**Алгоритмические стратегии и методы поиска оптимального решения разработки**

Одним из важнейших этапов в создании компьютерной программы является разработка алгоритма.

Разработано множество алгоритмических стратегий, определены ключевые принципы этих стратегий, что позволяет их применять как универсальный инструмент для широкого класса задач.

Для того, чтобы применить некоторую стратегию в разработке алгоритма, надо знать как ведут себя алгоритмы в разных ситуациях.

- Метод «грубой силы»
- Разделяй и властвуй
- Жадные алгоритмы
- Динамическое программирование
- Поиск с возвратом и его модификации
- Эвристические алгоритмы
- Алгоритмы численных приближений
- Сравнение с образцом

**Метод «грубой» силы**

Алгоритм разрабатывается на основе полного перебора всех возможных вариантов решения.

Вычислительная сложность полного перебора зависит от размера задачи (размерности всех возможных решений задачи). Т.е. время выполнения будет линейно зависеть от размера задачи.

Метод грубой силы используется для многих элементарных, но важных алгоритмических задач, таких как:
- линейный поиск в массиве,
- поиск максимального или минимального значений в списке,
- алгоритмы простых сортировок и т.д.

В качестве альтернативы линейному поиску можно сопоставить алгоритм бинарного поиска, в котором число переборов снижено за счет применения сортировки и метода «разделяй и властвуй».

Рассмотрим еще задачу, которую можно решить методом «грубой силы», а затем применим к ней метод, который снизит количество переборов.

**Возведение числа в неотрицательную степень**: $a^n = a*a*...a$ – метод грубой силы.

Альтернатива этому методу – алгоритм быстрого возведения в степень - использовать битовое представление показателя степени n.

$n=1001_2$ $a^n = (((1* a^1)^2 * a^0)^2 * a^0)^2 * a^1$

- Схема Горнера для n через двоичный код. В связи с этим, для возведения $a^9$ потребуется всего 4 операции умножения вместо 9.

**Умножение матриц**

$C_{i,j} = \sum_{k=1}^{n} A_{i,k} * B_{k,j}$ сложность ($n^3$)

Алгоритмы эффективного умножения квадратных матриц:
алгоритм Штрассена ($O(n^{2,81})$ ) даёт выигрыш на больших плотных матрицах начиная, примерно, от 64×64.

Алгоритм Штрассена является асимптотически не самым быстрым из существующих алгоритмов быстрого умножения матриц, но он проще программируется и эффективнее при умножении матриц относительно малого размера, поэтому именно он чаще используется на практике.

**Выводы**

Методы грубой силы редко дают искусные или эффективные алгоритмы, но они представляют важную стратегию разработки алгоритмов.

1. Он применим к большому числу задач.
2. Для некоторых важных задач дает вполне рациональные алгоритмы.
3. Может быть применим к небольшим по размеру задачам и дать эффективный алгоритм.
4. Может служить мерилом для определения эффективности других алгоритмов.

**Метод декомпозиции – «Разделяй и властвуй»**

Многие полезные алгоритмы имеют рекурсивную структуру.
Такие алгоритмы часто разрабатываются с помощью метода декомпозиции, или разбиения:

• сложная задача разбивается на несколько более простых независимых подзадач, которые подобны исходной задаче, но имеют меньший объем;
Полученные простые задачи представляют новую задачу, которая приведет к получению нового, т.е. не полученного ранее решения.
• далее эти вспомогательные задачи решаются рекурсивным методом,
• после чего, полученные решения комбинируются для получения решения исходной задачи.

**Методы получения оптимального решения из множества возможных вариантов**

Алгоритмы, предназначенные для решения задач оптимизации, обычно представляют собой последовательность шагов, на каждом из которых предоставляется некоторое множество выборов.

Требуется получить одно оптимальное решение из множества возможных решений. Такие задачи требуют в некоторых случаях полного перебора, но это затратное по времени.

**Динамическим программированием (в наиболее общей форме)** называют процесс пошагового решения задач, когда на каждом шаге выбирается одно решение из множества допустимых (на этом шаге) решений, причем такое, которое оптимизирует целевую функцию или функцию ограничения.
В основе динамического программирования лежит принцип оптимизации Беллмана.

В **жадном алгоритме** (greedy algorithm) всегда делается выбор, который кажется самым лучшим в данный момент — т.е. производится локально оптимальный выбор в надежде, что он приведет к оптимальному решению глобальной задачи.

Если в рассматриваемой задаче при поиске оптимального решения нельзя применить ни один из известных методов, который бы позволил отыскать оптимальный вариант решения, то прибегают к последнему средству – полному перебору: метод поиска с возвратом или альфа-бета отсечение.

