Лекция 4. Динамическое программирование.

План лекции

- Суть динамического программирования
- Преимущества перед «наивными» методами решения
- Динамика и рекурсия. Числа Фибоначчи
- Задача о кузнечике
- Стоимость маршрута в матрице
- Наибольшая общая подпоследовательность

Динамическое программирование

Динамическое программирование, динамика – это способ решения больших и сложных задач с помощью разделения их на мелкие подзадачи.

Главный принцип динамики заключается в запоминании ранее вычисленных значений.

Задача о лестнице

У лестницы n ступенек, пронумерованных числами 1, 2, . . . , n снизу вверх. На каждой ступеньке написано число. Начиная с подножия лестницы (его можно считать ступенькой с номером 0), требуется взобраться на самый верх (ступеньку с номером n).

За один шаг можно подниматься на одну или на две ступеньки. После подъёма числа, записанные на посещённых ступеньках, складываются.

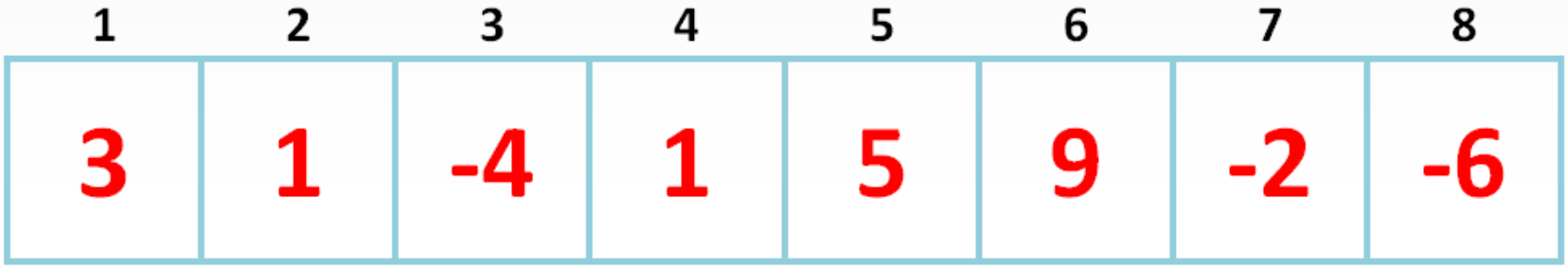
Нужно подняться по лестнице так, чтобы сумма этих чисел была как можно больше.

