

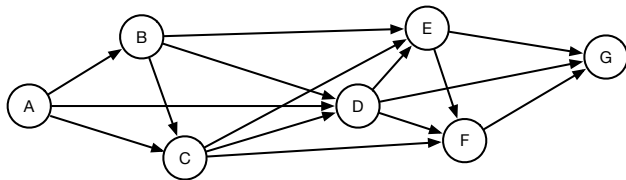
Алгоритмы и структуры данных

Лекция 11

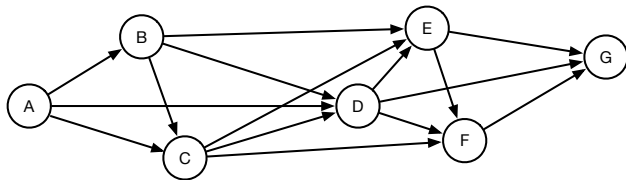
Динамическое программирование-1.

Сергей Леонидович Бабичев

Задача 1. Найти количество маршрутов из пункта A в пункт G .



Задача 2. Найти количество маршрутов из пункта A в пункт G .



Решение.

Обозначим число маршрутов из A до i как $F(i)$. Тогда:

- $F(G) = F(F) + F(E) + F(D)$, так как доехать в G можно из F , E , и D общее число есть сумма способов.
- $F(F) = F(E) + F(D) + F(C)$
- $F(E) = F(D) + F(C) + F(B)$
- $F(D) = F(C) + F(B) + F(A)$
- $F(C) = F(B) + F(A)$
- $F(B) = F(A)$ и $F(A) = 1$

Ответим на несколько вопросов:

- Разбивается ли задача на подзадачи? Да. Решение $F(G)$ требует решения $F(F)$, $F(E)$, $F(D)$ и т.д.
- Решаются ли подзадачи тем же методом, что и основная? Да. Вычисление $F(F), \dots$ аналогично.
- Достаточно ли для решения задачи иметь только решения подзадач и потом вычислить результат? Да. Нужно сложить результаты.
- Каждая из подзадач проще основной? Да. Но немного. Каждая подзадача на единицу ближе ко входу, чем задача.
- Имеются ли подзадачи, решаемые без рекурсии? Да. $F(A)$.
- Совпадают ли часть подзадач в решении задачи? Да. Одни и те же подзадачи повторяются в списке по несколько раз.

Тогда это — *задача динамического программирования*.

Отличие задач ДП от *разделяй и властвуй*

Мы изучили сортировку слиянием. Задача ли это динамического программирования или задача *разделяй и властвуй*? Ответим на те же вопросы.

- Разбивается ли задача на подзадачи? Да. Массив разбивается на два подмассива.
- Решаются ли подзадачи тем же методом, что и основная? Да. Подмассивы можно сортировать тем же методом.
- Достаточно ли для решения задачи иметь только решения подзадач и потом вычислить результат? Да. Нужно слить два подмассива.
- Каждая из подзадач проще основной? Да. **И намного. Каждая подзадача в два раза меньше основной.**
- Имеются ли подзадачи, решаемые без рекурсии? Да. Когда размер массива меньше порогового, мы используем простую сортировку.
- Совпадают ли часть подзадач в решении задачи? **Нет. Каждая подзадача — уникальна и они не пересекаются совсем.**

Отличие задач ДП от *разделяй и властвуй*

- Эти два пункта отличия принципиальны.
- Если подзадача намного проще основной задачи — нет смысла сохранять её решение для последующего использования.
- Если подзадачи не совпадают — нет смысла сохранять решения подзадач для последующего использования.
- Если же подзадача *немного* проще задачи и эти подзадачи при решении частично совпадают — имеет смысл сохранять решения подзадач и не решать их повторно.

Задача 3. Функция задана рекуррентной последовательностью:

$$f(n) = \begin{cases} 0, & \text{если } n = 0, \\ 1, & \text{если } n = 1, \\ f(n-1) + f(n-2), & \text{если } n > 1. \end{cases}$$

Найти $f(5)$.