**Жадный алгоритм**

Такой алгоритм делает на каждом шаге локально оптимальный выбор, - в надежде, что итоговое решение также окажется оптимальным.

Примером служит задача о выдаче сдачи имеющимися купюрами или монетами. Так если в банкомате есть монеты достоинством 25,10,5, 1 копейка и нужно вернуть сдачу 63 копейки.

Мы не задумываясь преобразуем эту величину в две монеты по 25 копеек, одну 10 и три по 1 копейке. Т.е. мы составили самый короткий список монет нужного достоинства.

Т.е. алгоритм, которым мы воспользовались, состоял в выборе сначала монеты самого большого достоинства, но так, чтобы сумма этих монет не превысила 63. Считаем сколько еще осталось выдать ( 38 копеек). Определяем список монет, снова считаем, сколько еще осталось выдать и т.д.

Итак, жадный алгоритм ищет на каждом шаге локальное оптимальное решение. Так если бы надо было сдать сдачи в 15 копеек, а монеты были достоинством 1, 5 и 11, то жадный алгоритм сначала бы выдал 11 копеек, затем 4 по 1 копейке, всего5 монет. Хотя можно было бы выдать 3 монеты по 5 копеек.

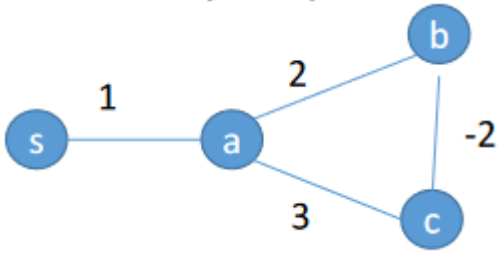
**Алгоритм Дейкстры** является жадным, так как он всегда выбирает вершину ближайшую к источнику, среди тех, кратчайший путь к которым еще не известен. И применяет метод динамического программирования.

**Алгоритм Крускала** – тоже жадный, так как он выбирает ребро из оставшихся ребер, которые не создадут цикл и с минимальной стоимостью.

Но, не каждый жадный алгоритм позволяет получить оптимальное решение в целом. «Жадная стратегия» может обеспечить только сиюминутную выгоду, а в целом результат может оказаться неблагоприятным.

Рассмотрим это на примерах Крускала и Дейкстры на графе с отрицательным весом ребра.

На алгоритме Крускала это никак не отразиться, и можно построить дерево с минимальной стоимостью. А алгоритм Дейкстры, в некоторых случаях, не позволит получить кратчайшие пути правильно.




Жадный алгоритм правильно определяет путь s->a =1.
Кратчайший путь до b от s (или a), по алгоритму получаем путь s->a->b =3.
Но далее получаем кратчайший путь к c через b s->a->b->c длиной 1.
Жадный выбор b вместо c является не оправданным.
Оказывается, что путь s->a->c->b имеет длину лишь 2, поэтому вычисленное расстояние для b является не меньшим.

Но для многих задач такие алгоритмы действительно дают оптимальное решение.

Пример – простая, но не вполне тривиальная задача о выборе заявок.

**Задача о выборе заявок при составлении расписания**

Пусть даны n заявок на проведение занятий в одной и той же аудитории. Два разных занятия не могут перекрываться по времени.
В каждой заявке указаны начало и конец занятия(si и fi для i-й заявки).
Разные заявки могут пересекаться, и тогда можно удовлетворить только одну из них.

Мы отождествляем каждую заявку с промежутком [si , fi),так что конец одного занятия может совпадать с началом другого, и это не считается пересечением.

Формально говоря, заявки с номерами i и j совместны, если интервалы [si , fi) и [sj,fj) не пересекаются (иными словами, если fi ≤sj или fj ≤si).

Задача о выборе заявок состоит в том, чтобы набрать максимальное количество совместных друг с другом заявок.

Жадный алгоритм работает следующим образом.
Мы предполагаем, что заявки упорядочены в порядке возрастания времени окончания: f1 ≤f 2 ≤... ≤fn

Если это не так, то можно отсортировать их за время O(n log n); заявки с одинаковым временем конца располагаем в произвольном порядке. Тогда алгоритм выглядит так (f и s –массивы):

```
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
```

Формируем множество A из номеров выбранных заявок, j –номер последней из них, при этом $f_j = \max \{f_k : k \in A\}$, поскольку заявки отсортированы по возрастанию времени окончания.

Вначале A содержит заявку номер 1, и j=1 (строки 2-3).
Далее (цикл в строках 4-7) ищется заявка, начинающаяся не раньше окончания заявки с номером j.
Если таковая найдена, она включается в множество A и переменной j присваивается ее номер(строки 6-7).

