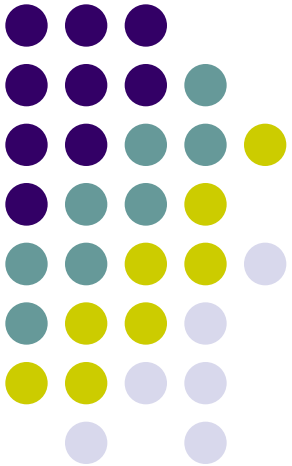
Спортивное программирование

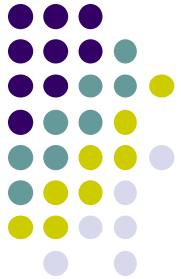
Занятие 3

Динамическое программирование

Шиян В.И.
Кубанский государственный университет
кафедра вычислительных технологий

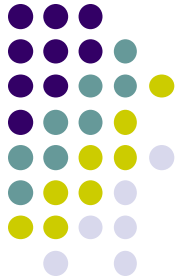


Динамическое программирование



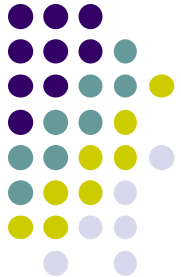
Динамическое программирование – это техника проектирования алгоритмов, которую можно использовать для нахождения оптимальных решений задач и подсчета числа таких решений.

Основные понятия



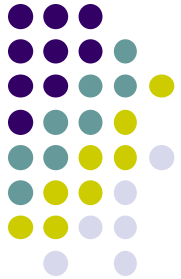
Рассмотрим основные понятия динамического программирования в контексте задачи о размене монет. Сначала посмотрим жадный алгоритм, который не всегда находит оптимальное решение, а затем обсудим, как можно эффективно решить задачу, применив динамическое программирование

Когда жадный алгоритм не работает



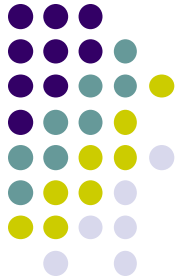
Пусть имеется множество номиналов монет $coins = \{c_1, c_2, \ldots, c_k\}$ и денежная сумма n. Задача заключается в том, чтобы разменять сумму n, использовав как можно меньше монет. Количество монет одного номинала не ограничено. Например, если ***coins = {1, 2, 5}*** и ***n = 12***, то оптимальное решение ***5 + 5 + 2 = 12***, так что достаточно трех монет.

Когда жадный алгоритм не работает



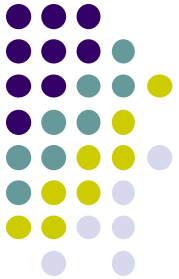
Существует естественный жадный алгоритм решения задачи: всегда выбирать монету максимально возможного номинала, так чтобы общая сумма не превысила *n*. Например, если ***n = 12***, то сначала выбираем две монеты номинала 5, а затем одну монету номинала 2. Стратегия кажется разумной, но всегда ли она оптимальна?

Когда жадный алгоритм не работает



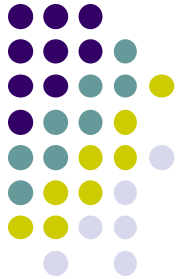
Оказывается, что не всегда. Например, если ***coins = {1, 3, 4}*** и ***n = 6***, то оптимальное решение состоит из двух монет ***(3 + 3 = 6)***, тогда как жадный алгоритм дает решение с тремя монетами ***(4 + 1 + 1 = 6)***. Этот простой контрпример показывает, что жадный алгоритм не корректен.

Когда жадный алгоритм не работает



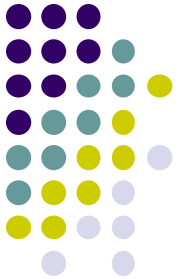
Тогда как же решить задачу? Можно было бы, конечно, попытаться отыскать другой жадный алгоритм, только вот никаких очевидных стратегий не просматривается. Альтернатива – применить алгоритм полного перебора всех возможных способов размена. Такой алгоритм точно даст правильные результаты, но при больших входных данных будет работать очень медленно.

Когда жадный алгоритм не работает



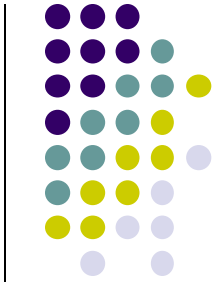
Однако, воспользовавшись динамическим программированием, можно создать алгоритм, который близок к полному перебору, но при этом эффективен. Следовательно, можно применять его к обработке больших входных данных, сохраняя уверенность в правильности результата. Ко всему прочему, ту же технику можно применять к решению многих других задач.

Когда жадный алгоритм не работает



Чтобы воспользоваться динамическим программированием, нужно сформулировать задачу рекурсивно, так чтобы ее решение можно было получить, зная решения меньших подзадач. В задаче о размене монет естественная рекурсивная постановка заключается в вычислении значений следующей функции *solve(x)*: каково минимальное число монет, сумма номиналов которых равна *x*? Очевидно, значение функции зависит от номиналов монет.

