COMP1201
Algorithms

Dynamic Programming

Jie Zhang
jie.zhang@soton.ac.uk
Electronics and Computer Science
University of Southampton

Dynamic Programming (DP)

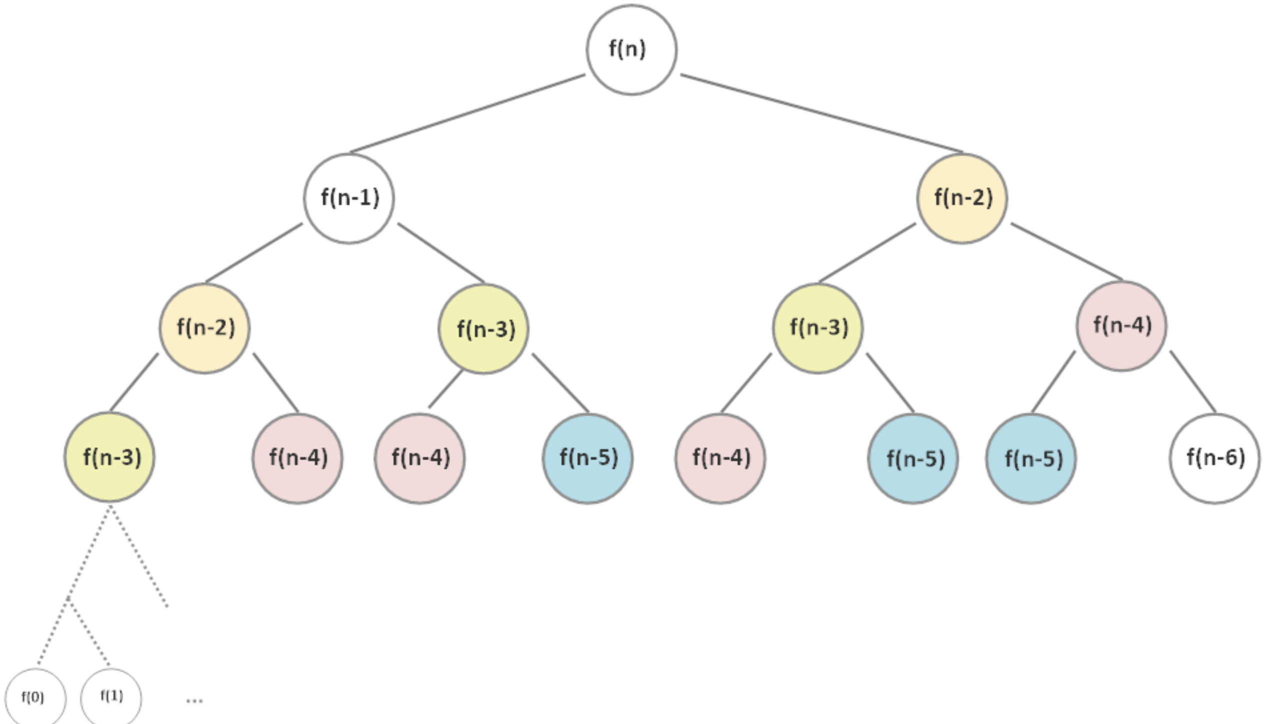
- DP ≈ solve sub-problems (remember the solutions)
  & re-use them receptively

- Fibonacci numbers
- A toy problem

- Why good? Poly-time
- Applicability

An example

- Fibonacci numbers
  - $f(0) = 0, f(1) = 1$
  - $f(n) = f(n\text{-}1) + f(n\text{-}2)$



First attempt

- A naïve recursive algorithm

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

- Time complexity
  - $\Omega(2^{n/2}) < T(n) < O(2^n)$, actually $T(n) \sim \Theta(\varphi^n)$, where φ is the golden ratio.
- What if we remember $f(k)$?
  - Cache

Recursion with memory

- memo = {}

  $f(n)$: if n is in memo, return memo[n]

  $$\left.\begin{array}{l} \text{o.w., } f(0) = 0, f(1) = 1 \\ f = f(n\text{-}2) + f(n\text{-}1); \end{array}\right\} \text{Same as before}$$