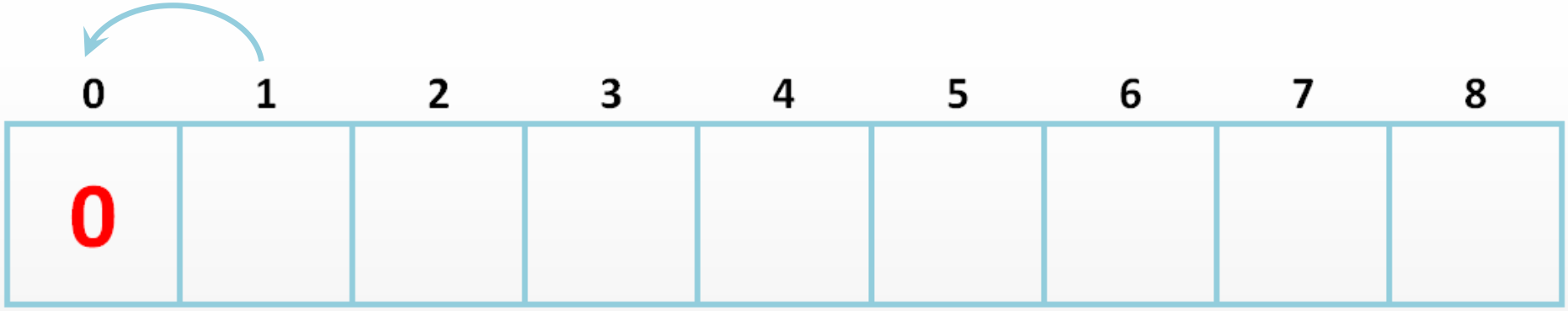
Задача о лестнице

1	2	3	4	5	6	7	8
3	1	-4	1	5	9	-2	-6

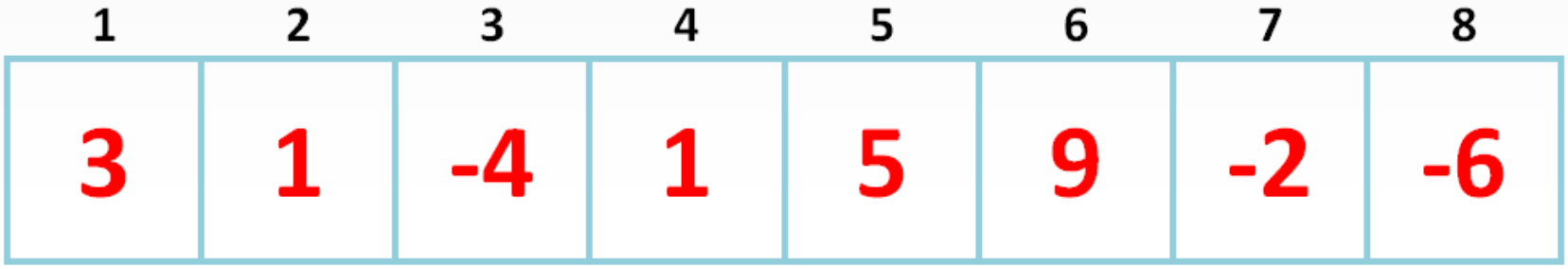
0	1	2	3	4	5	6	7	8
0								

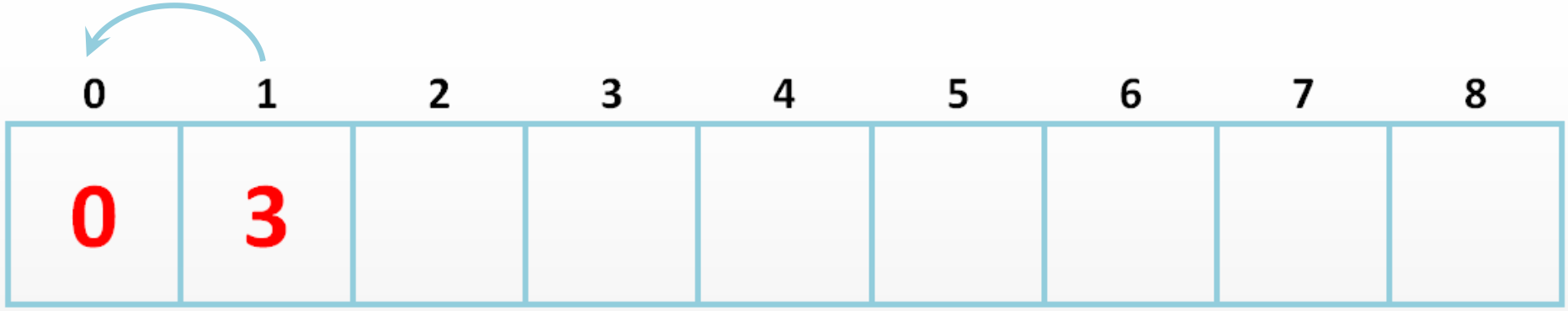
Задача о лестнице



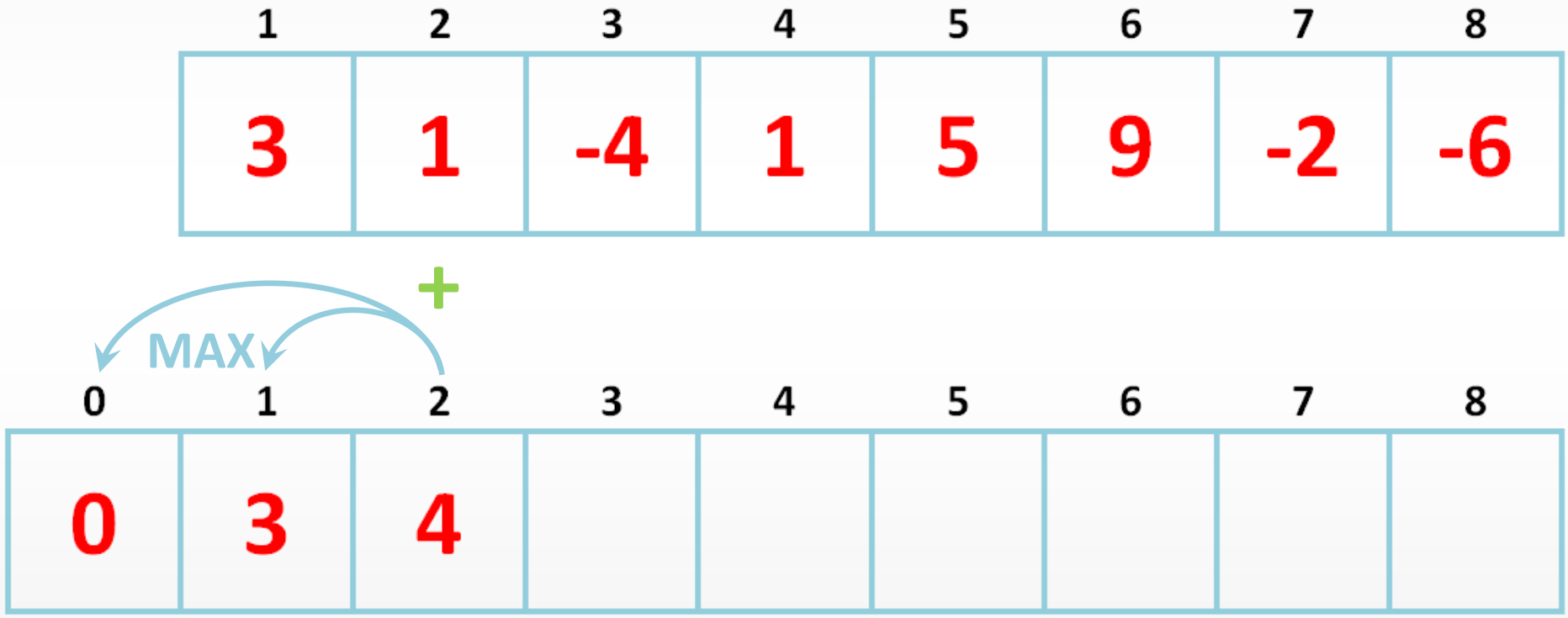


Задача о лестнице

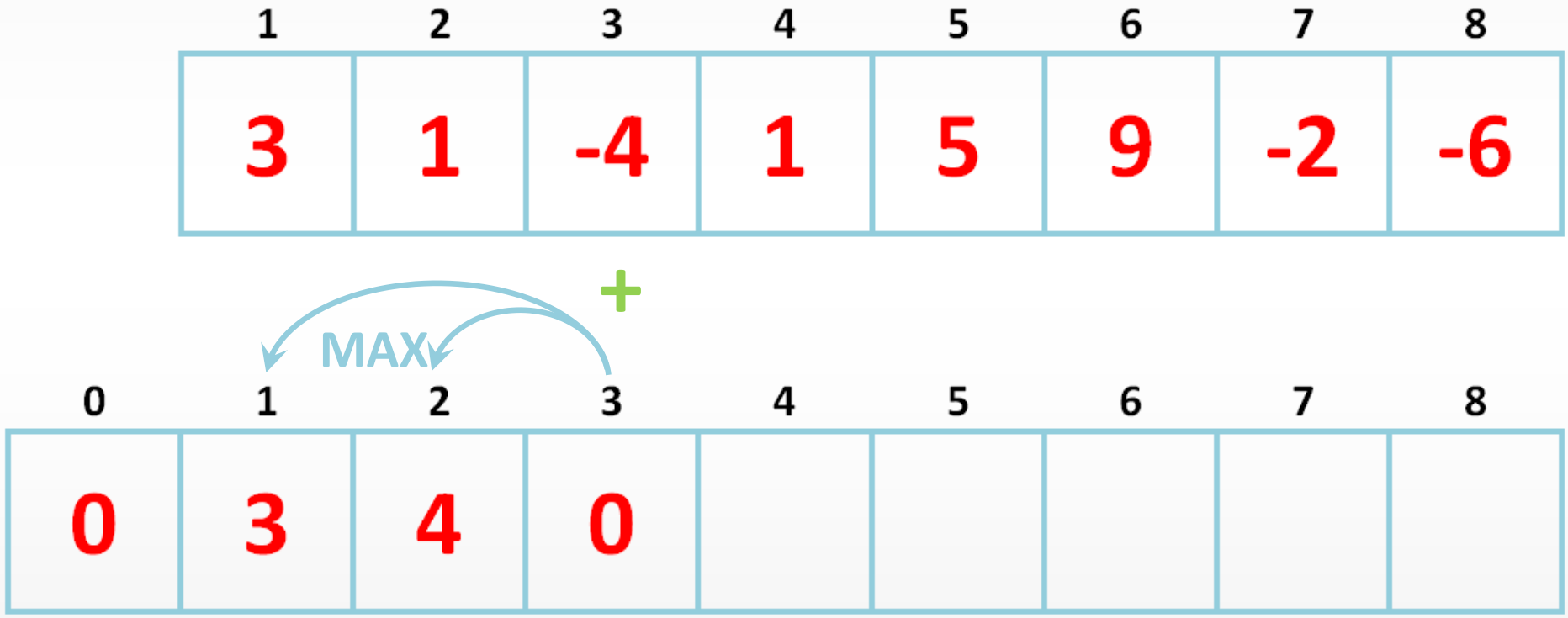