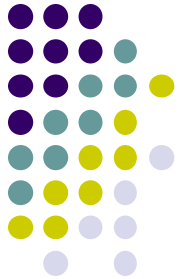
Когда жадный алгоритм не работает



Например, ниже приведены значения функции для небольших *x* в случае, когда ***coins = {1, 3, 4}***:

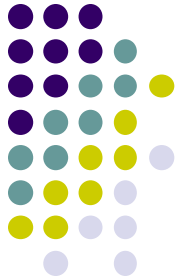
- ***solve(0) = 0***
- ***solve(1) = 1***
- ***solve(2) = 2***
- ***solve(3) = 1***
- ***solve(4) = 1***
- ***solve(5) = 2***
- ***solve(6) = 2***
- ***solve(7) = 2***
- ***solve(8) = 2***
- ***solve(9) = 3***
- ***solve(10) = 3***

Когда жадный алгоритм не работает



Например, ***solve(10) = 3***, потому что для размена суммы 10 нужно, по крайней мере, 3 монеты. Оптимальное решение ***3 + 3 + 4 = 10***.

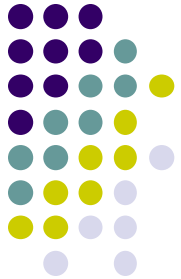
Когда жадный алгоритм не работает



Важное свойство функции *solve* заключается в том, что ее можно вычислить, зная значения для меньших аргументов. Идея в том, чтобы рассмотреть первую монету, выбранную для размена. Так, в примере ранее первой монетой может быть 1, 3 или 4. Если первой выбрать монету 1, то останется решить задачу о размене суммы 9 с помощью минимального количества монет: это задача того же вида, что и исходная, только меньше по размеру. Разумеется, то же рассуждение применимо к монетам 3 и 4. Поэтому можно выписать следующую рекурсивную формулу вычисления минимального количества монет:

solve(x) = min(solve(x - 1) + 1,
solve(x - 3) + 1,
solve(x - 4) + 1).

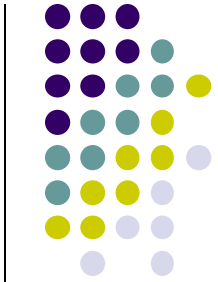
Когда жадный алгоритм не работает



Базой рекурсии является равенство *solve(0) = 0*, поскольку для размена нулевой суммы монеты не нужны. А дальше можно написать, например:

solve(10) = solve(7) + 1 = solve(4) + 2 = solve(0) + 3 = 3.

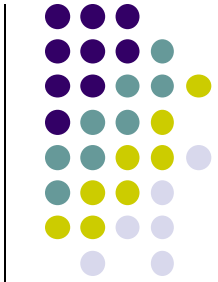
Когда жадный алгоритм не работает



- Теперь можно представить общую рекурсивную функцию, которая вычисляет минимальное количество монет, необходимых для размена суммы x:

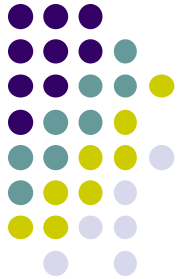
$$solve(x) = \begin{cases} \infty & x<0 \\ 0 & x=0 \\ min_{c \in coins} solve(x-c)+1 & x>0 \end{cases}$$

Когда жадный алгоритм не работает



Прежде всего если $x < 0$, то значение бесконечно, потому что разменять отрицательную сумму невозможно. Далее, если $x = 0$, то значение равно 0, потому для размена нулевой суммы монеты не нужны. Наконец, если $x > 0$, то переменная с пробегает все возможные варианты выбора первой монеты.

Когда жадный алгоритм не работает

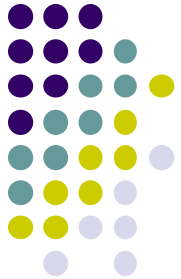


Отыскав рекурсивную функцию, решающую задачу, можно написать реализацию решения на C++ (константа *INF* обозначает бесконечность):

```
int solve(int x) {
    if (x < 0) return INF;
    if (x == 0) return 0;
    int best = INF;
    for (auto c : coins) {
      best = min(best, solve(x - c) + 1);
    }
    return best;
}
```

Однако эта функция неэффективна, потому что сумму можно разменять многими способами, и функция проверяет все. По счастью, это несложно исправить и сделать функцию эффективной.

Когда жадный алгоритм не работает

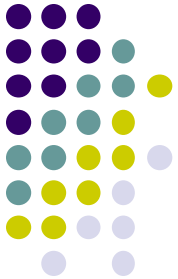


Запоминание. Ключевая идея динамического программирования – ***запоминание***, т. е. сохранение каждого значения функции в массиве сразу после его вычисления. И если впоследствии это значение понадобится снова, то можно достать его из массива, не делая рекурсивных вызовов. Для этого создадим два массива:

```
bool ready[N];
int value[N];
```

где *ready[x]* – признак, показывающий, было ли вычислено значение *solve(x)*, а *value[x]* – само значение, если оно было вычислено. Константа *N* выбирается так, чтобы все необходимые значения уместились в массив.

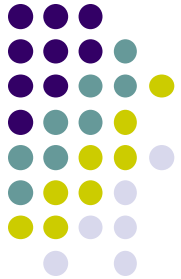
Когда жадный алгоритм не работает



Теперь функцию можно реализовать эффективно:

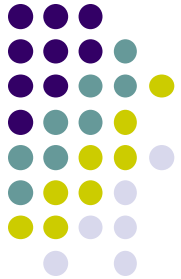
```
int solve(int x) {
    if (x < 0) return INF;
    if (x == 0) return 0;
    if (ready[x]) return value[x];
    int best = INF;
    for (auto c : coins) {
      best = min(best, solve(x - c) + 1);
    }
    ready[x] = true;
    value[x] = best;
    return best;
}
```

Когда жадный алгоритм не работает



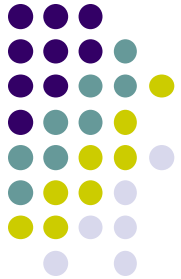
Функция сначала обрабатывает рассмотренные ранее случаи *x < 0* и *x = 0*. Затем она проверяет (глядя на *ready[x]*), сохранено ли значение *solve(x)* в элементе *value[x]*, и если да, то сразу возвращает его. В противном случае значение *solve(x)* вычисляется рекурсивно и сохраняется в *value[x]*.