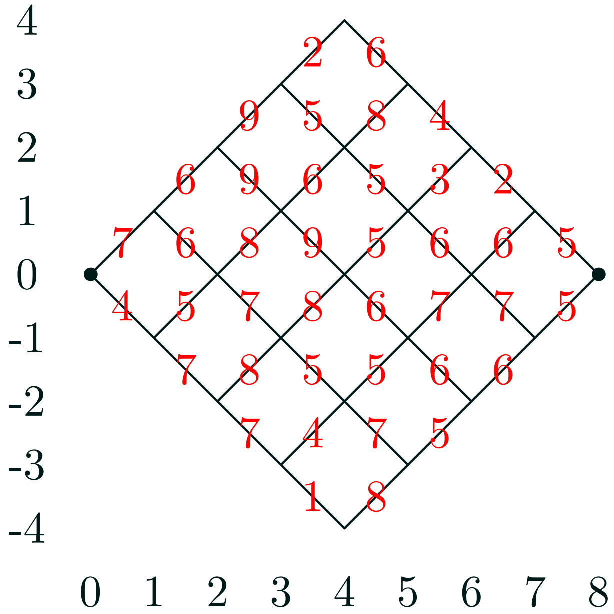
  memo[n] = f

  return f

- This way, we only compute each $f(k)$ once
- Run-time = # subproblems × time/subproblem
- Therefore, time complexity is $\Theta(n)$
- Space complexity is $\Theta(n)$
  - The height of the tree

A Toy Problem

- Consider the problem of find a minimum cost path from point (0,0) to (8,0) on the lattice

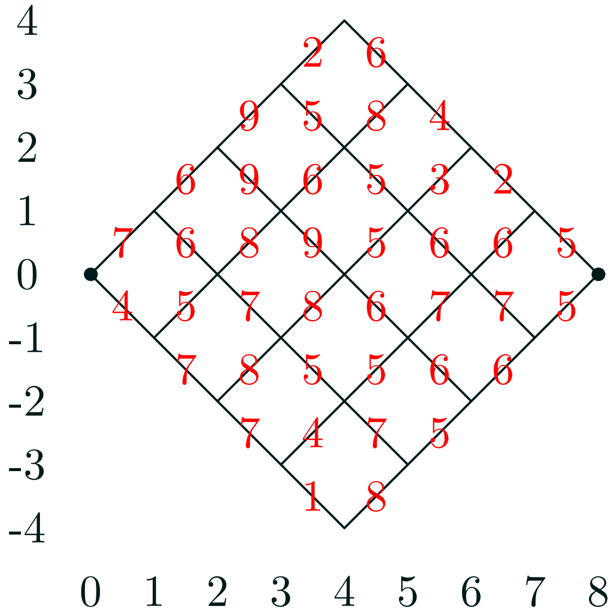


- The costs of traversing each link is shown in red

- The cost of a path is the sum of weights on each link

A Toy Problem

- Consider the problem of find a minimum cost path from point (0,0) to (8,0) on the lattice

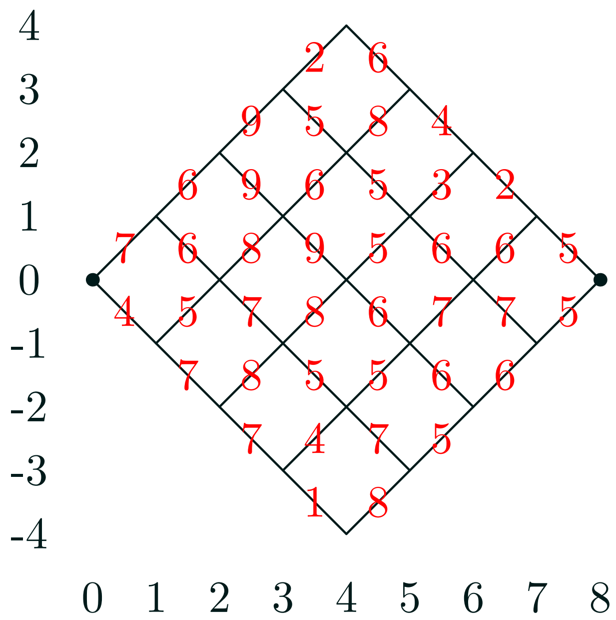


- The costs of traversing each link is shown in red

- The cost of a path is the sum of weights on each link

A Toy Problem

- Consider the problem of find a minimum cost path from point (0,0) to (8,0) on the lattice



- The costs of traversing each link is shown in red

- The cost of a path is the sum of weights on each link

Brute Force

- The obvious brute force strategy is to try every path
- For a problem with n steps we require $n/2$ to be diagonally up and $n/2$ to be diagonally down
- The total number of paths is

$$\binom{n}{n/2} \approx \sqrt{\frac{2}{\pi\, n}}\, 2^n$$

- For the problem shown above with $n = 8$ there are 70 paths
- For a problem with $n = 100$ there are 1.01×10^{29} paths