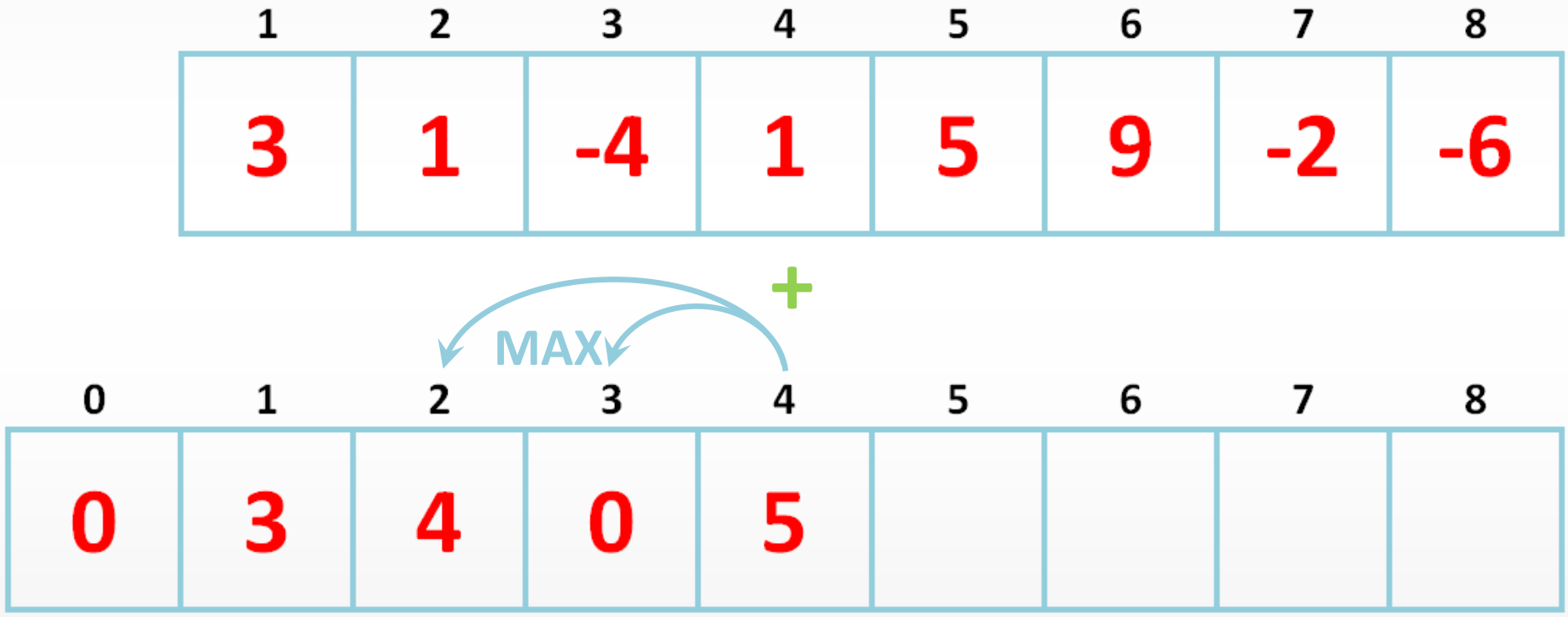
Задача о лестнице



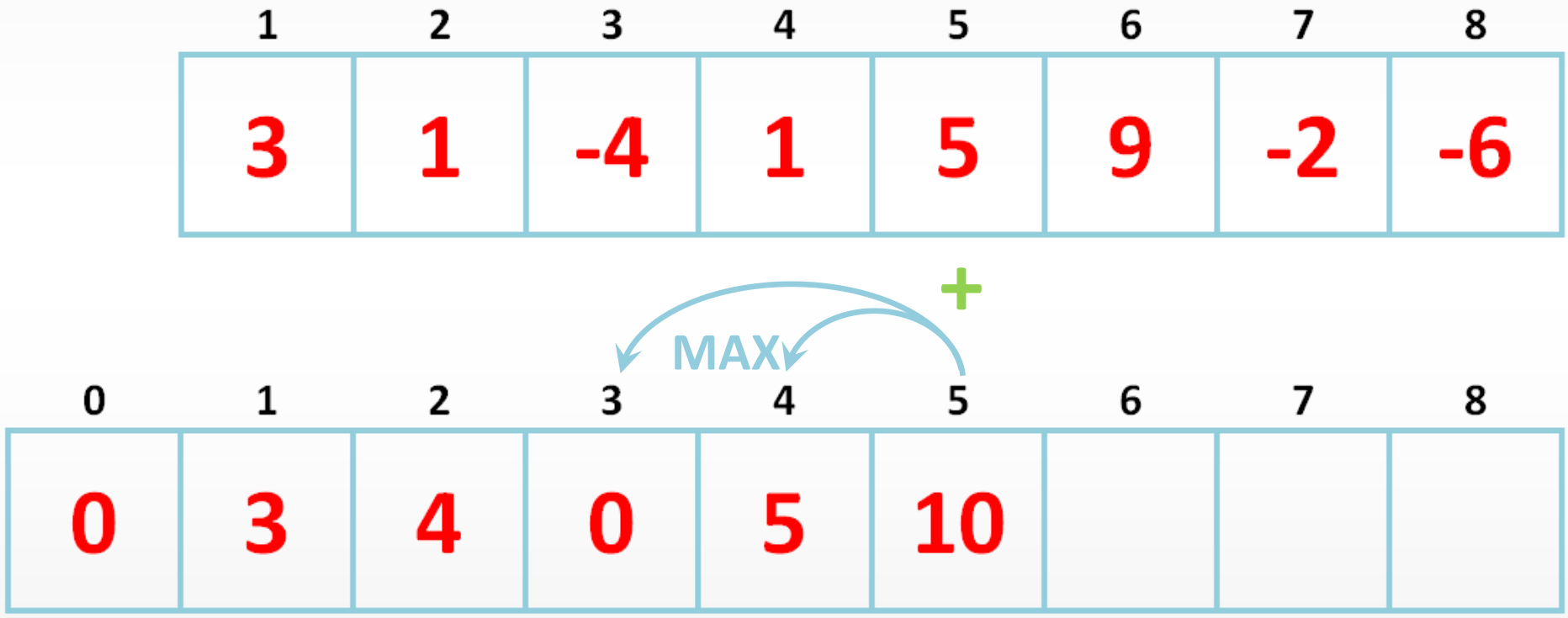
Задача о лестнице



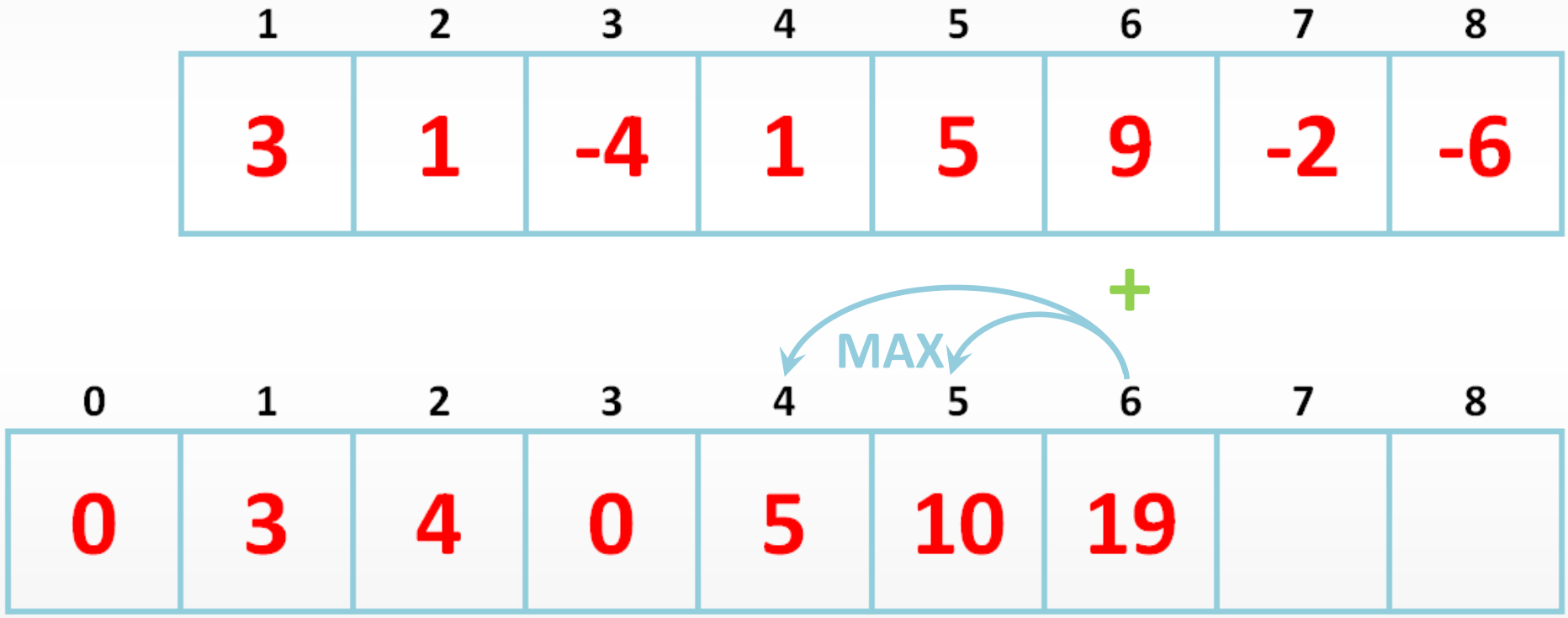
Задача о лестнице



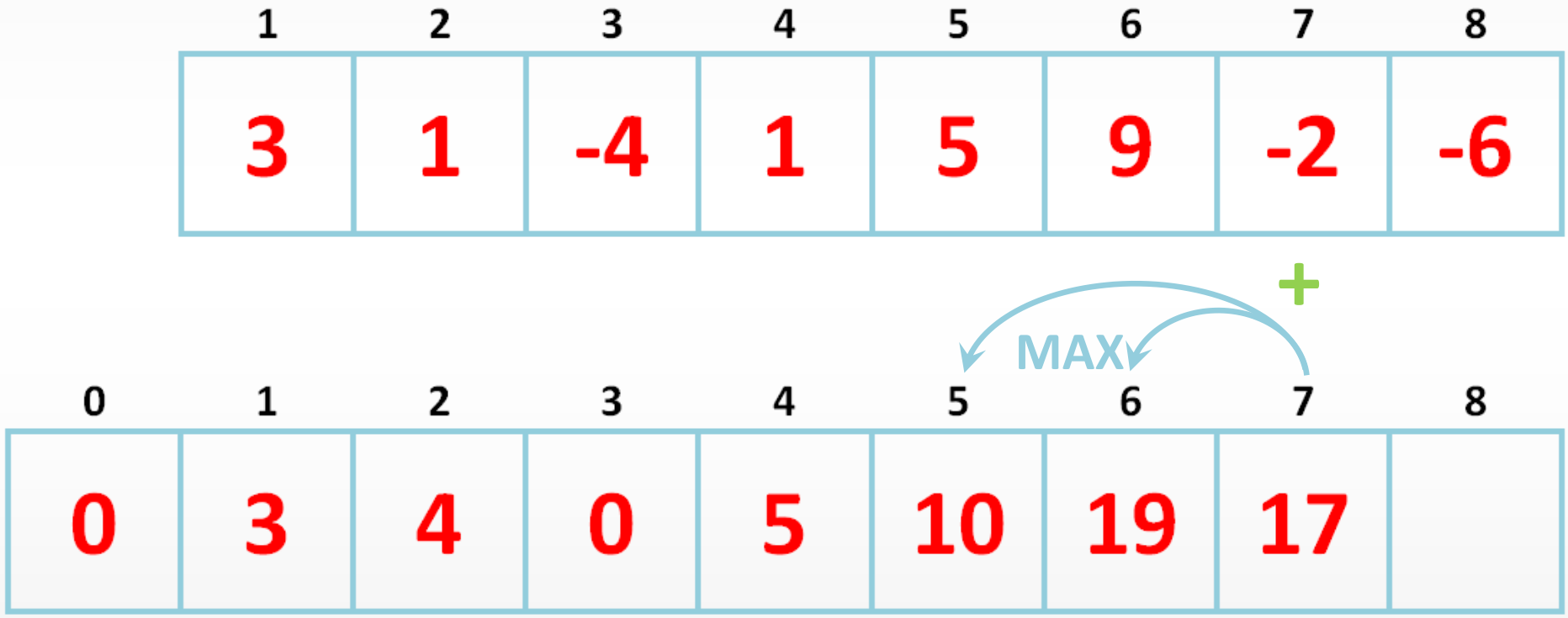
Задача о лестнице



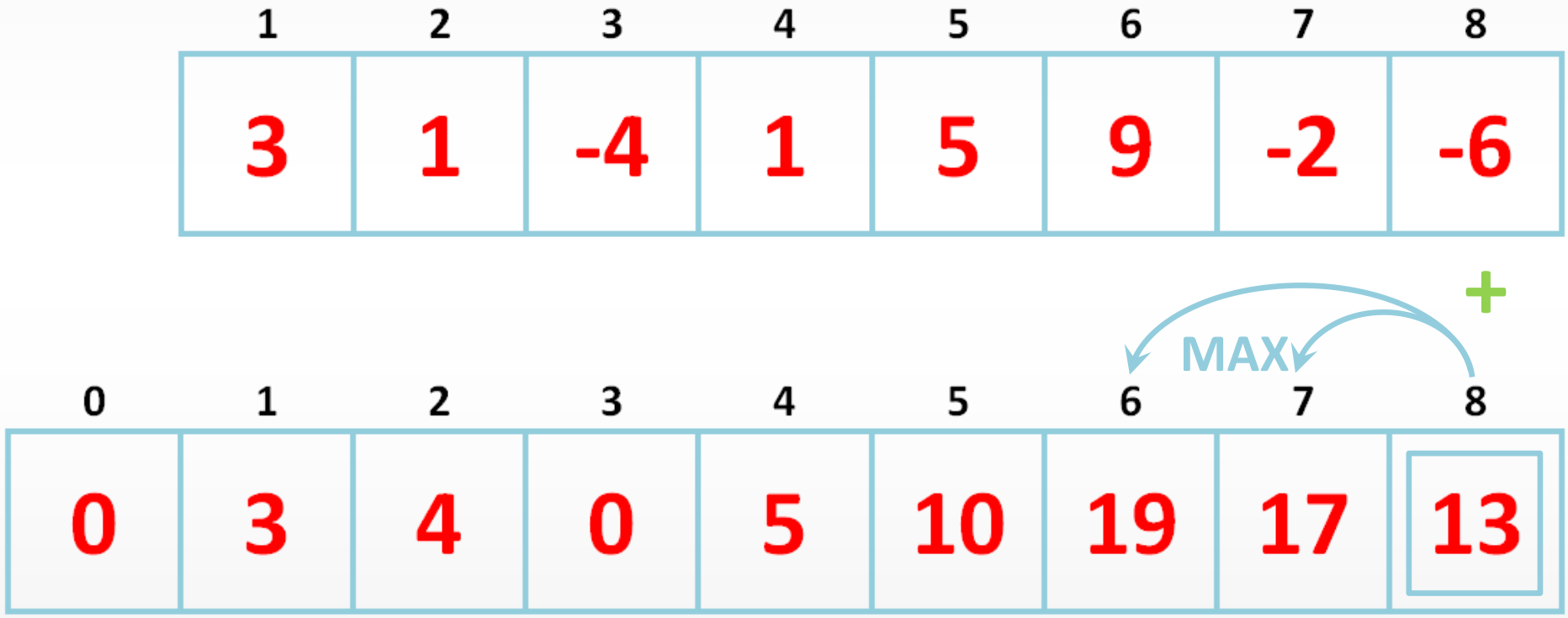
Задача о лестнице



Задача о лестнице



Задача о лестнице



Задача о лестнице

```
#include <iostream>
using namespace std;
int main() {
    long long n, a[102], b;

    cin >> n >> a[1];
    a[0] = 0;
                //Если количество ступеней - 1, ответ -
    if(n < 2) { // единственная ступенька.
        cout << a[1];
        return 0;
    }

    for(int i = 2; i <= n; i++) {
        cin >> b;
        a[i] = max( a[i-1], a[i-2] ) + b;
    }

    cout << a[n];
    return 0;
}
```

Динамическое программирование

Как правило, динамика выполняется за линейное время O(n).

Если посчитать, та же задача про лестницу обыкновенным полным перебором решалась бы за $O(2^{n-3})$.

ДИНАМИКА ВЫГОДНЕЕ ПЕРЕБОРА.

