, чтобы $s_i \geq \sum_{j \neq i} m_j \Leftrightarrow s_i + m_i \geq M$, где M – суммарная масса всех коробок. Решение: сортировка по убыванию $s_i + m_i$.

1.2. Общая идея для придумывания компаратора

```
bool isLess( Object a, Object b ) {
  return order(a, b) is strictly better than order(b, a)
  // иногда задача более очевидна для двух объектов, чем для N
  // часто можно написать F(a, b) < F(b, a), где F - некоторая функция оценки
}
```

Пример с коробками: важно, сколько можно поставить сверху на часть башни, которую мы уже построили, поэтому $F(i, j) = \min(s_i - m_j, s_j)$. Мы получили другой, но тоже корректный, компаратор.

Когда так можно делать? Когда полученный порядок транзитивный.

Задача про два станка. Каждую деталь нужно обработать сперва на первом станке, затем на втором. На каждом станке в каждый момент времени можно обрабатывать только

одну детать. Каждую деталь в каждый момент времени можно обрабатывать только на одном из станков. Известны времена выполенения для каждой детали a_i, b_i. Нужно найти порядок обработки деталей (на обоих станках порядок должен быть одинаков!) так, чтобы обработать все детали как можно раньше.

Решение с помощью универсального компаратора. $F(i,j) = a_i + \max(a_j, b_i) + b_j$ – когда обе детали будут обработаны.

Упрощаем решение. $a_i + \max(a_j, b_i) + b_j = X - \min(a_j, b_i)$, где X – сумма всех четырёх чисел, константа. Получаем новый короткий компаратор:

```
bool isLess(i, j) { return min(a[j], b[i]) > min(a[i], b[j]); }
```

Замечание. Транзитивность предлагается доказать самостоятельно.

Упражнение. Для самостоятельного решения. В каком-то порядке выписать данные строки из круглых скобок так, чтобы получилась правильная скобочная последовательность.

1.3. Жадность вместе с динамикой

Задача: построить башню, но не из всех коробок, а из максимального возможного числа (полное условие см. выше).

Главная идея для решения: если мы зафиксировали множества коробок, из которых мы построим башню, мы знаем, что ставить их нужно в порядке убывания $m_i + s_i$.

Динамика: в порядке убывания $m_i + s_i$ мы уже рассмотрели j коробок из них k взяли в башню, и вся эта конструкция может выдержать коробки суммарной массы f. Получили $f[j,k] \to \max$. $\mathcal{O}(n^2)$.

Другая задача. Школьники выбираются из ямы глубины H, у каждого есть высота h_i и длина рук l_i. Школьники бесконечно сильные и ловкие, поэтому могут строить любые башни друг из друга. Самый верхний в башне школьник может выбраться, если $h_1+h_2+...+h_k+l_k \geq H$. Нужно, чтобы выбралось как можно больше школьников.

Если могут выбраться все. То последним выберется тот, у которого $h_i + l_i \geq H$, т.е. школьников можно расставить в порядке $h_i + l_i$. А те школьники, кто не выберется никогда просто несколько уменьшат H...

Динамика: перебираем школьников в порядке убывания $h_i + l_i$, j уже перебрали, k выбрались, не выбравшиеся имеют суммарную высоту f. Получили $f[j,k] \to \max$. $\mathcal{O}(n^2)$.

1.4. Избавляемся от динамики

Задача: ещё раз решим задачу про башню из максимального числа коробок. На этот раз за $\mathcal{O}(n \log n)$.

Идея! Будем в порядке возрастания $m_i + s_i$ строить башню сверху вниз. Пытаемся подложить очередную коробку под низ уже построенной конструкции. Если получилось, так и сделаем. Если нет, то, может быть, она легче какой-то другой коробки в башне и выгодно сделать замену (количество коробок не изменится, зато суммарная масса уменьшится).

Алгоритм

```
vector<Box> boxes; // наши коробки
sort(boxes.begin(), boxes.end(), []( Box a, Box b ){ return a.m + a.s < b.m + b.s; });
int M = 0; // суммарная масса башни
auto comparator = []( Box a, Box b ) { return a.m > b.m; };
multiset <int, decltype(comparator)> boxes(comparator);
for (Box b : boxes) {
  if (b.s >= M)
    M += b.m, boxes.insert(b);
  else {
    int Max = boxes.begin()->m;
    if (b.s >= M - Max && b.m < Max) {
      M -= Max, boxes.erase(boxes.begin());
      M += b.m, boxes.insert(b);
    }
  }
}
```

Перебор

Мм... а почему так быстро работает?