Brute Force

- The obvious brute force strategy is to try every path
- For a problem with n steps we require $n/2$ to be diagonally up and $n/2$ to be diagonally down
- The total number of paths is

$$\binom{n}{n/2} \approx \sqrt{\frac{2}{\pi\, n}}\, 2^n$$

- For the problem shown above with $n = 8$ there are 70 paths
- For a problem with $n = 100$ there are 1.01×10^{29} paths

Brute Force

- The obvious brute force strategy is to try every path
- For a problem with n steps we require $n/2$ to be diagonally up and $n/2$ to be diagonally down
- The total number of paths is

$$\binom{n}{n/2} \approx \sqrt{\frac{2}{\pi\, n}}\, 2^n$$

- For the problem shown above with $n = 8$ there are 70 paths
- For a problem with $n = 100$ there are 1.01×10^{29} paths

Brute Force

- The obvious brute force strategy is to try every path
- For a problem with n steps we require $n/2$ to be diagonally up and $n/2$ to be diagonally down
- The total number of paths is

$$\binom{n}{n/2} \approx \sqrt{\frac{2}{\pi\, n}}\, 2^n$$

- For the problem shown above with $n = 8$ there are 70 paths
- For a problem with $n = 100$ there are 1.01×10^{29} paths

Brute Force

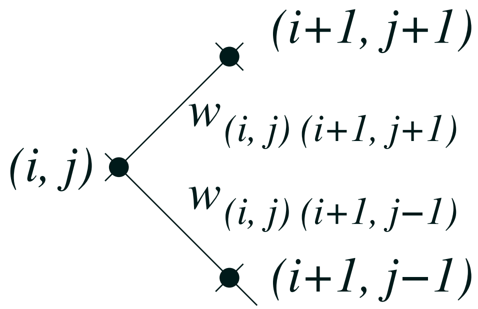
- The obvious brute force strategy is to try every path
- For a problem with n steps we require $n/2$ to be diagonally up and $n/2$ to be diagonally down
- The total number of paths is

$$\binom{n}{n/2} \approx \sqrt{\frac{2}{\pi\, n}}\, 2^n$$

- For the problem shown above with $n = 8$ there are 70 paths
- For a problem with $n = 100$ there are 1.01×10^{29} paths

Building a Solution

- We can solve this problem efficiently using dynamic programming by considering optimal paths of shorter length
- Let $c_{(i,j)}$ denote the cost of the optimal path to node (i, j)
- We denote the weights between two points on the lattice by $w_{(i,j)(i+1,j\pm1)}$



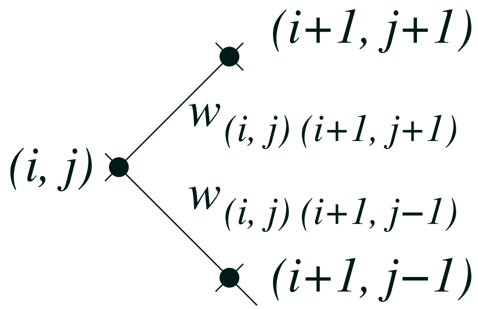
- Clearly $c_{(0,0)} = 0$

Building a Solution

- We can solve this problem efficiently using dynamic programming by considering optimal paths of shorter length

- Let $c_{(i,j)}$ denote the cost of the optimal path to node (i, j)

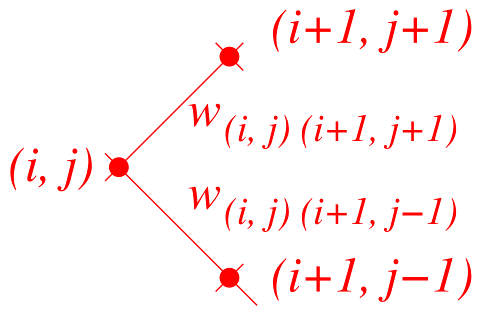
- We denote the weights between two points on the lattice by $w_{(i,j)(i+1,j\pm1)}$



- Clearly $c_{(0,0)} = 0$

Building a Solution

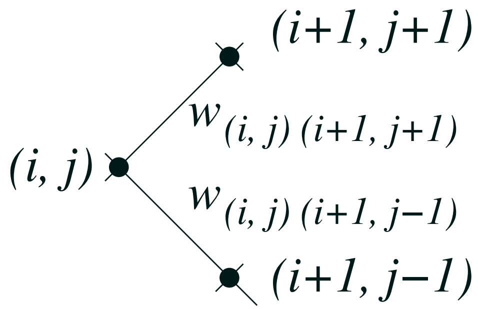
- We can solve this problem efficiently using dynamic programming by considering optimal paths of shorter length
- Let $c_{(i,j)}$ denote the cost of the optimal path to node (i,j)
- We denote the weights between two points on the lattice by $w_{(i,j)(i+1,j\pm1)}$