Рекуррентность и рекурсия

Рекуррентная форма \rightarrow рекурсивный алгоритм

```
int f(int n) {  
    if (n == 0) return 0;  
    if (n == 1) return 1;  
    return f(n-1) + f(n-2);  
}
```


Рекуррентность и рекурсия

Рекуррентная форма \rightarrow рекурсивный алгоритм

```
int f(int n) {  
    if (n == 0) return 0;  
    if (n == 1) return 1;  
    return f(n-1) + f(n-2);  
}
```

Три вопроса:

- 1 Корректен ли он? — Да, так как это просто другая запись.
- 2 Как оценить его сложность?
- 3 Как его ускорить?

Оценка сложности

Второй вопрос — сложность алгоритма.

Добавим в программу одну строку — *трассировку исполнения*.

```
int f(int n) {  
    printf("f(%d)\n", n);  
    if (n == 0) return 0;  
    if (n == 1) return 1;  
    return f(n-1) + f(n-2);  
}
```

Оценка сложности: прогон программы.

```
$ ./a.out
```

```
f(5)
```

```
f(4)
```

```
f(3)
```

```
f(2)
```

```
f(1)
```

```
f(0)
```

```
f(1)
```

```
f(2)
```

```
f(1)
```

```
f(0)
```

```
f(3)
```

```
f(2)
```

```
f(1)
```

```
f(0)
```

```
f(1)
```

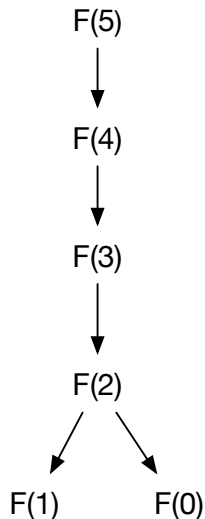
```
$
```

Как ускорить?

- Почему так медленно?
- Мы много раз повторно вычисляем значение функции от одних и тех же аргументов.
- Третий вопрос: можно ли уменьшить сложность алгоритма по времени, то есть ускорить алгоритм?
- Да. Надо просто запоминать результаты вычислений.
- Если мы уже вызывали функцию от этого аргумента, нужно это проверить и вернуть нужный результат.

```
int f(int n) {  
    enum {MAXN = 100};  
    static int c[MAXN];  
    if (n == 0) return 0;  
    if (n == 1) return 1;  
    if (c[n] > 0) return c[n];  
    return c[n] = f(n-1) + f(n-2);  
}
```

Дерево вызовов модифицированной функции для $n = 5$



Решение задачи о количестве маршрутов

- Абсолютно каждая задача на динамическое программирование может быть представлена в рекурсивном виде.
- Этот рекурсивный вид называется «уравнением Беллмана» задачи.
- Ричард Беллман — основатель раздела математики «динамическое программирование».

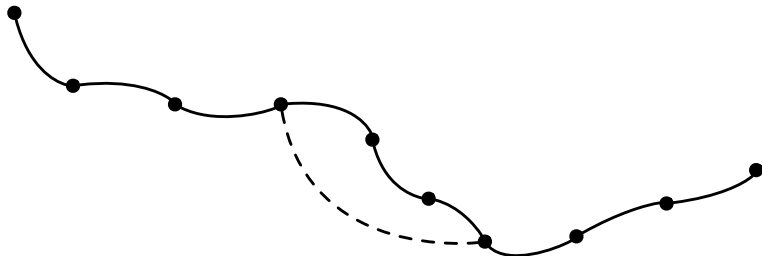
Принцип Беллмана

- Беллман, 1940-е годы, принцип оптимальности:
«Каковы бы ни были первоначальное состояние и решение, последующие решения должны составлять оптимальное поведение относительно уже решённого состояния.»
- Арис:
«Если вы не научились использовать наилучшим образом то, что у вас есть, вы не научитесь использовать то, что у вас будет»

Уравнение Беллмана

- Имеется управляемая система и S — её текущее состояние.
- $W_i = f_i(S, x_i)$ — функция выигрыша (стоимости) при использовании управления x на i -м шаге.
- $S' = \varphi_i(S, x_i)$ — состояние, в которое переходит система при воздействии x . Независимо от значения S нужно выбрать управление на этом шаге так, чтобы выигрыш на данном шаге плюс оптимальный выигрыш на всех последующих шагах был максимальным.