Алгоритм Greedy-Activity-Selector требует всего лишь $\theta(n)$ шагов (не считая предварительной сортировки). Как и подобает «жадному» алгоритму, на каждом шаге он делает выбор так, чтобы остающееся свободным время было максимально.

**Жадный алгоритм задачи Коммивояжера**

**Wiki. Эвристический алгоритм (эвристика)** — алгоритм решения задачи, включающий практический метод, не являющийся гарантированно точным или оптимальным, но достаточный для решения поставленной задачи.

Позволяет ускорить решение задачи в тех случаях, когда точное решение не может быть найдено.

Имеются «жадные» алгоритмы, которые не формируют оптимального алгоритма, но которые с большей вероятностью дают «хорошие» решения.

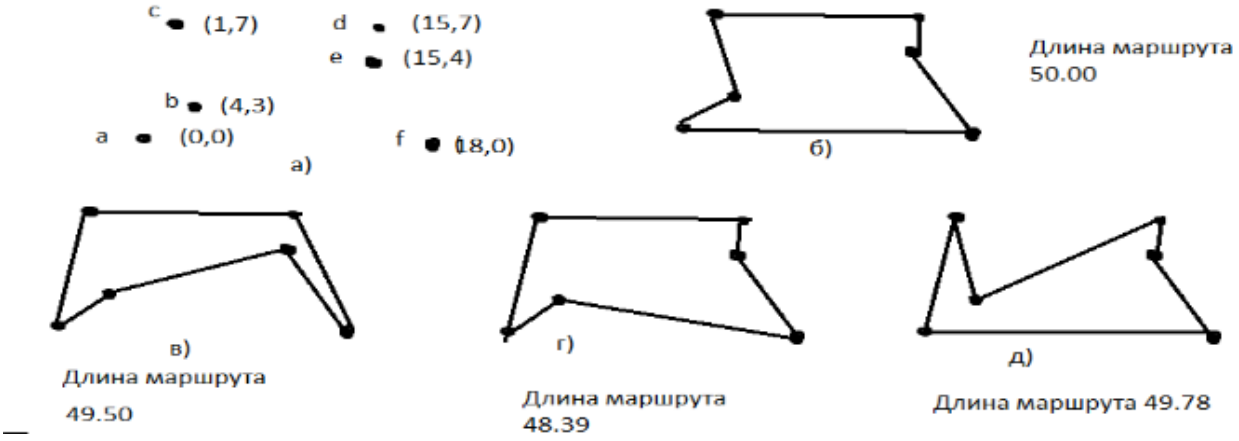
Можно считать вполне удовлетворительным «почти оптимальное решение». В таких случаях метод полного перебора или жадный алгоритм или другой эвристический метод позволят получить хорошее решение, которое может оказаться оптимальным.

**Задача.** Поиск в неориентированном взвешенном графе такого маршрута (простого цикла, включающего все вершины), у которого сумма весов всех ребер будет минимальной. Такой путь называют гамильтоновым циклом. Практическое применение: путь почтальона, обход доски шахматным конем и т.д.

Жадный алгоритм задачи коммивояжера является вариантом алгоритма Крускала.

В «жадном» алгоритме коммивояжера так же рассматриваются сначала минимальные ребра. Причем критерием выбора ребра является то, что это ребро в сочетании с уже выбранными ребрами удовлетворяет условию:
• не приводит к появлению вершины со степенью три и более
• не образует цикла (за исключением случая, когда количество выбранных ребер равно количеству вершин в рассматриваемо задаче).

Совокупность выбранных ребер образуют совокупность несоединенных путей. Такое положение сохраняется до последнего шага, когда единственный оставшийся путь замыкается, образуя маршрут.




Согласно жадному алгоритму
1. Сначала выбирается ребро (d,e)=3
2. Затем рассматриваются ребра: (b,c) (a,b) (e,f) длина каждого ребра равна 5 В каком порядке они будут выбираться значения не имеет –так как все они удовлетворяют условиям выбора. Мы должны выбрать их если используем «жадный алгоритм».
3. Следующее кратчайшее ребро (a,c) длина 7.08, оно может образовать цикл с ребрами (a,b) и (b,c), поэтому принимаем решение отвергнуть его.
4. Ребро (d,f) тоже придется отвергнуть по той же причине.
5. Ребро (b,e) тоже отвергаем, так как оно повышает степень вершин b и e до трех и не образует маршрут с теми ребрами, которые отобраны.
6. Так же отвергается ребро (b,d).
7. Затем выбираем и принимаем ребро (c,d)
Образовался путь: a->b->c->d->f выбирая ребро (a,f) получаем маршрут б).

Он по своей оптимальности находится на 4 месте среди всех возможных маршрутов. Его стоимость всего на 4% больше стоимости оптимального маршрута.