Случай из жизни

2.1. Основы

Основы всего – рекурсия, прекалк, отсечение по времени, запоминание.

Рекурсия и глобальные переменные.

```
int used[N][N];
void go( int x, int y ) {
  if (used[y][x]) return;
  used[y][x] = 1;
  go(x + 1, y - 1);
  if (someInterestingEvent) {
    used[y][x] = 0; // стандартная ошибка - забыть убрать пометку
    return;
  }
  go(x - 1, y - 1);
  go(y + 1);
  used[y][x] = 0; // стандартная ошибка - забыть убрать пометку
}
```

Запоминание. При решении перебором задачи "поиск гамильтонова пути", если добавить строчку "`if (++was[SetOfVisited][LastVertex] > 1) return;`", асимптотика решение изменится с $\mathcal{O}(n!)$ до $\mathcal{O}(2^n n^2)$. От перебора до ленивой динамики один шаг.

Хеш состояния. По ходу рекурсии легко пересчитывать хеш состояния:

```
used[y][x] = 1, hash += deg[y * w + x]; // прибавили

used[y][x] = 0, hash -= deg[y * w + x]; // вычли обратно
```

Таким образом можно получить отсечение запоминанием для сложных состояний:

```
long long hash = 0;
set<long long> was; // множество уже посещённых состояний
...
if (was.count(hash)) return;
```

Прекалк. Рассмотрим сложную задачу "посчитать число двудольных графов из n вершин с точностью до изоморфизма". $n \leq 10$. Если ваше решение работает целых 5 минут, это повод запустить его один раз на всех n, посчитать ответы, а в систему отправить решение, которое мгновенно выводит один из 10 правильных предподсчитанных ответов.

Отсечение по времени. Если задача – найти гамильтонов путь, возможно, ваш перебор быстро находит путь, если такой путь есть в графе. А вот если такого пути в графе нет, работает долго. Значит, если перебор работает больше 2 секунд, это повод завершиться с результатом "No path".

```
#include <ctime>
const double TL = 2; // seconds
inline bool isTimeLimit() {
  static int is = 0, cnt = 0;
  if (++cnt == 1000) { // каждый раз вызывать clock() - слишком долго
    cnt = 0;
    is |= clock() > (TL - 0.1) * CLOCKS_PER_SEC; // время работы с запуска программы
  }
}
```

2.2. Отсечение по ответу

Пусть мы ищем в графе самый длинный простой путь. Эта задача не проще чем поиск гамильтонова пути (т.к. если гамильтонов путь есть, мы его найдём). Решение:

```
unordered_set<long long> was; // посещённые состояние
vector<int> g[N]; // соседи для каждой из вершин
int best = 0; // длина наибольшего из путей
void go( long long visited, int last, int dep ) { // да, в графе не более 63 вершин
  if (!was.insert(visited * n + last]).second) // хеш, запоминание
    return; // insert вернул, что такое состояние уже было
  best = max(best, dep);
  if (dep + dfs(visited, v) <= best) // отсечение по ответу!
    return; // dfs за O(E) вернул количество достижимых непосещённых вершин
  visited |= 1LL << last, dep++;
  for (int x : g[v])
    go(visited, x, dep);
}
```

Случай другой, поиск кратчайшего пути перебором.

```
int used[N], d[N]; // посещённость вершины, расстояние до вершины
vector<Edge> g[N]; // соседи для каждой из вершин
int best = infinity; // длина наименьшего из путей
void go( int last, int length ) { // length >= 1
  if (last == GOAL)
    best = min(best, last);
  if (length >= best)
    return; // отсечение по ответу!
  if (used[last] && d[last] <= length)
    return; // отсечение запоминанием
  d[last] = length, used[last] = 1;
  for (Edge e : g[v])
    go(visited, e.next, length + e.weight); // e.weight >= 0
}
```

Что забавно, не смотря на то, что в худшем случае этот код работает за экспоненциальное время, в среднем он ведёт себя не хуже алгоритма Форда-Беллмана.