Динамика и рекурсия

Стоит задача вычислить k-е число Фибоначчи – элемент последовательности, в которой каждый элемент равен сумме двух предыдущих.

$$a_1 = 1; \quad a_2 = 1; \quad a_n = a_{n-1} + a_{n-2}$$

1 1 2 3 5 8 13 21 34 55 89

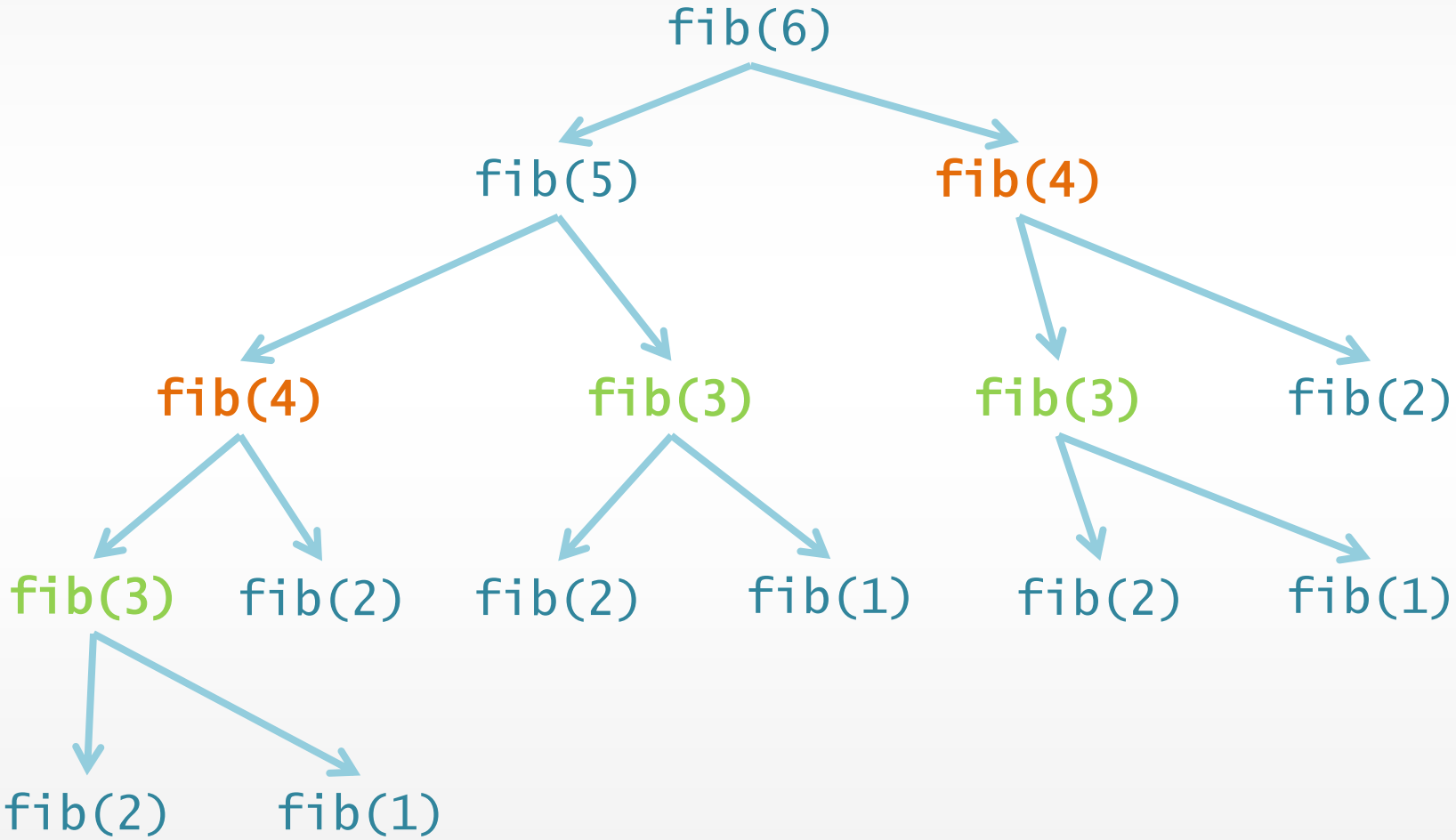
Динамика и рекурсия

Можно для этого использовать рекурсивную функцию:

```
int fib(int index) {
    if(index == 1 || index == 2)
        return 1;
    return fib(index-1) + fib(index-2)
}
```

Но что будет делать этот код при больших числах?