$$W_i(S) = \max_{x_i} \{f_i(S, x_i) + W_{i+1}(\varphi_i(S, x_i))\}$$



Уравнение Беллмана для задачи о количестве маршрутов

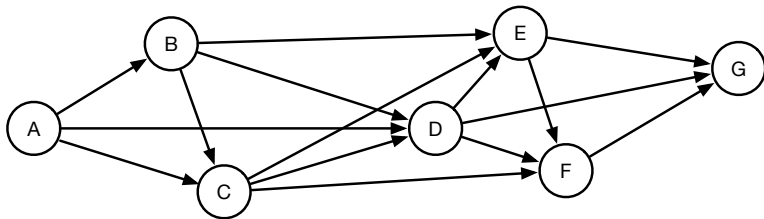
Пусть A — матрица смежности,

$$A[i, j] = \begin{cases} 1, & \text{если имеется прямой путь из } i \text{ в } j \\ 0, & \text{если не имеется прямого пути из } i \text{ в } j. \end{cases}$$

Тогда:

$$F(x) = \sum_{i=1}^k F(i) \cdot A[i, x]$$

Дорожный граф и его матрица смежности.



$$A = \begin{matrix} & \begin{matrix} A & B & C & D & E & F & G \end{matrix} \\ \begin{matrix} A \\ B \\ C \\ D \\ E \\ F \\ G \end{matrix} & \begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{matrix}$$

Задача о возрастающей
подпоследовательности наибольшей
длины.

Задача о возрастающей подпоследовательности наибольшей длины

- Имеется последовательность чисел a_1, a_2, \dots, a_n .
- Подпоследовательность $a_{i_1}, a_{i_2}, \dots, a_{i_k}$ называется возрастающей, если

$$1 \leq i_1 < i_2 < \dots < i_k \leq n$$

и

$$a_{i_1} < a_{i_2} < \dots < a_{i_k}$$

- Требуется найти максимальную длину возрастающей подпоследовательности.

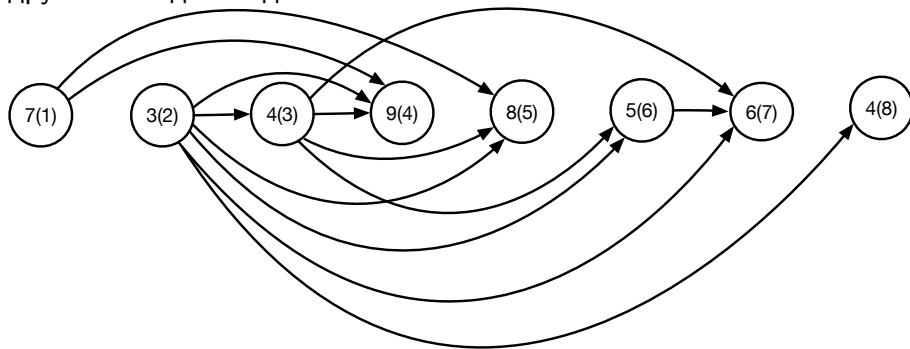
Пример: $a_i = \{7, 3, 4, 9, 8, 5, 6, 4\}$.

Одна из возрастающих подпоследовательностей есть $\{7, 9\}$.

$\{3, 4, 5, 6\}$ есть возрастающая подпоследовательность наибольшей длины.

Задача о возрастающей подпоследовательности

- Соединим направленными рёбрами элементы, которые могут быть друг за другом в подпоследовательности.



- Задача — найти не количество путей, а длину наибольшего пути.

Задача о возрастающей последовательности

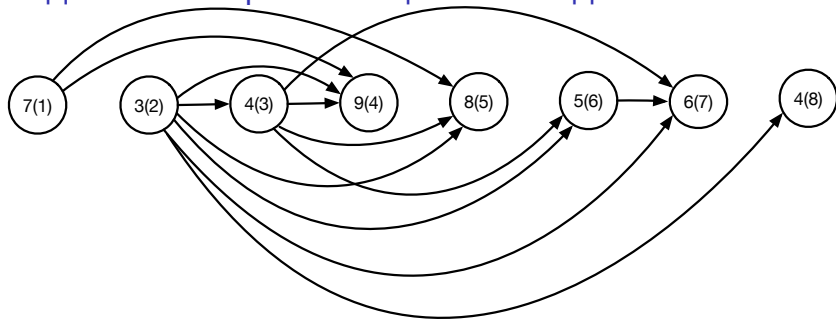
- В задаче на количество путей консолидация подзадач имела вид