Замечание. Большинство задач на перебор можно интерпретировать, как поиск кратчайшего или самого длинного пути в некотором большом графе.

2.3. Усиливаем отсечение по ответу: жадность, iterative deepening

Отсечение по ответу сработывает чаще, если мы заранее знаем ответ.

Жадность помогает нам пойти в первую очередь в ветку перебора, который даст нам ответ близкий к оптимальному. Например в задачах про гамильтонов путь и простой путь максимальной длины хорошо работает правило Варнсдорфа – сперва идти в вершину минимальной остаточной степени.

Iterative Deepening – внешний перебор ответа.

```
bool solve( int answer ) { ... }
int main() {
  int ans = 1;
  while (!solve(ans))
    ans++;
  printf("answer = %d\n", ans);
}
```

2.4. Алгоритм A^*

Алгоритм A^* ("А-звёздочка") – модификация Дейкстры. Данный алгоритм решает задачу поиска кратчайшего пути между двумя заданными вершинами. В отличии от Дейкстры, которая ищет расстояние от заданной вершины до всех, A^* ищет путь лишь между двумя вершинами.

Дейкстра: перебирать вершины в порядке возрастания $d[v]$. Алгоритм Дейкстры каждую вершину достаёт из кучи не более одного раза и работает за время $\mathcal{O}(E)$

Алгоритм A^*: перебирать вершины в порядке возрастания $d[v]+f(v)$, где $f(v)$ – функция оценки расстояния от вершины v до конечной вершины. A^* для произвольной функции f может одну и ту же вершину достать из кучи несколько раз, и в худшем случае работает за экспоненциальное время.

Время работы для хорошей f. Если функция f вместе с весами рёбер удовлетворяют неравенству треугольника $(\forall a, b : f(a) \le f(b) + w(a, b))$, то A^* каждую вершину переберёт не более одного раза. Без доказательства. Доказывается также, как и корректность алгоритма Дейкстры.

Замечание. Пусть вершины – точки на плоскости, $w(a, b) = |a - b|$, а $f(v) = |v - end|$, тогда алгоритм A^* при возможности пойдёт от вершины $start$ к вершине end практически по прямой (не будет перебирать лишние вершины). Какие вершины переберёт A^*? Он перебирает их в порядке возрастания $d[v]+f(v)$, значит, ровно те, у которых $d[v]+f(v) \le d[end]+f(end) = d[end] + 0$. Значит, если функция f обладает двумя свойствами: $f(v) \ge 0, f(end) = 0$, то A^* переберёт вершин не больше чем "Дейкстра с break" (алгоритм Дейкстры, останавливающийся при дохождении до вершины end и перебирающий только вершины $v: d[v] \le d[end]$). Далее мы везде предполагаем, что функция f такими двумя свойствами обладает.

Как это связано с перебором? Рассмотрим два подхода. Первый – рекурсивный перебор отсечением по ответу:

```
int best = infinity; // растояние, которое мы минимизируем
void go(state, distance) {
  if (distance + f(state) >= best) // отсечение по ответу!
    return;
  if (state is final) {
    best = distance;
    return;
  }
  for (...) {
    go(nextState, newDistance);
  }
}
```

А второй – алгоритм A^*, применённый к тому же графу, на котором мы только что пробовали кратчайший путь найти перебором. В алгоритме A^* нужна некая функция f, мы будем использовать ту же функцию, что и для отсечения по ответу.

```
unordered_map<State, int> distance;
multiset<pair<State, int>> heap;
void insert ( int d, State s ) {
  if (!distance.count(s) || distance[s] > d)
    distance[s] = d, heap.insert({d + f(s), s});
}
insert(0, start);
while (heap.size()) {
  State s = heap.begin()->second;
  if (s == end)
    break; // нашли путь до конца
  heap.erase(heap.begin());
  for (...) {
    insert(newDistance, nextState);
  }
}
```

Утверждение. Первая версия переберёт все состояния, что и вторая, и, возможно, некоторые ещё. Доказательство: пусть вторая версия побывала в состоянии s, тогда $d[s] + f(s) \leq d[end] \leq best$ (здесь $best$ – переменная из кода выше), а значит, в рекурсивном переборе в момент обработки состояниее s не сработает отсечение по ответу. ■

Если добавить “отсечение запоминанием” в первую версию, утверждение останется верным, но перестанет быть очевидным. Есть ли у перебора плюсы перед A^*? Да. Например, при переходе от сложного состояния к следующего и возврате обратно мы обычно тратим $\mathcal{O}(1)$ времени. В случае с A^*, нам нужно каждый раз подгружать состояние целиком (а оно может быть большим! например, двухмерный массив – какие клетки уже посещены, а какие ещё нет).