**Динамическое программирование**

Метод на основе таблицы решений подзадач
Известные алгоритмы с решением динамическим программированием
- Разбиение выпуклого многоугольника
- Задача о камнях
- Задача о рюкзаке

Динамическое программирование, как и метод декомпозиции позволяет комбинируя решения вспомогательных задач получать решение.

Динамическое программирование это табличный метод, а не метод создания программного кода.
Если в методе декомпозиции происходило деление задачи на несколько независимых подзадач, каждая из которых решается рекурсивно и потом из решений комбинируется решение задачи.

Динамическое программирование применяется тогда, когда полученные подзадачи не являются независимыми, когда разные вспомогательные подзадачи используют решения одних и тех же подзадач.

В динамическом программировании каждая подзадача решается только один раз, а затем ее решение сохраняется в таблице.
Это позволяет избегать одних и тех же вычислений.

Динамическое программирование применяется в задачах оптимизации. В таких задачах возможно многих вариантов решений.

Каждому варианту решения можно сопоставить некоторое значение и среди них надо найти оптимальное (максимальное или минимальное) значение.

Процесс разработки алгоритмов динамического программирования включает этапы:
1. Описание структуры оптимального решения
2. Рекурсивное определение значения, соответствующего оптимальному решению
3. Вычисление значения оптимального решения, с помощью метода восходящего анализа
4. Составление оптимального решения на основе результатов предыдущих этапов.

Решение задач с большим количеством возможных вариантов решения полным перебором, приводит к превышению времени выполнения.

Однако среди переборных и некоторых других задач можно выделить класс задач, обладающих одним хорошим свойством: имея решения некоторых подзадач (например, для меньшего числа n), можно практически без перебора найти решение исходной задачи.

**Такие задачи решают методом динамического программирования, а под самим динамическим программированием понимают сведение задачи к подзадачам.**

Если не удается разбить задачу на небольшое количество задач, объединение решений которых позволит получить решение исходной задачи как в методе Разделяй и властвуй. То в таких случаях можно попытаться разделить задачу на такое количество подзадач, сколько необходимо, а каждую из полученных вновь также декомпозировать на более мелкие и т.д.

Если алгоритм сводится только к такой последовательности действий, то в результате получим алгоритм с экспоненциальным временем выполнения.

Но часто разбиение задачи на подзадачи приводит к полиномиальному числу подзадач, и тогда некоторые подзадачи приходится решать многократно.

Если будем сохранять решения таких подзадач и при появлении такой задачи снова будем выбирать сохраненное решение, а не выполнять задачу, то получим алгоритм с полиномиальным временем  выполнения.

С точки зрения реализации иногда проще создать таблицу решений всех подзадач, которые придется решать в рамках поставленной задачи. Таблица заполняется решениями независимо от того, нужна ли нам на самом деле конкретная задача для получения общего решения.

**Заполнение таблицы подзадач для получения решения определенной задачи получило название динамического программирования.**

Формы алгоритма динамического программирования могут быть разными, но общей их темой является заполнение таблицы и порядок заполнения ее элементов.

**Последовательности**
Последовательность Фибоначчи Fn задается формулами:
F(1) = 1, F(2) = 1,
F(n) = F(n –1) + F(n –2) при n > 2.
Необходимо найти Fn по номеру n.

Один из способов решения, который может показаться логичным и эффективным, — решение с помощью рекурсии:

```
int F(int n) {
if (n < 2) return 1;
else return F(n - 1) + F(n - 2);
}
```

Но такая, казалось бы, простая программа уже при n = 40 работает заметно долго. Это связано с тем, что одни и те же промежуточные данные вычисляются по несколько раз — число операций нарастает с той же скоростью, с какой растут числа Фибоначчи — экспоненциально

Один из выходов из данной ситуации — сохранение уже найденных промежуточных результатов с целью их повторного использования:

```
int F(int n) {
if (A[n] != -1) return A[n];
if (n < 2) return 1;
else {
A[n] = F(n - 1) + F(n - 2);
return A[n];
}}
```

Приведенное решение является корректным и эффективным.

Но для данной задачи применимо и более простое решение:

```
F[0] = 1;
F[1] = 1;
for (i = 2; i < n; i++) F[i] = F[i - 1] + F[i - 2];
```

Такое решение можно назвать решением «с начала» — мы первым делом заполняем известные значения, затем находим первое неизвестное значение ($F_3$), потом следующее и т.д., пока не дойдем до нужного.

Именно такое решение и является классическим для динамического программирования: мы сначала решили все подзадачи (нашли все Fi для i < n), затем, зная решения подзадач, нашли ответ ($F_n = F_{n-1} + F_{n-2}$, $F_{n-1}$ и $F_{n-2}$ уже найдены).