$$R_i = \sum_{i=1}^{N_{R_{i-1}}} R_{i-1}$$

- В этой задаче длина наибольшего пути к вершине i есть максимум из наибольших путей к предыдущим вершинам плюс единица

$$L_i = 1 + \max_{j=1}^{N_{L_{i-1}}} (L_{i-1,j})$$

Задача о возрастающей последовательности



$$L_8 = 1 + \max(L_2)$$

$$L_7 = 1 + \max(L_2, L_3, L_6)$$

$$L_6 = 1 + \max(L_2, L_3)$$

$$L_5 = 1 + \max(L_1, L_2, L_3)$$

...

$$L_1 = 1$$

Задача о возрастающей последовательности

- Рекурсивно эта задача решается следующей программой:

```
int f(int a[], int N, int k) { // k - # of element
    int m = 0;
    for (int i = 0; i < k-1; i++) { // for lefts
        if (a[i] < a[k]) { // Is there a path?
            int p = f(a,N,i); // It's length
            if (p > m) m = p; // m = max(m,p)
        }
    }
    return m+1;
}
```


Задача о возрастающей последовательности

- Для последовательности $\{1, 4, 2, 5, 3\}$ решение будет таким:

- ▶ $f_5 = 1 + \max(f_1, f_3)$
- ▶ $f_4 = 1 + \max(f_1, f_2, f_3)$
- ▶ $f_3 = 1 + \max(f_1)$
- ▶ $f_2 = 1 + \max(f_1)$
- ▶ $f_1 = 1$
- ▶ Возвращаемся назад
- ▶ $f_2 = 1 + \max(f_1) = 2$
- ▶ $f_3 = 1 + \max(f_1) = 2$
- ▶ $f_4 = 1 + \max(f_1, f_2, f_3) = 3$
- ▶ $f_5 = 1 + \max(f_1, f_3) = 3$

Решение есть $\max(f_1, f_2, f_3, f_4, f_5) = 3$

Задача о возрастающей последовательности

- Чтобы повторно не решать решённые подзадачи вводим массив c размером N , хранящий значения вычисленных функций. Начальные значения его равны нулю.

```
int f(int *a, int N, int k, int *c) {
    if (c[k] != 0) return c[k]; // Already solved
    int m = 0;
    for (int i = 0; i < k; i++) {
        if (a[i] < a[k]) {
            int p = f(a,N,i,c);
            if (p > m) m = p; // m = max(m,p)
        }
    }
    return c[k] = m+1; // Put to cache and return
}
```

Рекурсия как база динамического программирования.

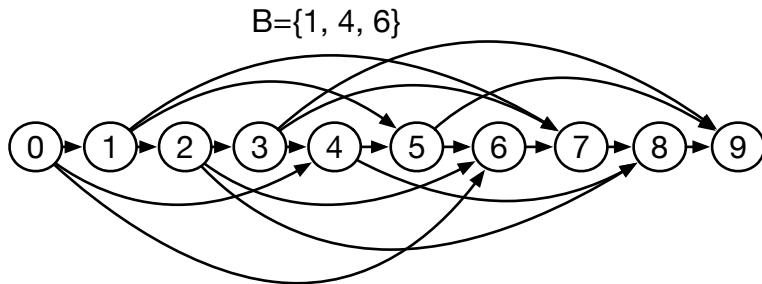
Рекурсия как база динамического программирования

- Пусть в банкомате имеется неограниченное количество банкнот (b_1, b_2, \dots, b_n) заданных номиналов.
- Задача заключается в том, чтобы выдать требуемую сумму денег наименьшим количеством банкнот.