2.5. Жадность и перебор. Плавный переход.

Пусть мы умеем задачу решать жадностью, которая не всегда находит оптимальный ответ... Тогда есть две крайности: запускать жадный алгоритм и запускать полный перебор. Грубо говоря, жадность ищет путь в графе и всегда перебирает только одно ребро, перебор ищет путь в графе и перебирает все возможные рёбра.

Промежуточная версия. Перебор, который сортирует рёбра в порядке, который задаёт жадность и перебирает лучшие k рёбер. Если $k = 1$, он ведёт себя как жадность, если же $k = +\infty$, то он ведёт себя как полный перебор. Какое выбрать k? Возьмём слишком маленькое, получим `WA`, возьмём слишком большое, получим `TL`... Чем больше k, тем дольше работает перебор, поэтому добавим отсечение по времени и будем перебирать k внешним циклом в main-е: `for (k = 1;; k++)` (здравствуй, Iterative Deepening!).

Другой подход. Иногда у жадности есть некоторая степень свободы. Например, пусть мы используем правило Варнсдорфа для поика гамильтонова пути и пытаемся идти в вершину "минимальной степени". Вершин минимальной степени может быть несколько (это и есть степень свободы), давайте пойдём в случайную. Теперь если запустить жадность несколько раз, она по-разному работает. Давайте пускать её, пока не `TL`. Этот метод близок в чём-то к перебору. Только перебор, который не успевает перебрать все варианты и останавливается по отсечении по времени, хуже тем, что сперва спускается вниз в какую-то одну ветку, а там внизу перебирает все варианты. А наша новая идея "много раз запустить жадность" успеет перебрать столько же вариантов, зато попробует сходить в разные ветки.

А если у жадности нет никакой свободы? Тогда можно переходам сопоставить положительные вещественные веса $w_1, \dots, w_k$ и переходить в i-ю из веток рекурсии с вероятностью $\frac{w_i}{\sum w_j}$.

Поиск максимальной подклики

Контролирующее множество! Доминирующее множество! Независимое множество! Полнейший подграф! Красота да и только.

Реклама спецкурса по NP-трудным задачам

3.1. Определения, сведения

Независимое множество – подмножество вершин, попарно не соединённых рёбрами.

Клика – подмножество вершин, попарно соединённых рёбрами.

Независимое множество ↔ клика. Инвертируем граф. Там где рёбра есть, они появятся и наоборот. Заметим, что клики перешли в независимые множества и наоборот. Значит, если мы умеем искать один из этих двух объектов, второй тоже умеем.

Контролирующее множество – такое подмножество вершин, что у каждого из рёбер хотя бы один из концов лежит в множестве.

Независимое множество ↔ контролирующее множество. Дополнение независимого – контролирующее. И наоборот.

Доминирующее множество – такое подмножество вершин, что у каждой вершины или она сама, или хотя бы один сосед лежит в множестве.

Задачи поиск максимальной клики, максимального независимого множества, минимального контролирующего множества равносильны и все NP-трудны. Задача поиска минимального доминирующего множества также NP-трудна, но ей мы сегодня заниматься не будем.