Когда жадный алгоритм не работает



Эффективность этой функции объясняется тем, что значение для каждого параметра *x* рекурсивно вычисляется только один раз. А после того как значение *solve(x)* сохранено в *value[x]*, его можно легко получить, когда функция снова будет вызвана с параметром *x*. Временная сложность алгоритма равна *O(nk)*, где *n* – подлежащая размену сумма, а *k* – количество номиналов монет.

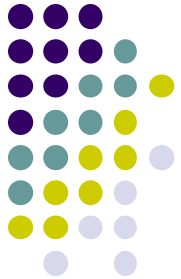
Когда жадный алгоритм не работает



Итеративная реализация. Отметим, что массив *value* можно также заполнить ***итеративно***, воспользовавшись следующим циклом:

```
value[0] = 0;
    for (int x = 1; x <= n; x++) {
        value[x] = INF;
        for (auto c : coins) {
            if (x - c >= 0) {
                value[x] = min(value[x], value[x - c] + 1);
            }
        }
}
```

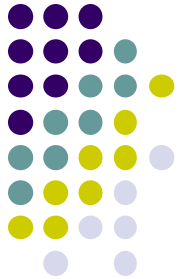
Когда жадный алгоритм не работает



Построение решения. Иногда просят не только найти значение в оптимальном решении, но и привести пример построения самого решения. Чтобы сделать это для задачи о размене монет, объявим новый массив, в котором для каждой разменниваемой суммы будем хранить первую монету в оптимальном решении:

```
int first[N];
```

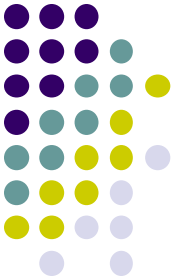
Когда жадный алгоритм не работает



Теперь модифицируем исходный алгоритм следующим образом:

```
value[0] = 0;
for (int x = 1; x <= n; x++) {
    value[x] = INF;
    for (auto c : coins) {
        if (x - c >= 0 && value[x - c] + 1 < value[x]) {
            value[x] = value[x - c] + 1;
            first[x] = c;
        }
    }
}
```

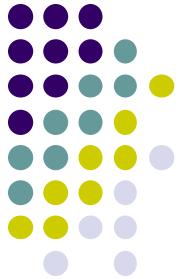
Когда жадный алгоритм не работает



Показанный ниже код печатает монеты, составляющие оптимальное решение для размена суммы *n*:

```
while (n > 0) {
    cout << first[n] << "\n";
    n -= first[n];
}
```

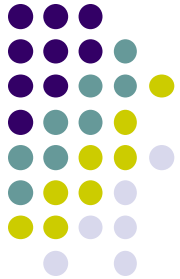
Подсчет решений



Рассмотрим теперь другой вариант задачи о размене монет: требуется найти, сколькими способами можно разменять сумму *x* монетами заданных номиналов. Например, если ***coins = {1, 3, 4}*** и ***x = 5***, то всего есть 6 способов:

- ***1 + 1 + 1 + 1 + 1***;
- ***3 + 1 + 1***;
- ***1 + 1 + 3***;
- ***1 + 4***;
- ***1 + 3 + 1***;
- ***4 + 1***.

Подсчет решений



И эту задачу можно решить рекурсивно. Обозначим *solve(x)* число способов разменять сумму *x*. Например, если ***coins = {1, 3, 4}***, то ***solve(5) = 6***, и рекурсивная формула имеет вид:

solve(x) = solve(x - 1) + solve(x - 3) + solve(x - 4).

Подсчет решений

А в общем виде рекурсивная функция выглядит так:

$$\begin{cases} 0 & x<0 \\ 1 & x=0 \\ \sum_{c \in coins} solve(x-c) & x>0 \end{cases}$$

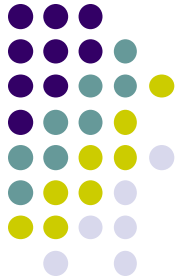
Если $x < 0$, то значение равно нулю, поскольку решений нет. Если $x = 0$, то значение равно 1, поскольку нулевую сумму можно разменять только одним способом. В остальных случаях вычисляем сумму всех значений вида $solve(x - c)$, где c – элемент $coins$.