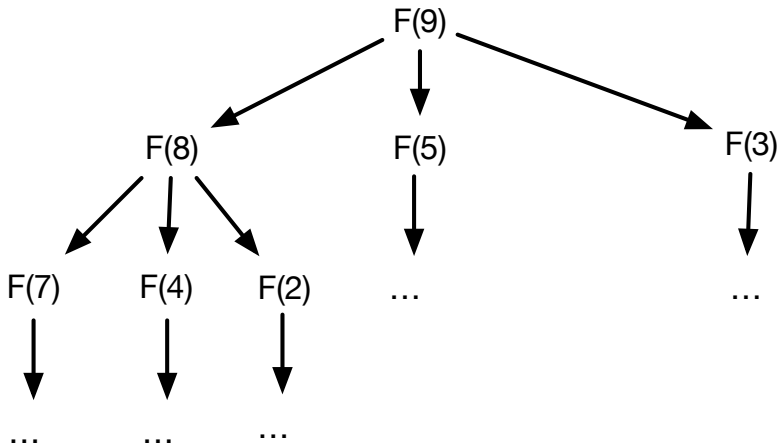
Задача о банкомате

- Жадное решение имеется только для некоторого набора b_i .
- Например, при $b = \{1, 6, 10\}$ и $x = 12$ жадное решение даст ответ 3 ($10 + 1 + 1$), хотя существует более оптимальное решение 2 ($6 + 6$).
- Для набора $b = \{6, 10\}$ жадный алгоритм решения не найдёт.

Задача о банкомате: граф



Задача о банкомате: дерево рекурсии



Задача о банкомате: уравнение Беллмана

$$f(x) = \begin{cases} \min_{i=1,n} \{f(x - b_i)\} + 1, & \text{если } x > 0 \\ 0, & \text{если } x = 0 \\ \infty, & \text{если } x < 0 \end{cases}$$

Задача о банкомате: простая рекурсия

$$f(x) = \begin{cases} \min_{i=1,n} \{f(x - b_i)\} + 1, & \text{если } x > 0 \\ 0, & \text{если } x = 0 \\ \infty, & \text{если } x < 0 \end{cases}$$

```
const int GOOGOL = 999999999;
int f(int x, int b[], int n) {
    if (x < 0) return GOOGOL;
    if (x == 0) return 0;
    int min = GOOGOL;
    for (int i = 0; i < n; i++) {
        int r = f(x - b[i], b, n);
        if (r < min) min = r;
    }
    return min + 1;
}
```

Задача о банкомате: борьба с излишней рекурсией

- Глубина рекурсии не превосходит $\frac{x}{\min_{i=1,n} b_i}$
- Количество рекурсивных вызовов растёт по экспоненте.
- Если при $b = \{1, 4, 6\}$ и $x = 30$ количество рекурсивных вызовов 285709, то для $x = 31$ оно уже 418729, а для $x = 40$ оно равно 597124768.

Задача о банкомате: борьба с излишней рекурсией

- Нам помогает сохранение промежуточных результатов.

```
const int GOOGOL = 999999999;
int f(int x, int *b, int n, int *cache) {
    if (x < 0) return GOOGOL;
    if (x == 0) return 0;
    if (cache[x] >= 0) return cache[x];
    int min = GOOGOL;
    for (int i = 0; i < n; i++) {
        int r = f(x - b[i], b, n, cache);
        if (r < min) min = r;
    }
    return cache[x] = min + 1;
}
```

Задача о банкомате: борьба с излишней рекурсией

- Массив *cache*, передаваемый в функцию решения должен быть подготовлен.
- Мы должны узнать, решалась ли эта задача ранее.
- Инициализация массива проводится элементами, которые не могут быть результатом решения задачи.
- Размер массива должен соответствовать x .

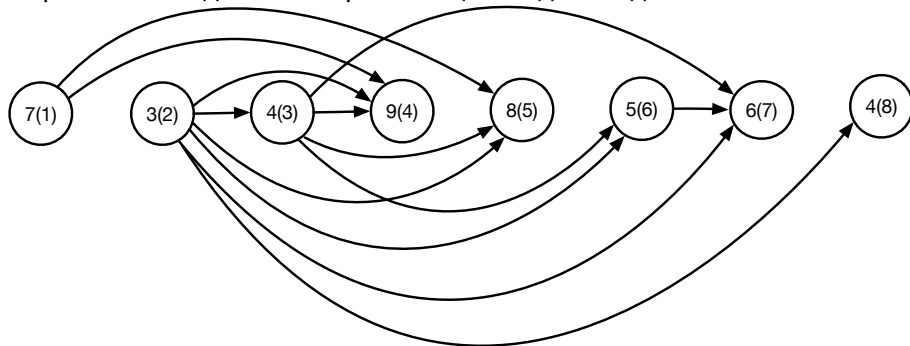
Нерекурсивный вариант решения – восходящее решение

- Решая задачу, мы замечаем, что аргументы функции располагаются «плотно» и между ними нет промежутков.
- В таких случаях короче получается вариант, когда мы вычисляем значения функции «снизу вверх».
- Для этого мы заполняем такой же массив ответов, но уже без рекурсии.