3.2. Meet in the middle

Решим задачу поиска максимальной клики за $\mathcal{O}(2^{n/2})$. Здесь $|V| = n$. Разобьём V на две половины $V = A \sqcup B$. Переберём часть клики, которая лежит в A:

```
int ga[N]; // ga[a] - множество соседей вершины a в A
int gb[N]; // gb[a] - множество соседей вершины a в B
void go( int a, int clique, int canAdd, int partOfB ) {
  if (a == |A|) {
    // вот наша клика clique, нужно с ней что-то делать
    return;
  }
  go(a + 1, clique, canAdd);
  if ((canAdd >> a) & 1)
    go(a + 1, clique | (1 << a), canAdd & ga[a], partOfB & gb[a]);
}
```

Данный код работает за $2^{|A|}$. Теперь что делать с `clique`? Мы можем перебрать все клики в `partOfB`, а можем сказать, что уже известно `f[partOfB]` – размер максимальной клики. А вот код, который предподсчитает нам массив `f`.

```
int adj[N]; // adj[b] - множество соседей вершины b в B
int b = 0; // старший бит в partOfB
for (int X = 1; X < (1 << |B|); X++) { // X - подмножество B
  if (X == (1 << (b + 1)))
    b++;
  f[X] = max(f[X ^ (1 << b)], 1 + f[X & adj[b]])
}
```

Данный предподсчёт работает за время $\mathcal{O}(2^{|B|})$. Получили решение за $\mathcal{O}(2^{|A|} + 2^{n-|A|}) = \mathcal{O}(2^{n/2})$ при $|A| = \frac{n}{2}$.

3.3. Расщепление

Напишем для начала полный перебор для поиска максимального независимого множества:

```
// IS = independent set = независимое множество
long long g[N]; // соседи v
void go( long long IS, long long canAdd ) {
  int v = <any vertex from canAdd>;
  canAdd ^= 1LL << v;
  go(IS, canAdd);
  go(IS | (1LL << v), canAdd & ~g[v]); // выкидываем всех соседей v
}
```

Это аккуратно реализованный полный перебор с двумя рекурсивными вызовами. Идея называется "расщепление по вершине v". Асимптотика нашего решения уже не 2^n, а следующая: $T(n) = T(n-1) + T(n-1-deg_v)$. Здесь deg_v, количество вершин в `canAdd & g[v]`, т.е. остаточная степень вершины v.

Расщепление по вершине максимальной степени. Чем больше deg_v, тем быстрее работает. Как гарантировать себе, что deg_v, например, хотя бы 2? Добавить немного жадности! Если у вершины степень 0 или 1, её выгодно брать в IS. Получили, что $T(n) \leq T(n-1) + T(n-3)$, решаем реккурентность, получаем $\mathcal{O}(1.4656^n)$.

От вершин степени два тоже можно избавиться. Если все вершины степени 2, то граф – набор циклов. Для таких графов ответ можно посчитать жадно за линейное время. Значит максимальная степень вершины – хотя бы 3. Получаем $T(n-1)+T(n-4)$ и $\mathcal{O}(1.3803^n)$

Лучшее время на текущий момент: $\mathcal{O}(2^{n/4}) = \mathcal{O}(1.1893^n)$. Алгоритм весьма сложен. Более простую версию за $\mathcal{O}(2^{n/3}) = \mathcal{O}(1.26^n)$ можно посмотреть здесь: Тарьян'1977.

3.4. Поиск минимального контролирующего множества

Упрощение задачи. Давайте не искать минимальное контролирующее множество, а искать какое-нибудь контролирующее множество размера не более чем k, или говорить, что такого нет. Исходную задачу задачу можно решить добавив внешний перебор по k: `for (k = 1; !solve(k); k++)`

Рассмотрим граф, состоящий из непокрытых рёбер. У каждого ребра нужно покрыть один из двух концов. $T(k) = 2T(k-1)$. Получили решение за $\mathcal{O}(2^k)$.

Расщепление. Но лучше выбрать вершину v и сказать, что мы или берём её, или берём всех её соседей. Получаем $T(k) = T(k-1) + T(k-deg_v)$, где deg_v – степень вершины. Пусть

все степени не более двух, тогда, как и в прошлой серии, ответ строится жадно. Тогда максимальная степень хотя бы три. Получаем $T(k) = T(k-1) + T(k-3) = \mathcal{O}(1.4656^k)$. Заметим, что при маленьких k это гораздо лучше, чем очень похожее $\mathcal{O}(1.4656^n)$.

3.5. Поиск независимого множества в графах с ограниченной степенью.

Пусть есть граф, в котором степени всех вершин не более 4. Как найти максимальное независимое множество из k вершин?