**Одномерное и к –мерное динамическое программирование**

Пусть исходная задача заключается в нахождении некоторого числа T при исходных данныхn1, n2, ..., nk. То есть мы можем говорить о функции T(n1, n2, ..., nk), значение которой и есть необходимый нам ответ. Тогда подзадачами будем считать задачи T(i1, i2, ..., ik) при i1 < n1, i2 < n2, ..., ik < nk.

Можно говорить об одномерном, двумерном и многомерном динамическом программировании при k = 1, k = 2, k > 2,соответственно.

**Задача.** Дана последовательность целых чисел. Необходимо найти ее самую длинную строго возрастающую подпоследовательность. A{2, 8, 5, 9, 12, 6}

Начнем решать задачу с начала — будем искать ответ, начиная с первых членов данной последовательности.

Для каждого номера i будем искать наибольшую возрастающую подпоследовательность, оканчивающуюся элементом в позиции i.

В массиве L будем записывать длины максимальных подпоследовательностей, оканчивающихся текущим элементом.

Пусть мы нашли все L[i] для1 <= i <= k –1.

Теперь можно найти L[k] следующим образом. Просматриваем все элементы A[i] для 1 <= i < k –1. Если A[i] < A[k], то k-ый элемент может стать продолжением подпоследовательности, окончившейся элементом A[i].

Длина полученной подпоследовательности будет на 1 больше L[i].

Чтобы найти L[k], необходимо перебрать все i от 1 до k –1: L[k] = max(L[i]) + 1, где максимум берется по всем i таким, что A[i] < A[k] и 1 <= i < k.

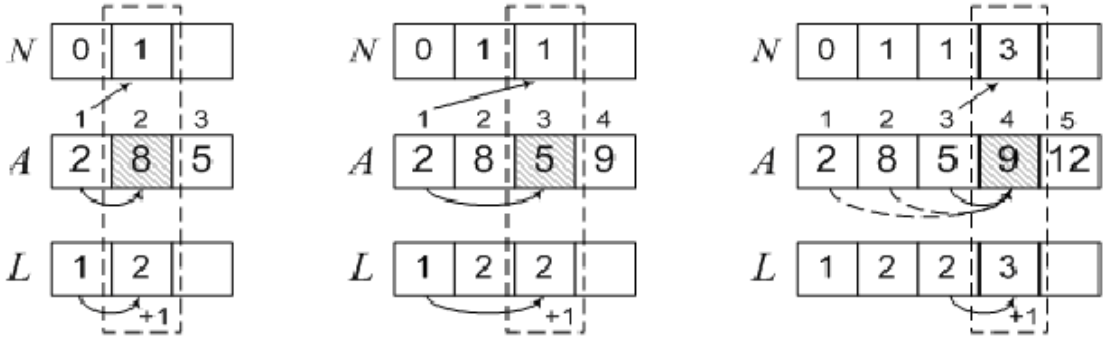
Здесь максимум из пустого множества будем считать равным 0. В этом случае текущий элемент станет единственным в выбранной последовательности, а не будет продолжением одной из предыдущих.

После заполнения массива L длина наибольшей возрастающей подпоследовательности будет равна максимальному элементу L.

Чтобы восстановить саму подпоследовательность, можно для каждого элемента также сохранять номер предыдущего выбранного элемента, например, в массив N.

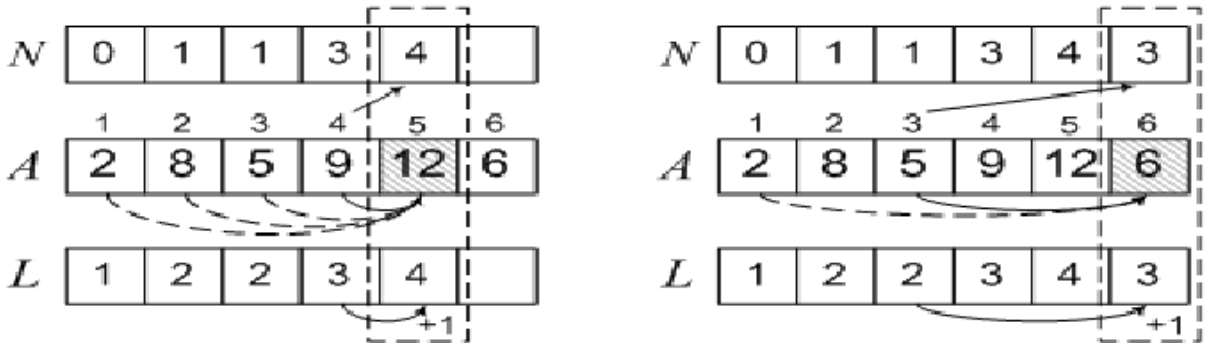
Рассмотрим решение этой задачи на примере последовательности
2, 8, 5, 9, 12, 6.

