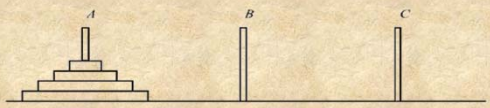
Тема 3

Методы разработки алгоритмов

Метод декомпозиции

Называется также методом «разделяй и властвуй», или методом разбиения, и, возможно, является самым важным и наиболее широко применимым методом проектирования эффективных алгоритмов

Предполагает такую декомпозицию (разбиение) задачи на более мелкие задачи, что на основе решений этих более мелких задач можно легко получить решение исходной задачи



Цель головоломки – перемещать диски (по одному) со стержня на стержень так, чтобы диск большего диаметра никогда не размещался выше диска меньшего диаметра и чтобы в конце концов все диски оказались нанизанными на стержень B

Задачу размера *n* перемещения наименьших дисков со стержня A на стержень B можно представить себе состоящей из двух подзадач размера *n* − *1*

Сначала нужно переместить *n* − 1 наименьших дисков со стержня A на стержень C, оставив на стержне A *n*-й наибольший диск. Затем этот диск нужно переместить с A на B. Потом следует переместить *n* − 1 дисков со стержня C на стержень B
Это перемещение *n* − 1 дисков выполняется путем рекурсивного применения указанного метода

Поскольку диски, участвующие в перемещениях, по размеру меньше тех, которые в перемещении не участвуют, не нужно задумываться над тем, что находится под перемещаемыми дисками на стержнях A, B или C

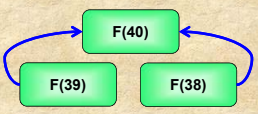
Легкость разработки алгоритмов по методу декомпозиции обусловила популярность этого метода

Во многих случаях эти алгоритмы оказываются более эффективными, чем алгоритмы, разработанные традиционными методами

Динамическое программирование

Суть метода можно объяснить на простом примере чисел Фибоначчи. Вычислить *N* чисел в последовательности Фибоначчи: 1, 1, 2, 3, 5, 8, ... , где первые два члена равны единице, а все остальные представляют собой сумму двух предыдущих, причем *N* меньше 100

```
function F(X:integer): longint;
begin
        if (X = 1) or (X = 2) then F := 1
                              else F := F(X-1) + F(X-2)
end;
```




```
                                  var D: array[1..100] of longint;

function F(X: integer): longint;
begin
  if D[X] = 0 then
          if (X=1) or (X=2) then D[X] := 1
                            else D[X] := F(X-1) + F(X-2);
  F := D[X];
end;
```

Такой подход динамического программирования называется подходом «сверху вниз»

Подход «снизу вверх»

```
D[1] := 1; D[2] := 1;
For i := 3 to N do D[i] := D[i-1] + D[i-2];
```

Раза в три быстрее

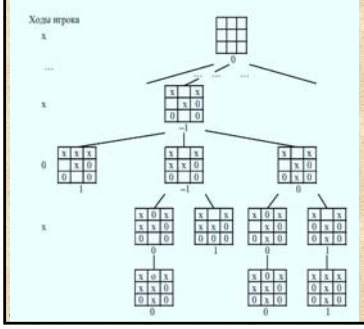
Необходимо решать большее количество подзадач, чем при рекурсии

Часто приходится использовать как промежуточный результат нисходящую форму, а иногда безрекурсивная (итеративная) форма оказывается чрезвычайно сложной и малопонятной

Если алгоритм превышает отведенное ему время на тестах большого объема, то необходимо осуществлять доработку этого алгоритма

Поиск с возвратом

Когда невозможно применить ни один из известных методов, способных помочь отыскать оптимальный вариант решения, остается прибегнуть к полному перебору (иначе – поиск с возвратом)



Рассмотрим игру двух лиц, которые обычно описываются множеством «позиций» и совокупностью правил перехода из одной позиции в другую, причем предполагается, что игроки ходят по очереди

Будем считать, что правилами разрешены лишь конечные последовательности позиций и что в каждой позиции имеется лишь конечное число разрешенных ходов

При таких условиях для каждой позиции *p* найдется число *N(p)* такое, что никакая игра, начавшаяся в *p*, не может продолжаться более *N(p)* ходов

Терминальными называются позиции, из которых нет разрешенных ходов. На каждой из них определена целочисленная функция *f(p)*, задающая выигрыш того из игроков, которому принадлежит ход в этой позиции; выигрыш второго игрока считается равным *–f(p)*

Если из позиции *p* имеется *d* разрешенных ходов $p_1, \dots p_d$, возникает проблема выбора лучшего из них. Будем называть ход **наилучшим**, если по окончании игры он приносит наибольший возможный выигрыш при условии, что противник выбирает ходы, наилучшие для него (в том же смысле)