Подсчет решений

Следующий код заполняет массив *cnt*, в котором *cnt[x]* равно значению *solve(x)* для $0 \le x \le n$:

```
cnt[0] = 1;
for (int x = 1; x <= n; x++) {
    for (auto c : coins) {
        if (x - c >= 0) {
            cnt[x] += cnt[x - c];
        }
    }
}
```

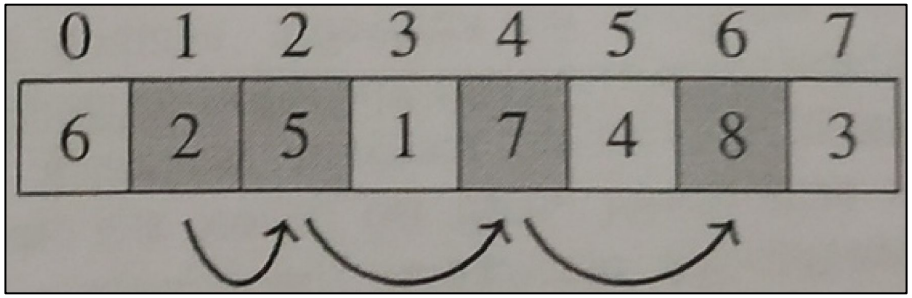
Другие примеры



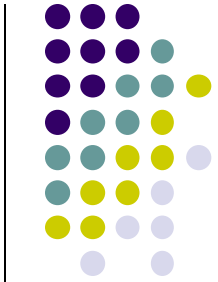
Можно рассмотреть ряд задач, которые эффективно решаются методом динамического программирования. Динамическое программирование – универсальная техника, имеющая много применений в проектировании алгоритмов.

Наибольшая возрастающая подпоследовательность

Наибольшей возрастающей подпоследовательностью в массиве из *n* элементов называется самая длинная последовательность элементов массива, простирающаяся слева направо и такая, что каждый следующий элемент больше предыдущего. На рисунке показана наибольшая возрастающая подпоследовательность в массиве из восьми элементов.



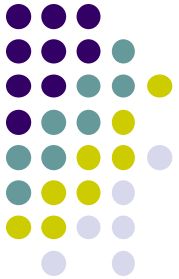
Наибольшая возрастающая подпоследовательность



Для эффективного поиска наибольшей возрастающей подпоследовательности воспользуемся динамическим программированием. Обозначим *length(k)* длину наибольшей возрастающей подпоследовательности, оканчивающейся в позиции *k*. Если суметь вычислить все значения *length(k)* для $0 \le k \le n - 1$, то можно найти и длину наибольшей возрастающей подпоследовательности. Значения этой функции для приведенного массива приведены ниже:

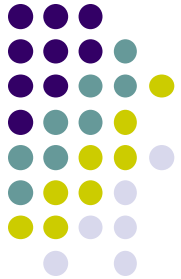
- ***length(0) = 1***
- ***length(1) = 1***
- ***length(2) = 2***
- ***length(3) = 1***
- ***length(4) = 3***
- ***length(5) = 2***
- ***length(6) = 4***
- ***length(7) = 2***

Наибольшая возрастающая подпоследовательность



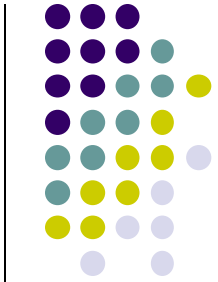
Например, ***length(6) = 4***, поскольку наибольшая возрастающая подпоследовательность, оканчивающаяся в позиции 6, состоит из 4 элементов.

Наибольшая возрастающая подпоследовательность



Чтобы вычислить значение *length(k)*, можно найти позицию *i* < *k*, для которой *array[i] < array[k]* и *length(i)* максимально. Тогда *length(k) = length(i) + 1*, поскольку это оптимальный способ добавить *array[k]* в подпоследовательность. Но если такой позиции *i* не существует, то *length(k) = 1*, т. е. подпоследовательность состоит только из элемента *array[k]*.

Наибольшая возрастающая подпоследовательность

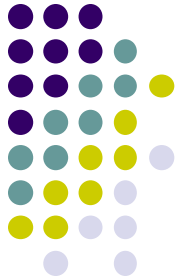


Поскольку значение функции всегда можно вычислить, зная ее значения при меньших аргументах, можно воспользоваться динамическим программированием. В следующем коде значения функции запоминаются в массиве *length*.

```
for (int k = 0; k < n; k++) {
    length[k] = 1;
    for (int i = 0; i < k; i++) {
        if (arr[i] < arr[k]) {
            length[k] = max(length[k], length[i] + 1);
        }
    }
}
```

Понятно, что получившийся алгоритм работаем за время $O(n^2)$.

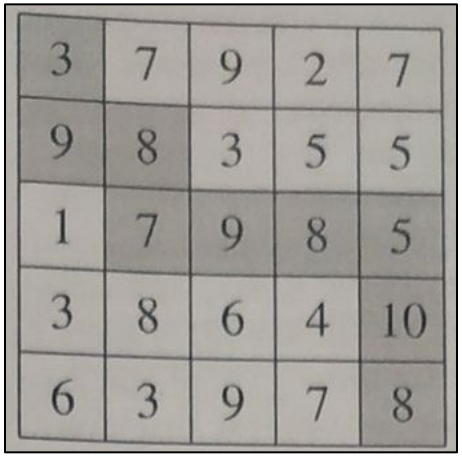
Пути на сетке



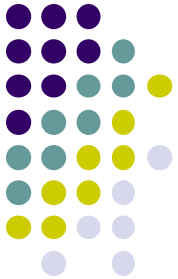
Следующая задача – поиск пути из левого верхнего в правый нижний угол сетки $n \times n$ при условии, что разрешено двигаться только вниз и вправо. В каждой клетке находится целое число, и путь должен быть таким, чтобы сумма значений в лежащих на нем клетках была максимальной.