Поскольку до 2 нет ни одного элемента, то максимальная подпоследовательность содержит только один элемент — L[1] = 1, а перед ним нет ни одного —N[1] = 0.
Далее, 2 < 8, поэтому 8 может стать продолжением последовательности с предыдущим элементом. Тогда L[2] = 2, N[2] = 1.




Меньше A[3] = 5 только элемент A[1] = 2, поэтому 5 может стать продолжением только одной подпоследовательности — той, которая содержит 2.
Тогда L[3] = L[1] + 1 = 2, N[3] = 1, так как 2 стоит в позиции с номером 1.

Аналогично выполняем еще три шага алгоритма и получаем окончательный результат




Теперь выбираем максимальный элемент в массиве L и по массиву N восстанавливаем саму подпоследовательность 2, 5, 9, 12

**Поиск с возвратом**

Поиск оптимального решения путем полного перебора всех возможных решений

Поиск с возвратом (backtracking) - это один из основных приемов поиска решений поставленной задачи.

Каким образом работает поиск с возвратом?

Предположим, для достижения некоторой цели человеку необходимо последовательно принять несколько решений и выполнить некоторые действия в соответствии с принятыми решениями.

Первоначально человек без колебаний и раздумий принимает несколько решений, но при решении очередной проблемы у него возникают сомнения, поскольку возможных решений, предположим, имеется два, и человеку они кажутся одинаково правильными.

Какое-либо из двух решений человек все равно принимает (но запоминает, в какой момент он сомневался, и какое из двух решений все же выбрал) и продолжает свое движение к поставленной цели.

Но, в какой-то момент оказывается, что решение, выбранное из двух, все же оказалось неправильным. Тогда человек вернется в точку принятия неверного решения, и пойдет по альтернативному пути. Не факт, что вновь выбранный путь окажется правильным, но человек попробует все возможные варианты нахождения решения.

**Еще одна аналогия**. Поиск с возвратом можно сравнить с поиском выхода из лабиринта. Нужно войти в лабиринт и на каждой развилке сворачивать влево, до тех пор, пока не найдется выход или тупик. Если впереди оказался тупик, нужно вернуться к последней развилке и свернуть направо, затем снова проверять все левые пути.

В конце концов, выход (если он есть) будет найден.

Общая схема задачи, решаемой путем перебора с возвратом

Даны N упорядоченных множеств U1,U2,...UN (N – известно) и требуется построить вектор решения A=(a1,a2,..aN), где $a_1 \in U_1$, $a_2 \in U_2$, ... , $a_N \in U_N$, удовлетворяющий заданному множеству условий и ограничений.

В алгоритме перебора вектор A строится покомпонентно слева направо. Предположим, найдены значения первых k-1 компонент, A=(a1,a2,..ak-1 ?...?), тогда заданное множество условий ограничивает выбор следующей компоненты ak, некоторым множеством $S_k \subset U_k$.

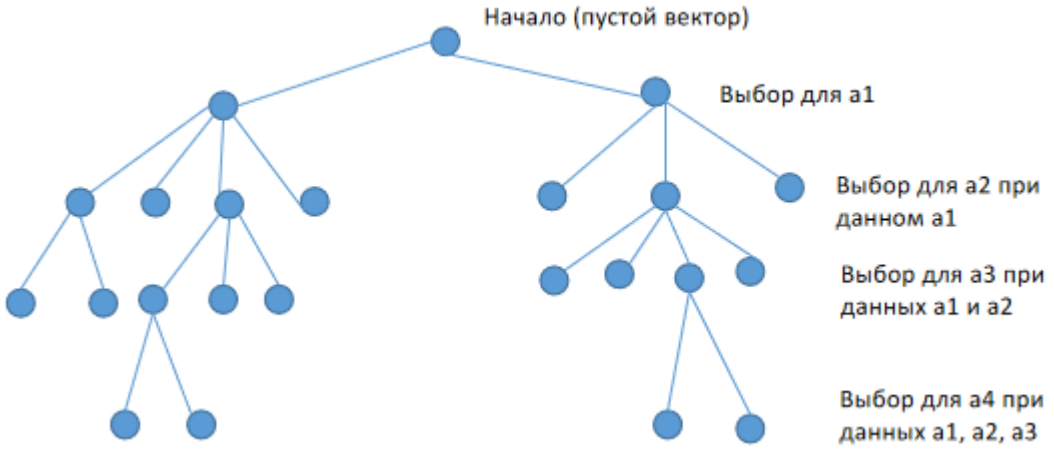
Если Sk<>[](не пустое), то можно выбрать в качестве a1 наименьший элемент Sk и перейти к выбору k+1 компоненты и так далее.