Пусть *f(p)* есть наибольший выигрыш, достижимый в позиции *p* игроком, которому принадлежит очередь хода, против оптимальной защиты. Так как после хода в позицию p_i выигрыш этого игрока равен $-f(p_i)$, имеем

$$f(p) = \begin{cases} f(p), \text{если } d = 0; \\ max\{-f(p_1), \dots, -f(p_d)\}, \text{если } d > 0 \end{cases}$$

Формула позволяет индуктивно определить *f(p)* для каждой позиции *p*

Функция *f(p)* равна **максимуму**, который гарантирован, если оба игрока действуют оптимально

Однако эта функция отражает результаты осторожной стратегии, которая не всегда хороша против плохих игроков или игроков, действующих по иному принципу оптимальности

Пусть, например, имеются два хода в позиции p_1 и p_2, причем p_1 гарантирует ничью (выигрыш 0) и не дает возможности выиграть, в то время как p_2 дает возможность выиграть, если противник просмотрит очень тонкий выигрывающий ход

В такой ситуации можно предпочесть рискованный ход в p_2, если нет уверенности в том, что противник всемогущ и всезнающ

Следующий алгоритм, называемый поиском с возвратом, вычисляет *f(p)*


```
function BackSearch(p: position): integer;
{оценивает и возвращает выигрыш f(p) для позиции p}
var m,i,t,d: integer;
begin
Определить позиции p1,...,pd, подчиненные p;
        if d = 0 then BackSearch := f(p)
                 else
                        begin
                                m := −∞;
                                for i:= 1 to d do
                                        begin
                                                t := − BackSearch(pi);
                                                if t > m then m:= t;
                                        end;
                                BackSearch := m;
                        end;
end;
```

Здесь +∞ обозначает число, которое не меньше *abs(f(p))* для любой терминальной позиции *p;* поэтому −∞ не больше *f(p)* и *−f(p)* для всех *p*. Этот алгоритм вычисляет *f(p)* на основе «грубой силы» − для каждой позиции он оценивает все возможные продолжения

Для наиболее интересных игр размер дерева является чрезвычайно огромным, порядка W^L, где *W* − среднее количество ходов в позиции, а *L* − количество уровней дерева

Метод ветвей и границ

Идея метода состоит в том, что можно не искать точную оценку хода, про который стало известно, что он не может быть лучше, чем один из ходов, рассмотренных ранее

Пусть в процессе перебора стало известно, что $f(p_1) = -10$

Отсюда заключаем, что $f(p) > 10$, и потому не нужно знать точное значение *f(p)*, если мы каким-либо образом узнали, что $f(p_2) > -10$ (поскольку отсюда следует, что $-f(p_2) \le 10$)

Итак, если p_{21} − допустимый ход из p_2 и $f(p_{21}) \le 10$, можно не исследовать другие ходы из p_2

Говорят, что ход в позицию p_2 «опровергается» (ходом в p_1), если у противника в позиции p_2 есть ответ столь же хороший, как его лучший ответ в позиции p_1

Ясно, что если ход можно опровергнуть, можно не искать наилучшее опровержение

Определим метод «ветвей и границ» как процедуру с двумя параметрами *p* и *bound*, вычисляющую *f '(p, bound)*.

Цель алгоритма − удовлетворить следующим условиям:

f(p, bound) = f(p), если *f(p)* < bound,
f(p, bound) > bound, если *f(p)* ≥ bound

Идею метода ветвей и границ реализует следующий алгоритм:

```
Function B&B(p: position, bound: integer): integer;
{оценивает и возвращает выигрыш F'(p) для позиции p}
label done;
var m,i,t,d: integer;
begin
Определить позиции p1, … ,pd, подчиненные p;
  if d = 0 then B&B := f(p) else
                        begin
                                m := −∞;
                                for i:= 1 to d do
                                begin
                                        t := −B&B(p1, −m);
                                        if t > m then m := t;
                                        if m >= bound then goto done;
                                end;
done:
                                B&B := m;
                        end;
end;
```


Метод альфа-бета отсечения

Метод «ветвей и границ» можно еще улучшить, если ввести не только верхнюю, но и нижнюю границу

Эта идея (ее называют минимаксной альфа-бета процедурой или просто альфа-бета отсечением) является значительным продвижением по сравнению с односторонним методом «ветвей и границ»