- Clearly $c_{(0,0)} = 0$

Building a Solution

- We can solve this problem efficiently using dynamic programming by considering optimal paths of shorter length
- Let $c_{(i,j)}$ denote the cost of the optimal path to node (i, j)
- We denote the weights between two points on the lattice by $w_{(i,j)(i+1,j\pm1)}$



- Clearly $c_{(0,0)} = 0$

Forward Algorithm

- Suppose we know the optimal costs for all the edge in column i
- Our task is to find the optimal cost at column $i+1$
- If we consider the sites in the lattice then the optimal cost will be

$$c_{(i+1,j)} = \min\big(c_{(i,j+1)} + w_{(i,j+1)(i+1,j)},\ c_{(i,j-1)} + w_{(i,j-1)(i+1,j)}\big)$$

- This is the defining equation in dynamic programming
- We have to treat the boundary sites specially, but this is just book-keeping

Forward Algorithm

- Suppose we know the optimal costs for all the edge in column i
- Our task is to find the optimal cost at column $i+1$
- If we consider the sites in the lattice then the optimal cost will be

$$c_{(i+1,j)} = \min\big(c_{(i,j+1)} + w_{(i,j+1)(i+1,j)},\ c_{(i,j-1)} + w_{(i,j-1)(i+1,j)}\big)$$

- This is the defining equation in dynamic programming
- We have to treat the boundary sites specially, but this is just book-keeping

Forward Algorithm

- Suppose we know the optimal costs for all the edge in column i
- Our task is to find the optimal cost at column $i+1$
- If we consider the sites in the lattice then the optimal cost will be

$$c_{(i+1,j)} = \min\bigl(c_{(i,j+1)} + w_{(i,j+1)(i+1,j)},\ c_{(i,j-1)} + w_{(i,j-1)(i+1,j)}\bigr)$$

- This is the defining equation in dynamic programming
- We have to treat the boundary sites specially, but this is just book-keeping

Forward Algorithm

- Suppose we know the optimal costs for all the edge in column i
- Our task is to find the optimal cost at column $i+1$
- If we consider the sites in the lattice then the optimal cost will be

$$c_{(i+1,j)} = \min\big(c_{(i,j+1)} + w_{(i,j+1)(i+1,j)},\ c_{(i,j-1)} + w_{(i,j-1)(i+1,j)}\big)$$

- This is the defining equation in dynamic programming
- We have to treat the boundary sites specially, but this is just book-keeping