Динамика и рекурсия

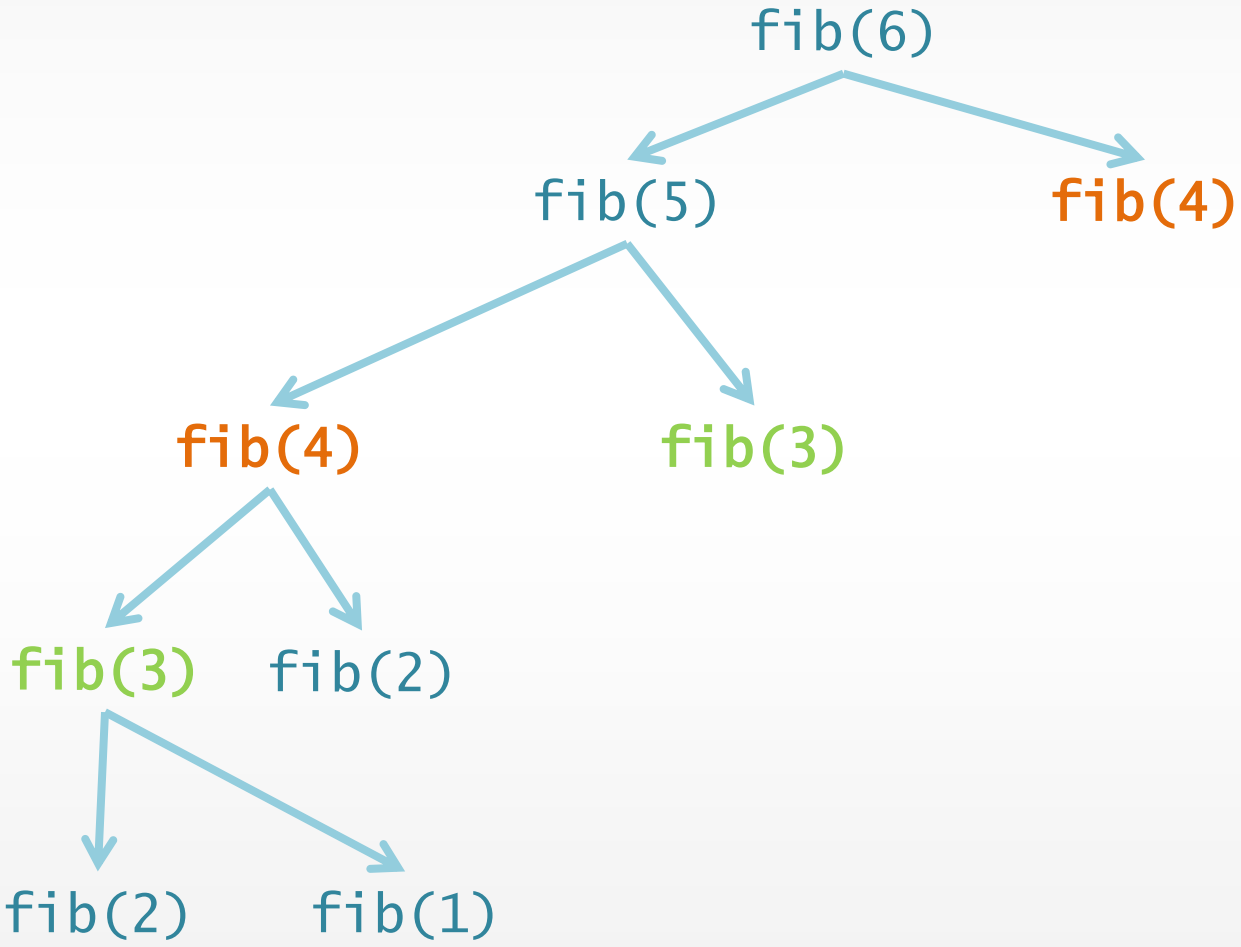


Динамика и рекурсия

Динамикой (то есть простым запоминанием результатов) можно избежать повторений в вычислениях:

```
vector <int> a(k, -1);
int fib(int index) {
    if(a[index] < 0)
        a[index] = fib(index-1) + fib(index-2);
    return a[index];
}
```

Динамика и рекурсия



Задача о кузнечике/зайчике/…

Зайчик прыгает по прямой просеке, для удобства разделённой на n клеток. Клетки пронумерованы по порядку натуральными числами от 1 до n. Некоторые клетки заболочены (‘ w ’): если зайчик прыгнет на такую клетку, ему несдобровать. Некоторые другие клетки просеки поросли вкусной зелёной травой (‘ ” ’): прыгнув на такую клетку, зайчик сможет отдохнуть и подкрепиться.

Зайчик начинает свой путь из клетки с номером 1 и хочет попасть в клетку с номером n, по пути ни разу не провалившись в болото и скушав как можно больше вкусной зелёной травы. Конструктивные особенности зайчика таковы, что из клетки с номером k он может прыгнуть лишь в клетки с номерами k + 1, k + 3 и k + 5.

Выясните, какое максимальное количество клеток с травой сможет посетить зайчик на своём пути.

Задача о кузнечике/зайчике/…

1	2	3	4	5	6	7	8
.	.	.	W	“	“	W	.

1	2	3	4	5	6	7	8
0	-1	-1	-1	-1	-1	-1	-1

Задача о кузнечике/зайчике/…

Массив динамики изначально занят -1, кроме первого элемента, который равен 0.

-1 значит, что на эту клетку заяц попасть не смог.

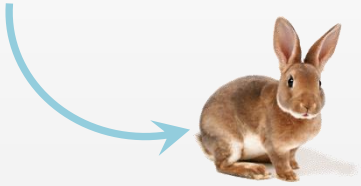
Числа ≥ 0 – это максимальное количество клеток с травой, которые на своём пути посетил заяц.

Логично, что в клетку k заяц может попасть только из клеток k-1, k-3 и k-5. Тогда максимальное количество клеток с травой, посещённых до клетки k включительно, будет равняться максимальному количеству очков среди тех клеток, откуда мы могли прийти, и +1 в том случае, если k – сама клетка с травой.

Задача о кузнечике/зайчике/…

1	2	3	4	5	6	7	8
.	.	.	W	“	“	W	.

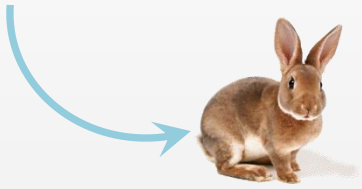
1	2	3	4	5	6	7	8
0	0	-1	-1	-1	-1	-1	-1



Задача о кузнечике/зайчике/…

1	2	3	4	5	6	7	8
.	.	.	W	“	“	W	.