Пути на сетке

На рисунке показан оптимальный путь на сетке ***5×5***. Сумма значений вдоль пути равна 67, и это наибольшая сумма на путях из левого верхнего в правый нижний угол.



Пути на сетке



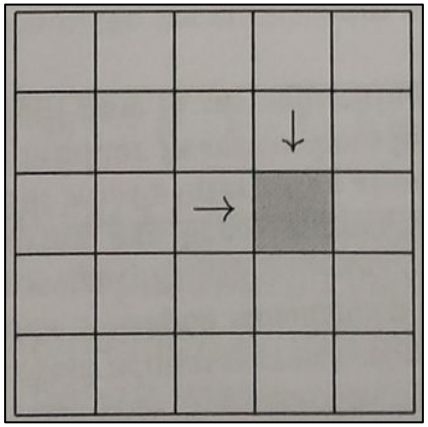
Пронумеруем строки и столбцы сетки числами от *1* до *n*, и пусть *value[y][x]* равно значению в клетке *(y, x)*. Обозначим *sum(y, x)* максимальную сумму на пути из левого верхнего угла в клетку *square(y, x)*. Тогда *sum(n, n)* – максимальная сумма на путях из левого верхнего в правый нижний угол. Так, в приведенном примере сетки ***sum(5, 5) = 67***.

Пути на сетке

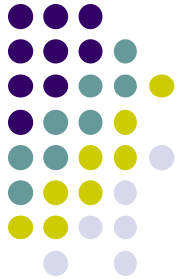
Справедлива формула

$$sum(y, x) = max(sum(y, x - 1), sum(y - 1, x)) + value[y][x],$$

основанная на наблюдении, что путь, заканчивающийся в клетке (y, x), может приходить в нее либо из клетки $(y, x - 1)$, либо из клетки $(y - 1, x)$ (рисунок). Поэтому нужно выбрать направление, доставляющее максимум сумме. Положим $sum(y, x) = 0$, если $y = 0$ или $x = 0$, чтобы рекуррентная формула была справедлива также для клеток, примыкающих к левому и верхнему краю.



Пути на сетке



Поскольку у функции два параметра, массив в методе динамического программирования тоже должен быть двумерным, например:

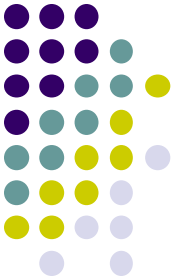
```
int sum[N][N];
```

а суммы вычисляются следующим образом:

```
for (int y = 1; y <= n; y++) {
    for (int x = 1; x <= n; x++) {
      sum[y][x] = max(sum[y][x - 1], sum[y - 1][x]) + value[y][x];
    }
}
```

Временная сложность этого алгоритм равна $O(n^2)$.

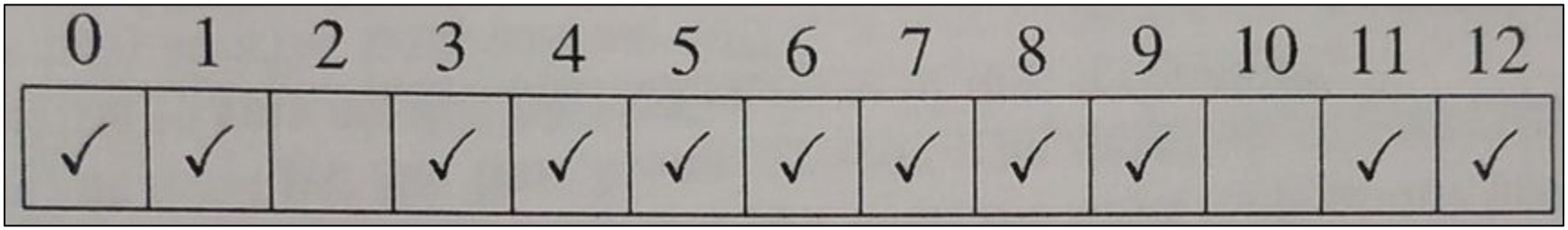
Задачи о рюкзаке



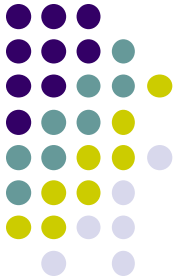
Под задачами о ***рюкзаке*** (или о ранце) понимаются задачи, в которых дано множество предметов и требуется найти подмножества, обладающие некоторыми свойствами. Часто такие задачи можно решить методом динамического программирования.

Задачи о рюкзаке

В этом разделе нас будет интересовать такая задача: пусть дан список весов $[w_1, w_2, \ldots, w_n]$, требуется найти все суммы, которые можно получить сложением весов. На рисунке показаны возможные суммы для весов ***[1, 3, 3, 5]***. В этом случае возможны все суммы от 0 до 12, за исключением 2 и 10. Например, сумма 7 возможна, потому что образована весами ***[1, 3, 3]***.



Задачи о рюкзаке



Для решения задачи рассмотрим подзадачи, в которых для построения сумм используются только первые k весов. Положим $possible(x, k) = true$, если сумму x можно образовать из первых k весов, и $possible(x, k) = false$ в противном случае. Значения функции можно вычислить рекурсивно по формуле $possible(x, k) = possible(x - w_k, k - 1)$ или $possible(x, k - 1)$, основанной на том факте, что вес w_k либо входит в сумму, либо нет. Если вес w_k включается, то остается образовать сумму $x - w_k$, используя только первые $k - 1$ весов, а если не включается, то требуется образовать сумму x, используя первые $k - 1$ весов.