Пусть v – вершина степени 2. Тогда в ответ нужно взять или её, или **обоих** её соседей, так как если мы возьмём только одного соседа, то всегда можно его заменить на v. Получаем $T(k) \leq T(k-1)+T(k-2)$. А пусть v – вершина степени 3, тогда $T(k) \leq T(k-1)+3T(k-2)$, т.к. у вершины степени 3 есть 3 способа выбрать каких-то двух соседей и добавить в ответ. Заметим, что если некоторые из соседей v соединены рёбрами, то рекурсивных вызовов становится меньше. Например, Если у вершины v три соседа a, b, c и присутствуют рёбра (a, b) и (b, c), то в случае, когда мы не берём v, мы обязаны взять в независимое множество a и c, получаем $T(k) = T(k-1) + T(k-2)$. Худший из рассмотренных случаев: $T(k) \leq T(k-1) + 3T(k-2)$ даёт асимптотику $\mathcal{O}(2.3028^n)$.

А что же делать с вершинами степени 4? Во-первых, заметим, что для компонент связности задачу следует решать отдельно. А во-вторых заметим для связного графа, что после удаления первой вершины и до конца работы перебора в графе всегда будет присутствовать вершина степени не более 3.

Игра на дереве размера 2^{32}

Семь бед, один ответ

Очередной любитель random_shuffle

Постановка задачи. Есть бинарное дерево, в листьях которого написаны слова `WIN`, `LOSE`. Дерево глубины n. Игра начинается из корня. Двое ходят по очереди. Ход – спуститься влево или вправо. Попав в лист, тот кто в него пришёл, как написано в листе, или проигрывает, или выигрывает. Вопрос – кто выиграет при оптимальной игре обоих? Худший случай для нас – полное бинарное дерево. На нём тривиальное решение работает за $\mathcal{O}(2^n)$.

Вопрос: можно ли решить задачу, не перебирая всё дерево? Оказывается, даже в худшем случае можно. Но не без рандома. Рассмотрим следующий код:

```
int go( int v ) {
  if (v is a leaf) return result[v];
  if (go(left[v]) == LOSE)  return WIN;
  if (go(right[v]) == LOSE) return WIN;
  return LOSE;
}
```

В худшем случае он всегда будет заходить в обе ветки рекурсии, но вдруг ему повезёт? Как сделать так, чтобы ему везло чаще?

```
int go( int v ) {
  if (v is a leaf) return result[v];
  if (rand() & 1) swap(left[v], right[v]); // перебираем детей в случайном порядке
  if (go(left[v]) == LOSE)  return WIN;
  if (go(right[v]) == LOSE) return WIN;
  return LOSE;
}
```

Утверждение: полученный код работает за время $T(n) = 1.687^n$.

Чему равно время работы для выигрышной вершины v?
$W(n) = 0.5 \cdot W(n-1) + L(n-1)$. Здесь 0.5 – вероятность того, что мы сперва пойдём “не в ту сторону”. Чему равно время работы для проигрышной вершины v?
$L(n) = W(n-1) + W(n-1) \Rightarrow W(n) = 0.5 \cdot W(n-1) + 2 \cdot W(n-2) = 1.687^n$.

Как этим пользоваться? Не смотря на то, что алгоритм работает быстро, нам нужно как-то получить, как-то хранить дерево размера 2^n. На самом деле часто дерево игры задано неявно. Например, двое ходят по очереди. Количество ходов ровно n. На выходе у нас строка из 0 и 1, по которой мы за $\mathcal{O}(n)$ можем посчитать результат игры.

Что будет на контесте?

Во-первых, контест учебный. Большая часть задач контеста была разборана на лекции. Если очень хотите, можно даже копипастить код из конспекта... Но зачем? Лучше научитесь сами, а если поймёте, что запутались, подсмотрите в конспекте.

Во-вторых, вас ждёт **16** (шестнадцать) задач. 16 – это много, поэтому напоминаю: не забудьте не полениться и прочесть (и понять!) все условия, а задачи решать в порядке возрастания сложности, а не в случайном (рандом – это хорошо, но не всегда!).

И наконец подсказка к контесту в целом – на контесте нет гробов. Все задачи имеют простое короткое красивое решение. Всем удачи!

КОНЕЦ