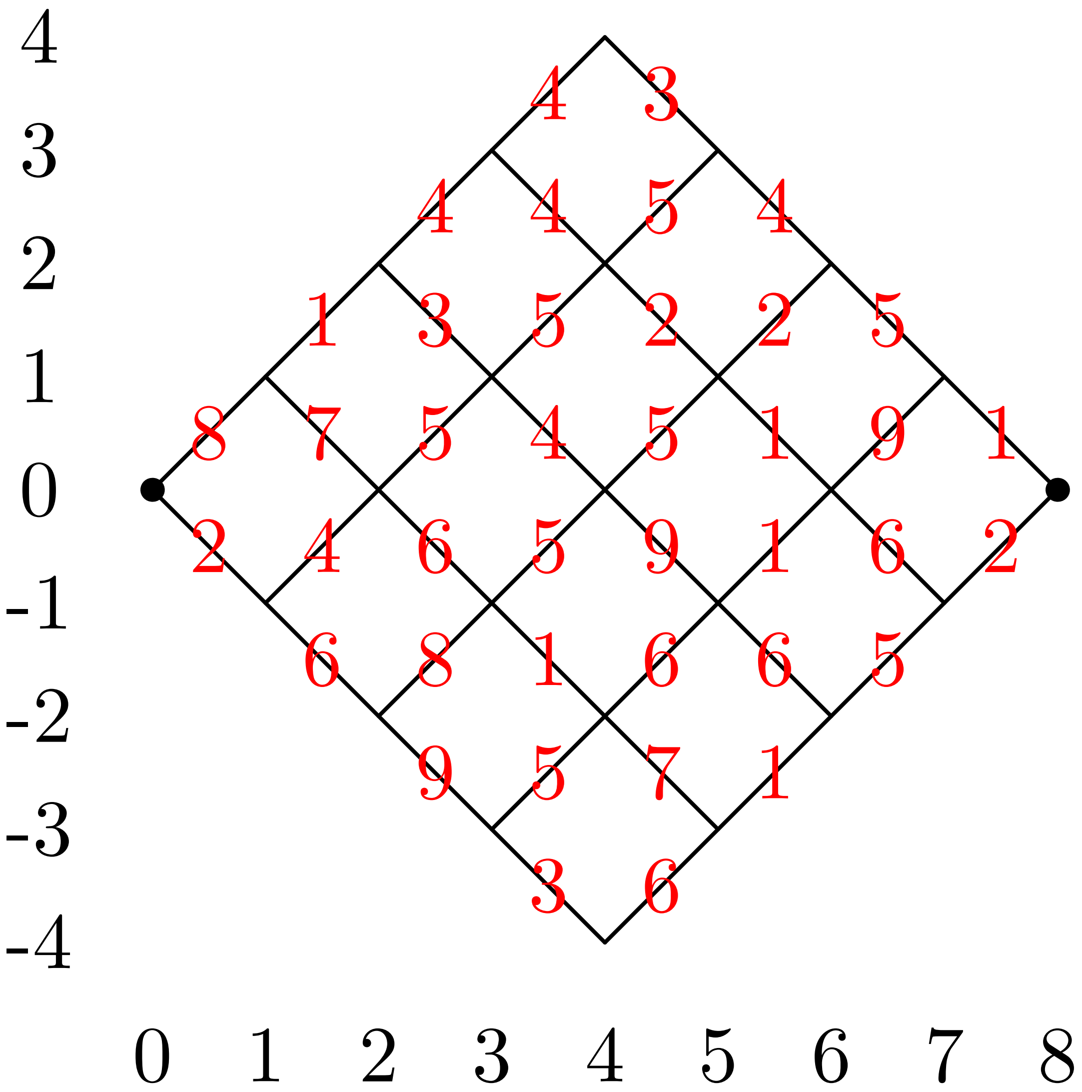
Forward Algorithm

- Suppose we know the optimal costs for all the edge in column i
- Our task is to find the optimal cost at column $i+1$
- If we consider the sites in the lattice then the optimal cost will be

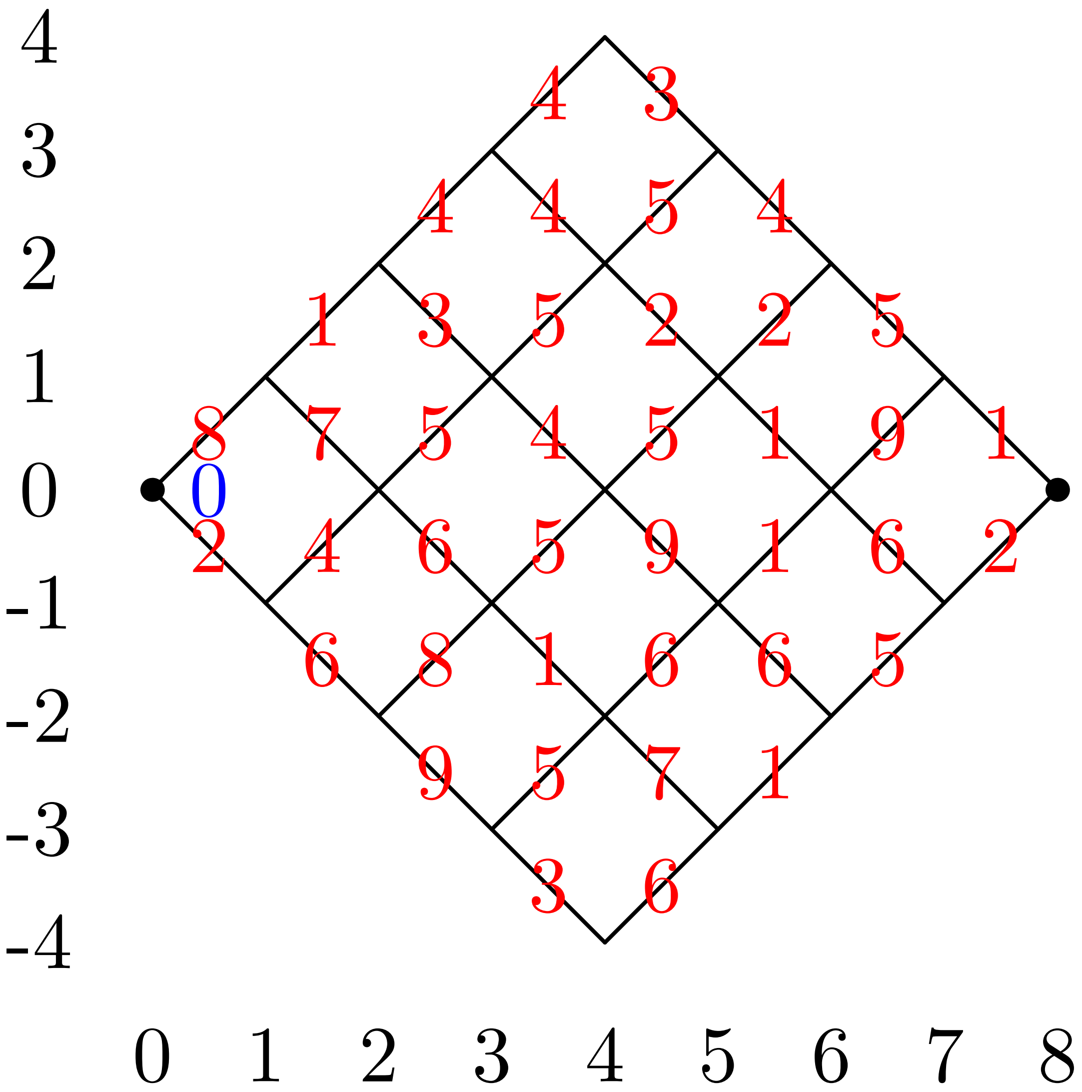
$$c_{(i+1,j)} = \min\bigl(c_{(i,j+1)} + w_{(i,j+1)(i+1,j)},\; c_{(i,j-1)} + w_{(i,j-1)(i+1,j)}\bigr)$$