Задачи о рюкзаке

- Базой рекурсии являются случаи

$$possible(x, k) = \begin{cases} true & x=0 \\ false & x \neq 0 \end{cases}$$

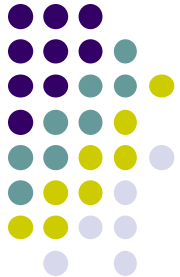
поскольку если веса вообще не используются, то можно образовать только сумму 0. Наконец, значение *possible(x, n)* сообщает, можно ли образовать сумму *x*, используя все веса.

Задачи о рюкзаке

На рисунке показаны все значения функции для весов ***[1, 3, 3, 5]*** (символом ✓ обозначены значения *true*). Например, глядя на строку ***k = 2***, понятно, что суммы ***[0, 1, 3, 4]*** можно образовать из весов ***[1, 3]***.

	0	1	2	3	4	5	6	7	8	9	10	11	12
$k = 0$	✓												
$k = 1$	✓	✓											
$k = 2$	✓	✓		✓	✓								
$k = 3$	✓	✓		✓	✓		✓	✓					
$k = 4$	✓	✓		✓	✓	✓	✓	✓	✓	✓		✓	✓

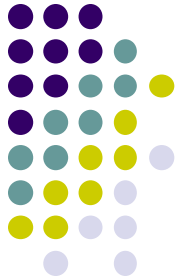
Задачи о рюкзаке



Обозначим *m* полную сумму весов. Показанной выше рекурсивной функции соответствует следующее решение методом динамического программирования:

```
possible[0][0] = true;
for (int k = 1; k <= n; k++) {
    for (int x = 0; x <= m; x++) {
        if (x - w[k] >= 0) {
            possible[x][k] |= possible[x - w[k]][k - 1];
        }
        possible[x][k] |= possible[x][k - 1];
    }
}
```

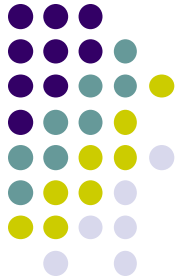
Задачи о рюкзаке



Но есть и более компактный способ реализовать вычисление, применяя всего лишь одномерный массив *possible[x]*, показывающий, можно ли выбрать подмножество весов, дающих в сумме *x*. Хитрость в том, чтобы для каждого нового веса обновлять массив справа налево:

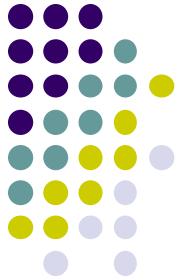
```
possible[0] = true;
for (int k = 1; k <= n; k++) {
    for (int x = m - w[k]; x >= 0; x--) {
     possible[x + w[k]] |= possible[x];
    }
}
```

Задачи о рюкзаке



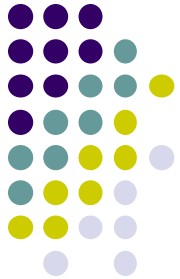
Отметим, что общую идею динамического программирования, представленную ранее, можно применить и к другим задачам о рюкзаке, например в случае, когда даны веса и ценности предметов и требуется найти подмножество с максимальной ценностью, соблюдая при этом ограничение на суммарный вес.

От перестановок к подмножествам



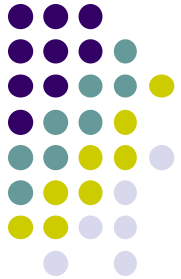
С помощью динамического программирования часто можно заменить итерирование по перестановкам итерированием по подмножествам. Выгода здесь в том, что количество подмножеств 2^n значительно меньше количества перестановок $n!$. Например, при $n = 20$ $n! \approx 2.4 \cdot 10^{18}$, а $2^n \approx 10^6$. Следовательно, для некоторых значений n все подмножества можно обойти эффективно, а все перестановки – нельзя.

От перестановок к подмножествам



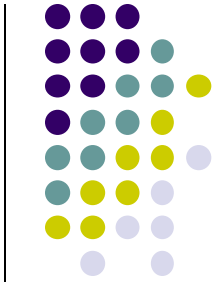
С помощью динамического программирования часто можно заменить итерирование по перестановкам итерированием по подмножествам. Выгода здесь в том, что количество подмножеств 2^n значительно меньше количества перестановок $n!$. Например, при $n = 20$ $n! \approx 2.4 \cdot 10^{18}$, а $2^n \approx 10^6$. Следовательно, для некоторых значений n все подмножества можно обойти эффективно, а все перестановки – нельзя.

От перестановок к подмножествам



В качестве примера рассмотрим следующую задачу: имеется лифт с максимальной грузоподъемностью *x* и *n* человек, желающих подняться с первого на последний этаж. Пассажиры пронумерованы от *0* до *n - 1*, вес *i*-го пассажира равен *weight[i]*. За какое минимальное количество поездок удастся перевезти всех на верхний этаж?

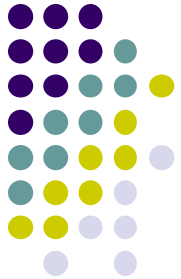
От перестановок к подмножествам



Пусть, например, ***x = 12***, ***n = 5*** и веса таковы:

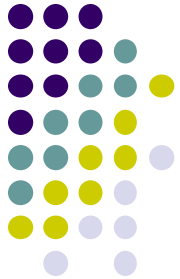
- ***weight[0] = 2***
- ***weight[1] = 3***
- ***weight[2] = 4***
- ***weight[3] = 5***
- ***weight[4] = 9***

От перестановок к подмножествам



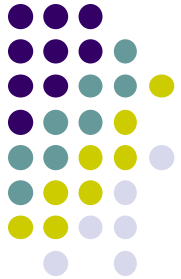
В этом случае минимальное число поездок равно 2. Одно из оптимальных решений выглядит так: сначала перевезти пассажиров 0, 2 и 3 (суммарный вес 11), а затем пассажиров 1 и 4 (суммарный вес 12).