1	2	3	4	5	6	7	8
0	0	0	-1	-1	-1	-1	-1



Задача о кузнечике/зайчике/…

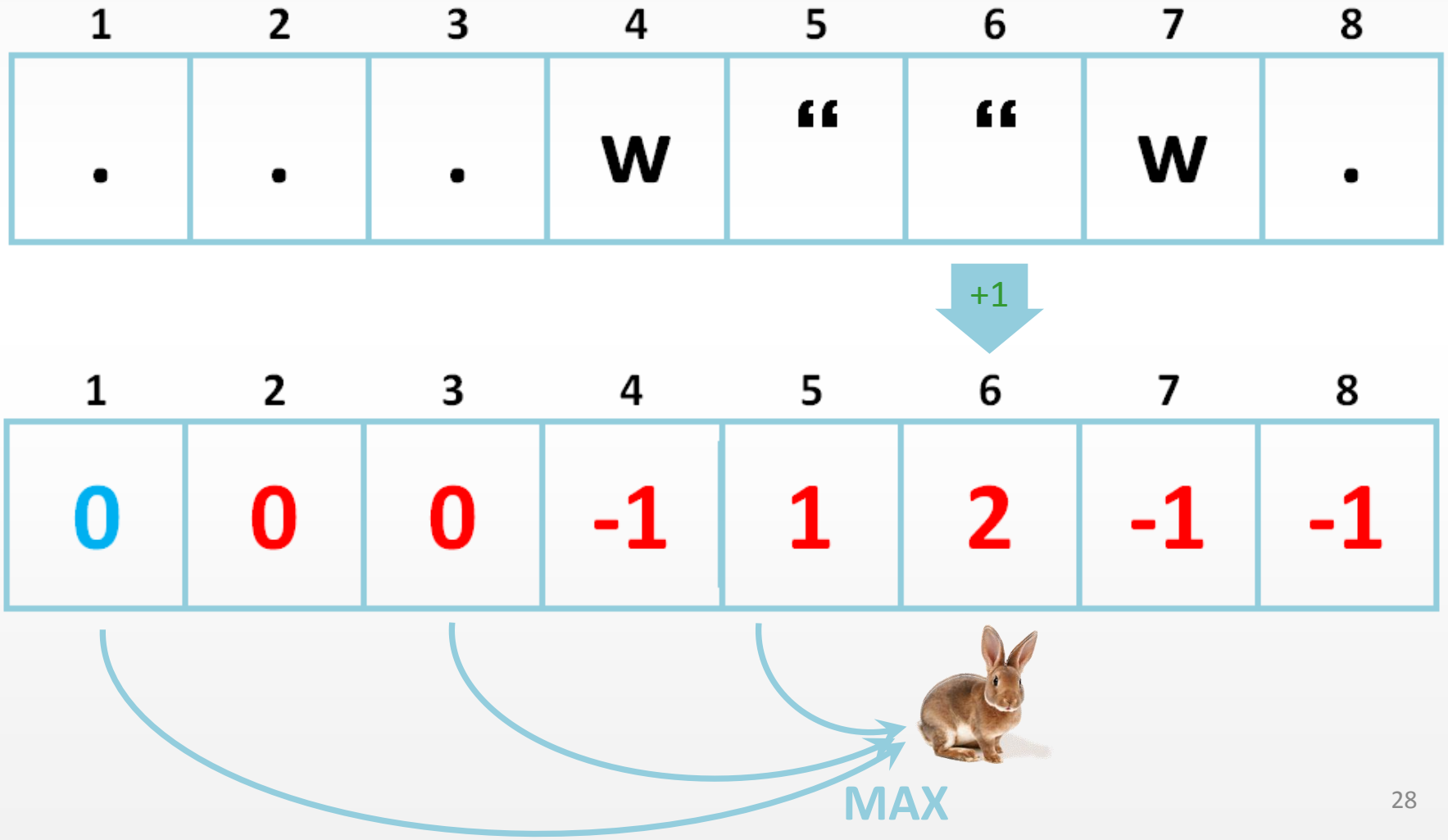
1	2	3	4	5	6	7	8
.	.	.	w	“	“	w	.

+1

1	2	3	4	5	6	7	8
0	0	0	-1	1	-1	-1	-1

MAX

Задача о кузнечике/зайчике/…



Задача о кузнечике/зайчике/…

1	2	3	4	5	6	7	8
.	.	.	w	“	“	w	.

1	2	3	4	5	6	7	8
0	0	0	-1	1	2	-1	1

MAX

Задача о кузнечике/зайчике/…

```
vector <int> a(n+2, -1);
char c; cin >> c;
a[1] = 0;
for(int i = 2, i <= n; i++) {
    cin >> c;
    if(c ==‘w’) continue;

    if(i < 3)
        a[i] = a[i-1];
    else if(i < 5)
        a[i] = max( a[i-1], a[i-3] );
    else
        a[i] = max( max( a[i-1], a[i-3] ), a[i-5] );

    if(c == ‘”’) a[i]++;
}

cout << a[n];
```

Стоимость маршрута в матрице

Справка:

Матрица, или двухмерный массив – это массив массивов, т.е. массив, каждый элемент которого – тоже массив.

```
int a[10][10];
for(int i=0;i<10;i++)
    for(int j=0;j<10;j++)
        a[i][j]=i*j;
```

Стоимость маршрута в матрице

Имеется матрица размера NxM, в каждой клетке которой записано некоторое положительное число. Какую максимальную сумму можно набрать, двигаясь по матрице из левого верхнего в правый нижний угол при условии, что ходить можно только вниз и вправо? N и M ≤ 100.

Стоимость маршрута в матрице

Пусть `a[][]` – это матрица с числами, а `dp[][]` – это матрица динамики.

```
for(int i = 0; i < N; i++)  dp[i][0] = 0;
for(int i = 0; i < M; i++)  dp[0][i] = 0;
for(int i = 1; i <= N; i++)
   for(int j = 1; j <= M; j++)
      dp[i][j] = max(dp[i-1][j], dp[i][j-1]) + a[i][j];
```

Стоимость маршрута в матрице

a[][]

5	9	2	1	2
3	1	2	3	1
8	3	2	4	1
8	7	5	5	4
2	1	2	3	3

Стоимость маршрута в матрице

0	0	0	0	0	0
0	5	14	16	17	19
0	8	15	18	21	22
0	16	19	21	25	26
0	24	31	36	41	45
0	26	32	38	44	48

dp[][]

Стоимость маршрута в матрице

`dp[][]`

Восстановление пути: двигаемся с конца, всегда переходя из `dp[i][j]` в максимальное из `dp[i-1][j]` и `dp[i][j-1]`

Наибольшая общая подпоследовательность

Имеются две последовательности символов (или чисел, неважно) длин N и M. Требуется найти их наибольшую общую подпоследовательность (LCS).

Пример:

a = “abcdf”

b = “abdca”

LCS(a,b) = “abc” или “abd”

Наибольшая общая подпоследовательность

		a	b	d	c	a
	0	0	0	0	0	0
a	0	1	1	1	1	1
b	0	1	2	2	2	2
c	0	1	2	2	3	3
d	0	1	2	3	3	3
f	0	1	2	3	3	3

В `dp[i][j]` записана длина наибольшей общей подпоследовательности подстрок искомых строк длин `i` и `j` соответственно.

Например, `dp[2][3]` – это ответ на задачу для строк “ab” и “abd”.

```
for(i = 1..N)
 for(j = 1..M)
  if(a[i] == b[j])
   dp[i][j] = dp[i-1][j-1] + 1;
  else
   dp[i][j] = max( dp[i-1][j],
                   dp[i][j-1] );
```

Наибольшая общая подпоследовательность

		a	b	d	c	a
	0	0	0	0	0	0
a	0	1	1	1	1	1
b	0	1	2	2	2	2
c	0	1	2	2	3	3
d	0	1	2	3	3	3
f	0	1	2	3	3	3

Чтобы найти саму LCS, надо пройтись по матрице `dp[][]` и найти места, где `dp[i][j] > dp[i-1][j]` и `dp[i][j] > dp[i][j-1]`, и после нахождения каждого такого уголка исследовать `dp[][]` по индексам, строго меньшим, чем у найденного уголка. Это поможет в ситуациях, когда есть больше, чем одна LCS.

Вопросы?