- This is the defining equation in dynamic programming
- We have to treat the boundary sites specially, but this is just book-keeping

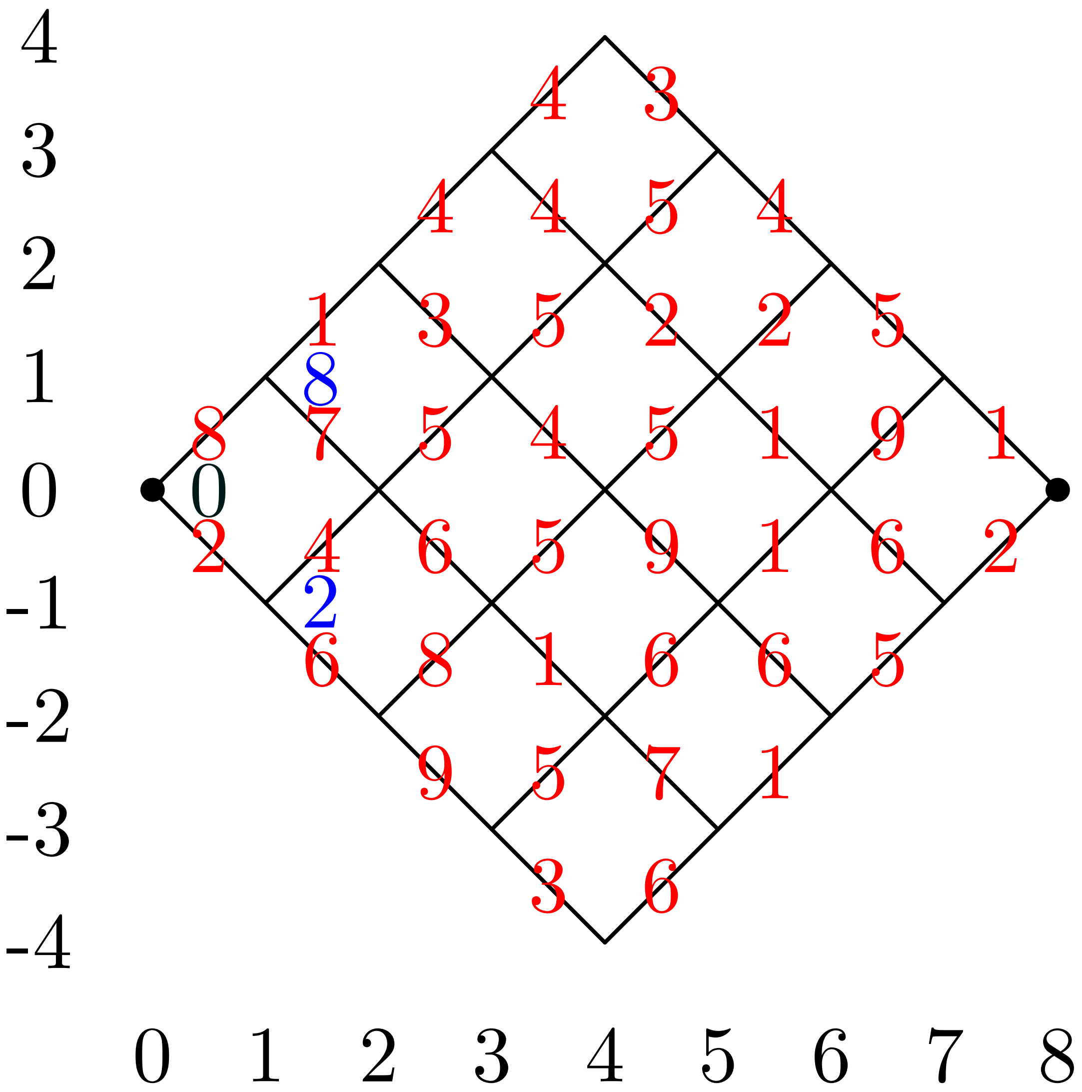
Example


4
3
2
1
0
-1
-2
-3
-4
0 1 2 3 4 5 6 7 8
4 3
4 4 5 4