От перестановок к подмножествам



Задачу легко решить за время $O(n!n)$, проверив все возможные перестановки n человек. Но, применив динамическое программирование, можно найти более эффективный алгоритм с временной сложностью $O(2^n n)$. Идея в том, чтобы для каждого подмножества пассажиров вычислить два значения: минимальное число необходимых поездок и минимальный вес пассажиров в последней группе.

От перестановок к подмножествам

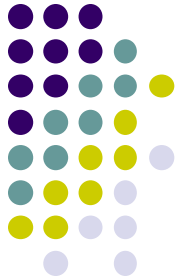


Обозначим *rides(S)* минимальное число поездок для подмножества *S*, а *last(S)* – минимальный вес последней группы в решении с минимальным числом поездок. Так, в примере выше

rides({3, 4}) = 2 и *last({3, 4}) = 5*,

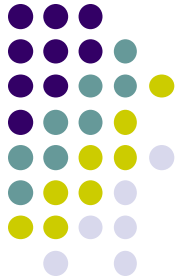
поскольку оптимальный способ поднять пассажиров 3 и 4 на последний этаж – везти их по отдельности, включив пассажира 4 в первую группу, тогда будет минимизирован вес второй группы. Понятно, что наша конечная цель – вычислить значение *rides({0 … n - 1})*.

От перестановок к подмножествам



Можно вычислять значения функций рекурсивно, а затем применить динамическое программирование. Чтобы вычислить значения для подмножества S, нужно перебрать всех пассажиров, принадлежащих S, и произвести оптимальный выбор последнего пассажира p, который входит в лифт. Каждый такой выбор порождает подзадачу с меньшим подмножеством пассажиров. Если $last(S \setminus p) + weight[p] \leq x$, то можно включить p в последнюю группу. В противном случае придется выполнить еще одну поездку специально для p.

От перестановок к подмножествам

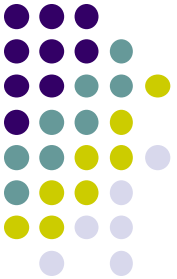


Вычисление по методу динамического программирования удобно реализовать с помощью поразрядных операций. Сначала объявим массив

```
pair<int, int> best[1 << N];
```

в котором для каждого подмножества *S* хранится пара *(rides(S), last(S))*.

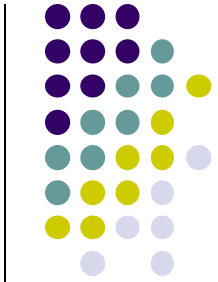
От перестановок к подмножествам



Для пустого подмножества поездки не нужны:

best[0] = { 0, 0 };

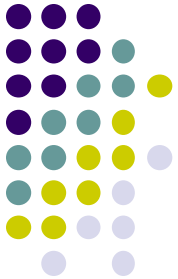
От перестановок к подмножествам



Заполнить массив можно следующим образом:

```
for (int s = 1; s < (1 << n); s++) {
        // начальное значение: необходимо n+1 поездок
        best[s] = { n + 1, 0 };
        for (int p = 0; p < n; p++) {
                if (s & (1 << p)) {
                        auto option = best[s ^ (1 << p)];
                        if (option.second + weight[p] <= x) {
                                // добавить p в существующую группу пассажиров
                                option.second += weight[p];
                        }
                        else {
                                // предусмотреть для p отдельную поездку
                                option.first++;
                                option.second = weight[p];
                        }
                        best[s] = min(best[s], option);
                }
        }
}
```

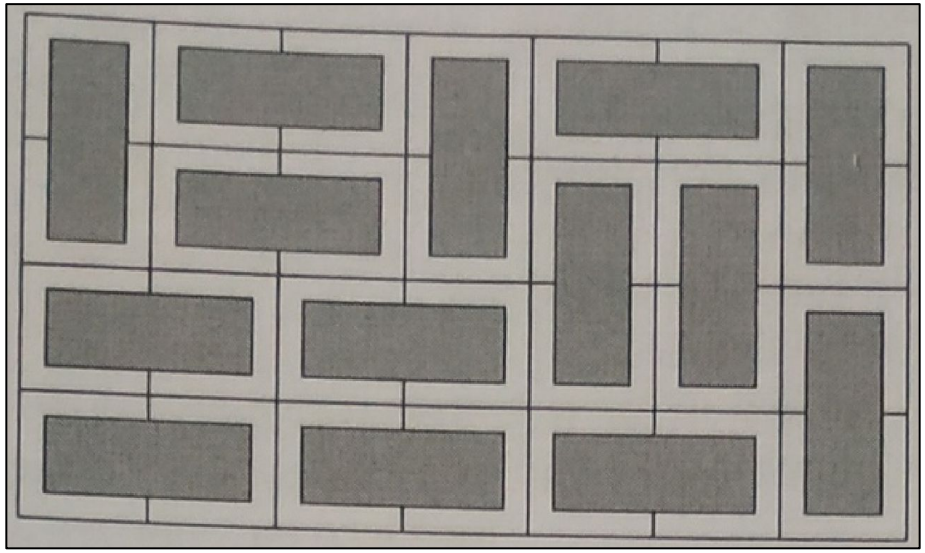
От перестановок к подмножествам



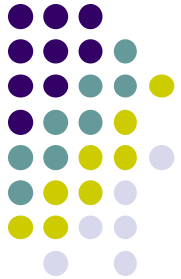
Отметим, что этот цикл обладает следующим свойством: для любых двух подмножеств S_1 и S_2 – таких, что $S_1 \subset S_2$, S_1, – обрабатывается раньше S_2. Следовательно, используемые в динамическом программировании значения вычисляются в правильном порядке.