Уход от рекурсии. Восходящее решение.

Восходящее решение

- Вернёмся к задаче о возрастающей подпоследовательности.

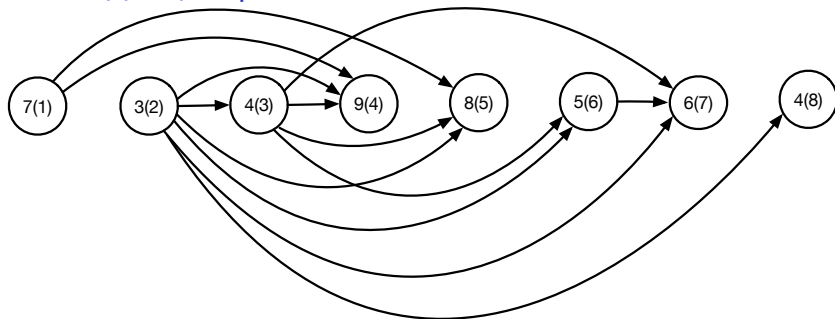


- При нисходящем решении нам требуется находить максимум из всех значений функций от $F(1)$ до $F(N)$, для чего рекурсивный вызов должен опуститься от $F(N)$ до $F(1)$.
- Для больших N уровень рекурсивных вызовов может превысить разумные рамки (размер стека в программах ограничен).

Восходящее решение

- При восходящем решении мы пробуем решать подзадачи *до того*, как они будут поставлены.
- Решая задачи в возрастающем порядке мы достигаем следующих целей:
 - 1 Гарантируется, что все задачи, решаемые позже, будут зависеть от ранее решённых.
 - 2 Не потребуются рекурсивные вызовы.
- Обязательное условие: монотонность аргументов при решении подзадач, если $f(k)$ — подзадача для $f(n)$, то $M(k) < M(n)$, где $M(x)$ есть некая метрика сложности решения.

Восходящее решение



- $F(1) = F(2) = 1$
- $F(3) = \max(F(2)) + 1 = 2$
- $F(4) = \max(F(1), F(2), F(3)) + 1 = 3$
- ...
- $F(8) = \max(F(2)) + 1 = 2$

После получения значений $F(i)$ не требуется вызывать функцию, можно использовать уже вычисленное значение.