Определим процедуру *f"* с параметрами p, alpha и beta (причем всегда будет выполнено alpha < beta), которая удовлетворяет следующим условиям:

f"(p, alpha, beta) ≤ alpha, если *f(p)* < alpha
f"(p, alpha, beta) = *f(p),* если alpha < *f(p)* < beta
f"(p, alpha, beta) ≥ beta, если *f(p)* > beta

```
function AB(p: position; alpha, beta: integer): integer;
{оценивает и возвращает выигрыш F"(p) для позиции p}
label done;
var m,i,t,d: integer;
begin
{Определить позиции p1, ... ,pd, подчиненные p;}
  if d = 0 then AB := f(p)
          else
          begin
                  m := alpha;
                  for i:= 1 to d do
                          begin
                                  t := −AB(p1, −beta, −m);
                                  if t > m then m := t;
                                  if m >= beta then goto done;
                          end;
done:
                  AB := m;
          end;
end;
```

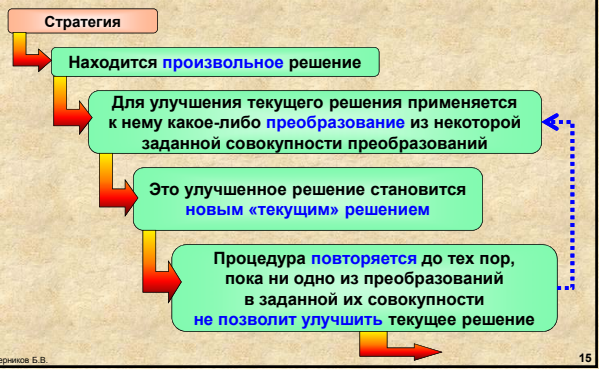

Выгода от альфа-бета отсечения заключается в более раннем выходе из цикла

Эти отсечения полностью безопасны (корректны), потому что они гарантируют, что отсекаемая часть дерева хуже, чем основной вариант

При оптимальных обстоятельствах перебор с альфа-бета отсечением должен просмотреть $W^{(L+1)/2} + W^{L/2}-1$ позицию,
где W – среднее количество ходов в позиции,
L – количество уровней дерева

Это намного меньше, чем перебор с возвратом – данное отсечение позволяет достигать примерно вдвое большей глубины за то же самое время

Локальные и глобальные оптимальные решения



Метод имеет смысл лишь в том случае, когда можно ограничить совокупность преобразований небольшим ее подмножеством,
что дает возможность выполнить все преобразования за относительно короткое время:
если «размер» задачи равняется n,
то можно допустить $O(n^2)$ или $O(n^3)$ преобразований

Если совокупность преобразований невелика, естественно рассматривать решения, которые можно преобразовывать одно в другое за один шаг, как «близкие»

Такие преобразования называются локальными, а соответствующий метод называется локальным поиском

Одной из задач, которую можно решить именно методом локального поиска, является задача нахождения минимального остовного дерева – речь идет о свойстве MST (minimal spanning tree − минимальное остовное дерево)

Остовное дерево – ациклический связный подграф данного связного неориентированного графа, в который входят все его вершины

В принципе, остовное дерево состоит из некоторого подмножества ребер графа, таких, что из любой вершины графа можно попасть в любую другую вершину, двигаясь по этим ребрам, и в нем нет циклов, то есть из любой вершины нельзя попасть в саму себя, не пройдя какое-то ребро дважды

Минимальное остовное дерево (или минимальное покрывающее дерево) в связанном взвешенном неориентированном графе − это остовное дерево этого графа, имеющее минимальный возможный вес, где под весом дерева понимается сумма весов входящих в него ребер

Допустим, есть n городов, которые необходимо соединить дорогами, так, чтобы можно было добраться из любого города в любой другой (напрямую или через другие города)

Разрешается строить дороги между заданными парами городов и известна стоимость строительства каждой такой дороги

Требуется решить, какие именно дороги нужно строить, чтобы минимизировать общую стоимость строительства

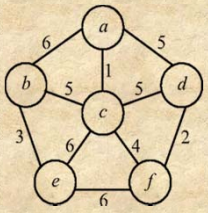
Эта задача может быть сформулирована в терминах теории графов как задача о нахождении минимального остовного дерева в графе:

- вершины графа представляют города
- ребра – это пары городов, между которыми можно проложить прямую дорогу
- вес ребра равен стоимости строительства соответствующей дороги

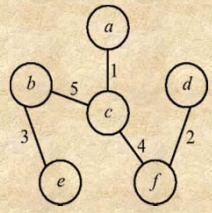
В графе $G = (V, E)$ рассмотрим U – некоторое подмножество V, такое что U и $V \setminus U$ не пусты

Квантор \ означает разность множеств, т.е. $V \setminus U$ означает множество элементов, принадлежащих V, но не принадлежащих U

Пусть (u, v) – ребро наименьшей стоимости, одна вершина которого – $u \in U$, а другая – $v \in V \setminus U$
Тогда существует некоторое MST, содержащее ребро (u, v)