Подсчет количества замощений

Иногда состояния в решении методом динамического программирования оказываются сложнее, чем фиксированные комбинации значений. В качестве примера рассмотрим задачу о нахождении количества различных способов замостить сетку размера $n \times m$ плитками размера ***1×2*** и ***2×1***. Например, существует 781 способ замостить сетку ***4×7***, один из них показан на рисунке.



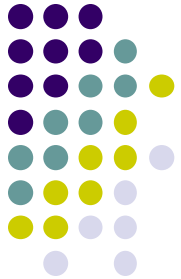
Подсчет количества замощений



Задачу можно решить методом динамического программирования, рассматривая сетку ряд за рядом. Каждый ряд в решении можно представить строкой, содержащей m символов из множества $\{\sqcap, \sqcup, \sqsubset, \sqsupset\}$. Так, решение, показанное на рисунке, состоит из четырех рядов, которым соответствуют такие строки:

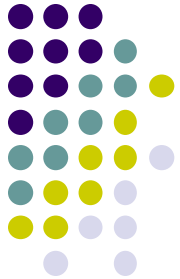
- $\sqcap\sqsubset\sqsupset\sqcap\sqsubset\sqsupset\sqcap$
- $\sqcup\sqsubset\sqsupset\sqcup\sqcap\sqcap\sqcup$
- $\sqsubset\sqsupset\sqsubset\sqsupset\sqcup\sqcup\sqcap$
- $\sqsubset\sqsupset\sqsubset\sqsupset\sqsubset\sqsupset\sqcup$

Подсчет количества замощений



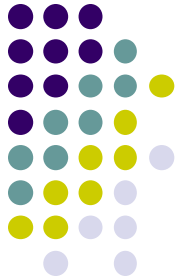
Пронумеруем ряды сетки числами от *1* до *n*. Обозначим *count(k, x)* число таких решений для рядов *1…k*, что ряду *k* соответствует строка *x*. Здесь можно воспользоваться динамическим программированием, потому что состояние ряда ограничено только состоянием предыдущего ряда.

Подсчет количества замощений



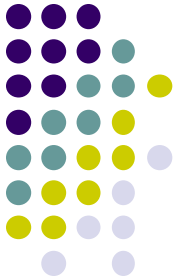
Решение допустимо, если ряд *1* не содержит символа ⌴, ряд *n* не содержит символа ⊓ и все соседние ряды совместимы. Например, ряды ⌴□□⌴⊓⊓⌴ и □□□⌴⌴⊓ совместимы, а ряды ⊓□⊓□□⊓ и □□□□□⌴ – нет.

Подсчет количества замощений



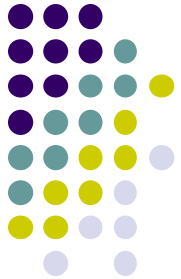
Поскольку ряд состоит из m символов и каждый символ можно выбрать четырьмя способами, общее число различных рядов не превышает 4^m. Можно перебрать $O(4^m)$ возможных состояний каждого ряда, и для каждого ряда существует $O(4^m)$ возможных состояний предыдущего ряда, поэтому временная сложность решения равна $O(n4^{2m})$. На практике разумно повернуть сетку, так чтобы более короткая сторона имела длину m, поскольку множитель 4^{2m} доминирует в оценке сложности.

Подсчет количества замощений



Эффективность решения можно повысить, представив ряды в более компактной форме. Оказывается, что достаточно знать, какие колонки предыдущей строки содержат верхний квадратик вертикальной плитки. Поэтому для представления ряда достаточно символов ⊓ и □, где □ – комбинация символов ⊔, ⊏ и ⊐. При таком представлении существует всего 2^m различных строк, так что временная сложность равна $O(n2^{2m})$.

Подсчет количества замощений

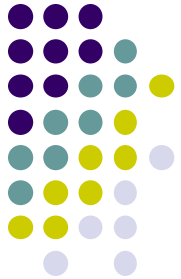


- Отметим, что существует явная формула для количества замощений:

$$\prod_{a=1}^{\lceil n/2 \rceil} \prod_{b=1}^{\lceil m/2 \rceil} 4 \cdot \left(\cos^2 \frac{\pi a}{n+1} + \cos^2 \frac{\pi b}{m+1} \right).$$

Эта формула очень эффективна, поскольку вычисляет количество замощений за время $O(nm)$, но, так как ответ выражается как произведение вещественных чисел, возникает проблема: как точно представлять промежуточные результаты.

Задания 1-3



https://codeforces.com/problemset/problem/189/A

https://codeforces.com/problemset/problem/1343/C

https://codeforces.com/problemset/problem/368/B