1 3 5 2 2 5
8 7 5 4 5 1 9 1
2 4 6 5 9 1 6 2
6 8 1 6 6 5
9 5 7 1
3 6

Example

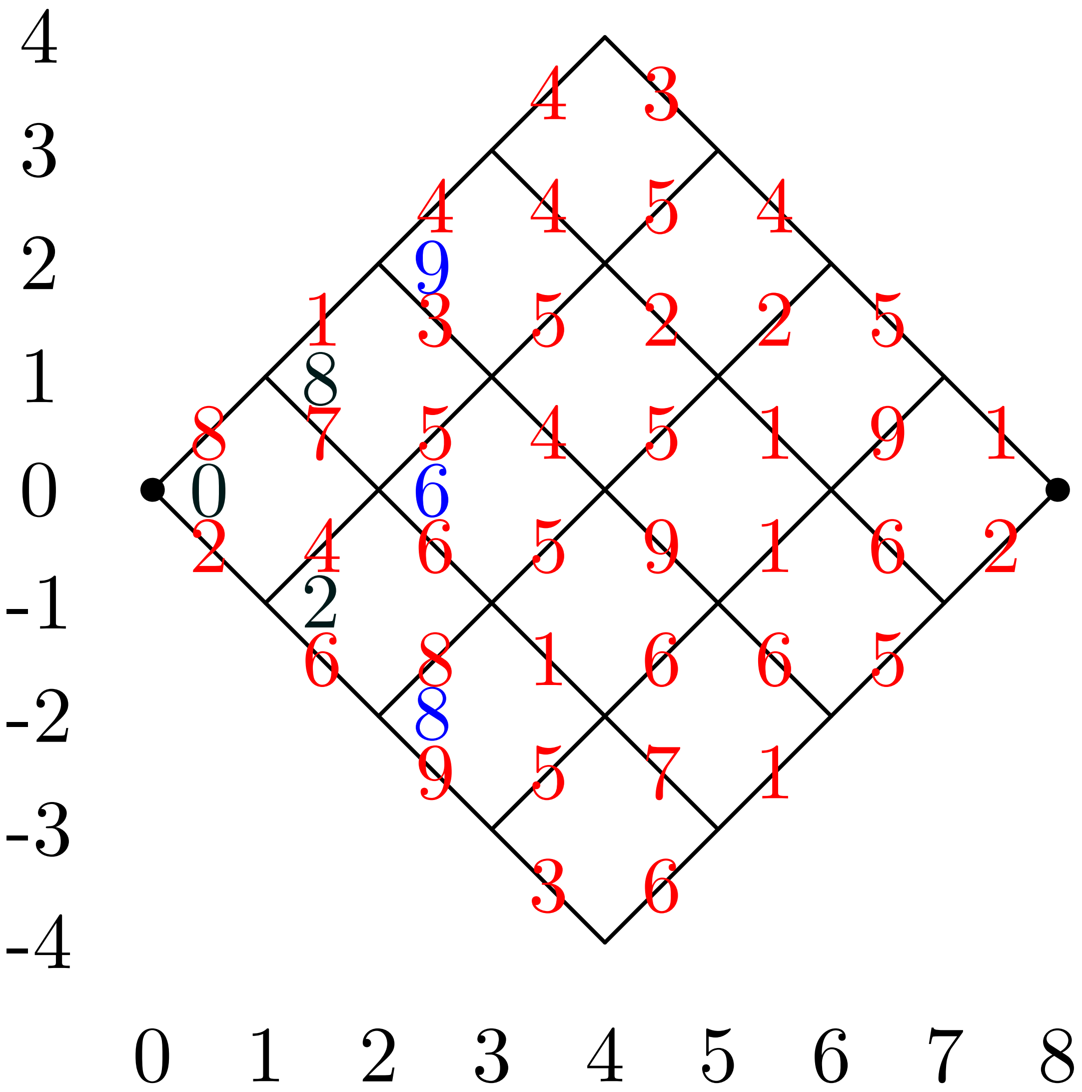

4
3
2
1
0
-1
-2
-3
-4
0 1 2 3 4 5 6 7 8
4 3
4 4 5 4
1 3 5 2 2 5
8 7 5 4 5 1 9 1
0
2 4 6 5 9 1 6 2
6 8 1 6 6 5
9 5 7 1
3 6