Если Sk=[](пустое), то возвращаемся к выбору k-1 компоненты, отбрасываем $a_{к-1}$ и выбираем в качестве нового $a_{к-1}$ тот элемент Sk-1, который непосредственно следует за только что отброшенным.

Может оказаться, что для нового $a_{к-1}$ множествоSk-1 непустое, тогда предпринимается снова попытка выбрать элемент $a_к$ .

Если невозможно выбрать $a_{к-1}$ , то возвращаемся еще на шаг назад и выбираем новый элемент $a_{к-2}$.

**Дерево поиска решения перебором с возвратом**




Начало – корень дерева – вектор пустой.
Сыновья корня – множество кандидатов на выбор $a_1$.
В общем случае – узлы к-ого уровня – это кандидаты на выбор $a_k$ при условии, что $a_1$, $a_2$,... $a_{k-1}$, выбраны так, как указывают предки этих узлов.

Вопрос, имеет ли задача решение, равносилен вопросу, являются ли какие-либо узлы дерева решениями.
Выполняя поиск решения мы хотим получить все такие узлы.

*Рекурсивная схема реализации алгоритма*

```
Backtrack(<вектор>, i)
If<вектор является решением>
   Вернуть его
Else do
    <вычислить S_i>
    for <a ∈ S_i> do
    Backtrack(<вектор|| a>, i+1>)
    /* || добавление к вектору компоненты */
    od
od
```

Сложность алгоритма – экспоненциальная, т.к. если решение имеет длину N, исследовать требуется порядка $|U_1| \times |U_2| \times \ldots \times |U_N|$ узлов дерева.
Если значение $U_j$ ограничено константой C, то получаем сложность $C^N$ узлов.

**Пример задачи для разбора общей схемы перебора**

Расстановка ферзей: на шахматной доске размером N*N требуется расставить N ферзей, таким образом, чтобы ни один ферзь не атаковал другого.

Способов расстановки ферзей $C_{N^2}^{N}$ (для N=8 это ≈4,4*$10^9$)

Учитываем условия:

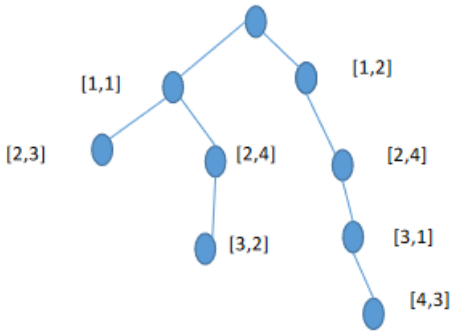
• каждый столбец содержит самое большее одного ферзя, что дает только $N^N$ расстановок (для N=8 это 1.7*$10^7$ )

• никакие два ферзя не могут находиться в одной строке, а поэтому, для того, чтобы вектор (a1,a2,...aN) , был решением, он должен быть перестановкой элементов (1,2, ... N), дает только N! Возможностей (для N=8, 4*$10^4$ ) возможностей.

• никакие два ферзя не могут находиться на одной диагонали, это условие сокращает число переборов еще больше (для N=8 в дереве остается всего 2056 узлов).

Таким образом с помощью ряда наблюдений мы исключили из рассмотрения большое число расстановок ферзей на доске размером N*N.
1) Такой способ сокращения числа переборов называется поиском с ограничениями или отсечением ветвей, так как при этом отсекаются поддеревья из дерева.
2) Другим усовершенствованием является слияние (склеивание ветвей). Идея состоит в том, чтобы избежать выполнения дважды одной и той же работы: если два поддерева данного дерева изоморфны, то мы хотим исследовать только одно из них.

В задаче о ферзях воспользуемся склеиванием. Заметим, что если $a_1 > \lceil N/2 \rceil$, то найденное решение можно отразить и получить решение для $a_1 \leq \lceil N/2 \rceil$. Так как имеем строки и столбцы на доске и диагонали – восходящие (i+j=C) и нисходящие(i-j=C)


Данные рисунки демонстрируют выполнение операций и структуры данных алгоритма.

**Незначительные модификации метода поиска с возвратом, связанные с представлением данных или особенностями реализации, имеют и иные названия: метод ветвей и границ, поиск в глубину, метод проб и ошибок и т. д.**

Поиск с возвратом практически одновременно и независимо был изобретен многими исследователями ещё до его формального описания.