Граф



Минимальное остовное дерево графа

Локальными являются такие преобразования, в ходе которых берется то или иное ребро, не относящееся к текущему остовному дереву, оно добавляется в это дерево (в результате должен получиться цикл), а затем убирается из этого цикла в точности одно ребро (предположительно, ребро с наивысшей стоимостью), чтобы образовалось новое дерево

Время, которое занимает выполнение этого алгоритма на графе из n узлов и e ребер, зависит от количества требующихся улучшений решения

Только проверка факта, что преобразования уже неприменимы, может занять $O(n-e)$ времени, поскольку для этого необходимо перебрать e ребер, а каждое из них может образовать цикл длиной примерно n

Алгоритмы локального поиска проявляют себя с наилучшей стороны как эвристические алгоритмы для решения задач, точные решения которых требуют экспоненциальных затрат времени (относятся к классу *EXPTIME*)

Начать следует с ряда произвольных решений, применяя к каждому из них локальные преобразования до тех пор, пока не будет получено локально-оптимальное решение, т. е. такое, которое не сможет улучшить ни одно преобразование

Видно, что на основе большинства произвольных начальных решений могут получаться разные локально-оптимальные решения

Возможно, одно из таких решений окажется глобально-оптимальным, т. е. лучше любого другого решения

Временная сложность алгоритма. О-символика

Временная сложность алгоритма зависит от количества входных данных

Обычно говорят, что временная сложность алгоритма имеет порядок T(n) от входных данных размера n

Точно определить величину T(n) на практике представляется довольно трудно

Поэтому прибегают к асимптотическим отношениям с использованием О-символики

Если число тактов (действий), необходимое для работы алгоритма, выражается как $11n^2 + 19n - \log n + 3n + 4$, то это алгоритм, для которого T(n) имеет порядок $O(n^2)$
Фактически, из всех слагаемых оставляется только то, которое вносит наибольший вклад при больших n (в этом случае остальными слагаемыми можно пренебречь), и игнорируется коэффициент перед ним

Когда используют обозначение O(), имеют в виду не точное время исполнения, а только его предел сверху, причем с точностью до постоянного множителя

Когда говорят, например, что алгоритму требуется время порядка $O(n^2)$, имеют в виду, что время исполнения задачи растет не быстрее, чем квадрат количества элементов

n	log n	n · log n	n^2
1	0	0	1
16	4	64	256
256	8	2 048	65 536
4 096	12	49 152	16 777 216
65 536	16	1 048 565	4 294 967 296
1 048 476	20	20 969 520	1 099 301 922 576
16 775 616	24	402 614 784	281 421 292 179 456

20 мкс

12 дней

Если операция выполняется за фиксированное число шагов, не зависящее от количества данных, то принято писать O(1)

Основание логарифма здесь не пишется

Пусть есть $O(\log_2 n)$.
Однако $\log_2 n = \log_3 n / \log_3 2$, а $\log_3 2$, как и любую константу, символ O() не учитывает. Поэтому $O(\log_2 n) = O(\log_3 n)$

Время выполнения алгоритма зависит не только от количества входных данных, но и от их значений

Чтобы учитывать этот факт, сохраняя при этом возможность анализировать алгоритмы независимо от данных, различают:

- максимальную сложность $T_{max}(n)$ – сложность наиболее неблагоприятного случая, когда алгоритм работает дольше всего
- среднюю сложность $T_{mid}(n)$ – сложность алгоритма в среднем
- минимальную сложность $T_{min}(n)$ – сложность в наиболее благоприятном случае, когда алгоритм справляется быстрее всего

Базовые принципы оценки временной сложности алгоритма :

- Время выполнения операций присваивания, чтения, записи обычно имеют порядок O(1). Исключение – когда операнды представляют собой массивы или вызовы функций
- Время выполнения последовательности операций совпадает с наибольшим временем выполнения операции в ней (правило сумм – если одна операция имеет порядок *O(f(n))*, а другая − порядок *O(g(n))*, то общее время будет иметь порядок *O(max(f(n), g(n))*)
- Время выполнения конструкции ветвления (if−then−else) состоит из времени вычисления логического выражения (обычно имеет порядок O(1)) и наибольшего из времени, необходимого для выполнения операций, исполняемых при истинном значении логического выражения и при ложном значении логического выражения
- Время выполнения цикла состоит из времени вычисления условия прекращения цикла (обычно имеет порядок O(1)) и произведения количества выполненных итераций цикла на наибольшее возможное время выполнения операций тела цикла
- Время выполнения операции вызова процедур определяется как время выполнения вызываемой процедуры
- При наличии в алгоритме операций безусловного перехода необходимо учитывать изменения последовательности операций, осуществляемых с использованием этих операций безусловного перехода