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int n; scanf("%d", &n);
    int *b = calloc(n, sizeof(int));
    for (int i = 0; i < n; i++) scanf("%d", b+i);
    int x; scanf("%d", &x);
    int *cache = calloc(x+1, sizeof (int));
    for (int i = 1; i <= x; i++) cache[i] = 9999999;
    cache[0] = 0;
    for (int i = 0; i <= x; i++) {
        for (int j = 0; j < n; j++) {
            if (cache[i] + 1 < cache[i+b[j]])
                cache[i+b[j]] = cache[i] + 1;
        }
    }
    printf("%d\n", cache[x]);
    free(cache); free(b);
}
```

Восходящее решение

- Восходящее решение не всегда оправдано.
- Пусть решение задачи определяется таким образом:

$$F(n) = \max(F((n + 1)/2), F(n/3)) + 1$$

$$F(0) = F(1) = 1$$

- Это описывает какую-то последовательность.
- При нисходящем способе для $F(8)$ мы получаем:
- $F(8) = \max(F(4), F(2)) + 1$
- $F(4) = \max(F(2), F(1)) + 1$
- $F(2) = \max(F(1), F(0)) + 1 = 2$
- $F(4) = 3$
- $F(8) = 4$

Восходящее решение

- При попытке решить задачу восходящим решением мы получим:
- $F(0)=F(1)=1$
- $F(2)=\max(F(1),F(0))+1=2$
- $F(3)=\max(F(2),F(1))+1=3$
- $F(4)=\max(F(2),F(1))+1=3$
- $F(5)=\max(F(3),F(1))+1=3$
- $F(6)=\max(F(3),F(2))+1=3$
- $F(7)=\max(F(4),F(2))+1=4$
- $F(8)=\max(F(4),F(3))+1=4$
- Значения $F(3),F(5),F(6),F(7)$ были вычислены зря.

Восходящее решение

- Поставленная задача более подходила под рекурсивный алгоритм с запоминанием промежуточных результатов (*memoizing*).
- Нисходящее решение $F(N)$ имеет сложность $O(\log N)$.
- Восходящее решение $F(N)$ имеет сложность $O(N)$.

Восходящее решение vs нисходящее

- Восходящее решение по алгоритмической корректности эквивалентно нисходящему.
- Необходимо обеспечить вычисление в соответствующем порядке.
- При простом целочисленном аргументе вычисления можно проводить в порядке увеличения аргумента.
- Восходящее решение требует меньшего размера стека, но может решать задачи, ответ на которые не понадобится при решении главной задачи.

Восстановление решения

Задача о банкомате: нахождение банкнот

- Мы искали решение задачи минимизации.
- До сих пор нас интересовал только ответ, значение минимума.
- Мы его получили.
- Мы не получили, какие именно банкноты требуется выдать.

Задача о банкомате: нахождение банкнот

- Мы искали решение задачи минимизации.
- До сих пор нас интересовал только ответ, значение минимума.
- Мы его получили.
- Мы не получили, какие именно банкноты требуется выдать.
- Имеется несколько подходов к получению детального решения:
 - ▶ Каждый раз при нахождении решения подзадачи мы сохраняем историю его получения.
 - ▶ Зная ответ, мы повторяем решение, воссоздавая историю получения.

Задача о банкомате: восстановление решения

Первый способ: сохранять цепочку вызовов.

- Потребуется: иметь список банкнот для каждого промежуточного решения.
- Всего промежуточных решений может быть N .
- Каждое из решений имеет различную длину (вектор).
- Потребуется N векторов.
- Явно, что всё потребует много памяти и времени.

Задача о банкомате: восстанавливаем решение

- Зная ответ и имея кэш-таблицу, можно восстановить решение.
- Предположим, что мы знаем, что точный ответ при заданных начальных значениях есть 7.
- Тогда возникает вопрос: какой предыдущий ход мы сделали, чтобы попасть в заключительную позицию?
- Введём термин *ранг* для обозначения наименьшего числа ходов, требуемого для достижения текущей позиции из начальной.
- Тогда каждый ход решения всегда переходит в позицию с рангом, большим строго на единицу.

Задача о банкомате: восстанавливаем решение

- Восстановление решения по таблице ответов:
 - 1 Заключительная позиция имеет ранг k .
 - 2 Делаем позицию текущей.
 - 3 Если текущая позиция имеет ранг 0, то это — начальная позиция и алгоритм завершён.
 - 4 Рассматриваем все позиции, ведущие в текущую и выбираем из них произвольную с рангом $k - 1$.
 - 5 Запоминаем ход, приведший к из позиции ранга $k - 1$ в ранг k .
 - 6 Понижаем ранг: $k \rightarrow k - 1$ и переходим к 2.

Задача о банкомате: восстанавливаем решение

```
void buildSolution(int x, int *b, int n, int *cache){
    vector<int> ret;
    for (int k = cache[x]; k >= 0; k--) {
        for (int i = 0; i < n; i++) {
            int r = x - b[i];
            if (r >= 0 && cache[r] == k-1) {
                x = r;
                printf("%d ", b[i]);
                break;
            }
        }
    }
    return ret;
}
```

Решение задачи о банкомате: восстановление

Третья возможность: добавить ещё один кэш, сохраняющий номинал лучшей банкноты при лучшем решении.

```
const int GOOGOL = 999999999;
int f(int x, int *b, int n,
    int *cache, int *bestnote) {
    if (x < 0) return GOOGOL;
    if (x == 0) return 0;
    if (cache[x] >= 0) return cache[x];
    int min = GOOGOL, best = -1;
    for (int i = 0; i < n; i++) {
        int r = f(x - b[i], b, n, cache, bestnote);
        if (r < min) {
            min = r;
            bestnote[x] = b[i];
        }
    }
    return cache[x] = min + 1;
}
```

Решение задачи о банкомате: восстанавливаем решение

Теперь для того, чтобы получить нужные для размена банкноты для суммы в x , достаточно пробежаться по кэшу банкнот:

```
while (x > 0) {  
    // output bestnote[x];  
    x -= bestnote[x];  
}
```