**Задача о лабиринте**
Дано клетчатое поле, часть клеток занята препятствиями. Необходимо попасть из некоторой заданной клетки в другую заданную клетку путем последовательного перемещения по клеткам. Перемещение означает движение в любом направлении.

Классический перебор выполняется по правилам, предложенным в 1891 году Э.Люка:

• в каждой клетке лабиринта выбирается еще не исследованный путь
• если из исследуемой в данный момент клетки нет путей, то возврат на одну клетку назад и пытаемся выбрать другой путь.

С помощью данной схемы находятся все выходы из лабиринта.
Решение этой задачи можно выполнить, применяя обход графа в глубину и создавая Остовное дерево.

```
#include <iostream>
using namespace std;
const int Nmax=5;
int Dx[4] = { 1,0,-1,0 };
int Dy[4] = { 0,1,0,-1 };
int xk = Nmax , yk = Nmax ;
int A[Nmax+2][Nmax+2];
void InitA() {
A[0][1] = -1; A[0][4] = -1; A[1][2] = -1;
A[2][5] = -1; A[3][1] = -1; A[3][3] = -1; A[4][3] = -1;
for (int i = 0; i < Nmax + 2; i++) { A[i][0] = -1;
A[i][Nmax + 1] = -1; }
for (int i = 0; i < Nmax + 2; i++) { A[0][i] = -1;
A[Nmax + 1][i] = -1; }
}
void printA() {
for (int i = 1; i < Nmax + 1; i++) {
for (int j = 1; j < Nmax + 1; j++)
cout << A[i][j] << " ";
cout << endl;
}
}
void Solve(int x, int y, int k) {
//x,y, -координаты клетки, к номер шага
A[x][y] = k;
if (x == xk && y == yk)
printA();
else
for (int i = 0; i < 4; i++) {
//cout << x + Dx[i] << " " << y + Dy[i] << " " << A[x + Dx[i]][y + Dy[i]] << endl;
if (A[x + Dx[i]][y + Dy[i]] == 0)
Solve(x + Dx[i], y + Dy[i], k + 1);
}
A[x][y] = 0;
}
int main()
{
int xn=1, yn=1, xk, yk, N;
InitA();
Solve(xn, yn,1);
printA();
return 0;
}
```

**Метод волны в задаче о лабиринте**

Найти кратчайший по количеству перемещений путь в лабиринте. Начальная клетка в левом верхнем углу, выход правый нижний угол – клетка.

1. Начальную клетку помечаем меткой 1
2. Значение метки увеличивается при каждом шаге на 1. Очередное значение метки записывается в свободные клетки, соседние с клеткой, имеющей предыдущую метку
3. В любой момент множество клеток, не занятых препятствиями, разбивается на три класса:
    1) Помеченные и изученные
    2) Помеченные и неизученные
    3) Непомеченные

Для хранения меток 2) лучше подойдет очередь.
Процесс заканчивается при достижении клетки выхода из лабиринта – есть выход, или невозможности записать очередное значение метки – тупик.

| | | | | | |
|---|---|---|---|---|---|
| **1** | | **9** | **8** | | **10** |
| 2 | 3 | | 7 | 8 | 9 |
| 3 | 4 | 5 | 6 | 7 | |
| 4 | | 6 | | 8 | 9 |
| 5 | 6 | 7 | | 9 | 10 |

**Задача о расписании**

Пусть программисту Васе дано n заданий. У каждого задания известен свой дедлайн, а также его стоимость (то есть если он не выполняет это задание, то он теряет столько-то денег). Вася настолько крут, что за один день может сделать одно задание. Выполнение задания можно начать с момента 0. Нужно максимизировать прибыль.

Классический пример применения жадины: Васе выгодно делать самые «дорогие задания», а наименее дорогие можно и не выполнять — тогда прибыль будет максимальна.
Возникает вопрос: каким образом распределить задания?

Будем перебирать задания в порядке убывания стоимости и заполнять расписание следующим образом:
 если для задания есть еще хотя бы одно свободное место в расписании раньше его дедлайна, то поставим его на самое последнее из таких мест,
 в противном случае в срок мы его не можем выполнить, значит, поставим в конец из свободных мест.

```
//tasks - массив заданий
Arrays.sort(tasks); //сортируем по убыванию стоимости
TreeSet<Integer> time = new TreeSet<Integer>();
for (int i = 1; i <= n; ++i) {
        time.add(i);
}
int ans = 0;
for (int i = 0; i < n; ++i) {
        Integer tmp = time.floor(tasks[i].time);
        if (tmp == null) { // если нет свободного места в расписании, то в конец
        time.remove(time.last());
        } else { //иначе можно выполнить задание, добавляем в расписание
          time.remove(tmp);
          ans += tasks[i].cost;
        }
}
```