Example

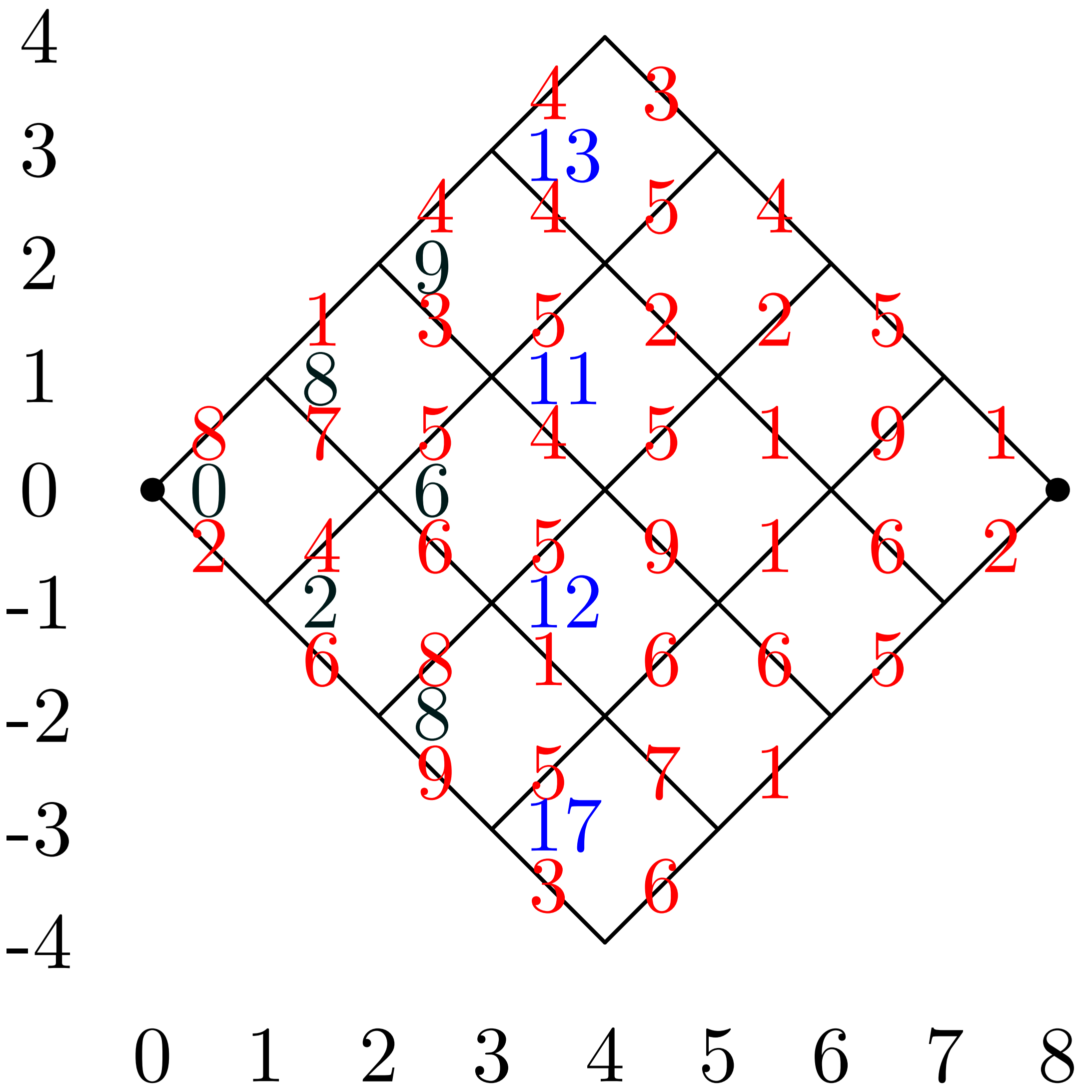


4
3
2
1
0
-1
-2
-3
-4
0 1 2 3 4 5 6 7 8
4 3
4 4 5 4
1 3 5 2 2 5
8
8 7 5 4 5 1 9 1
0
2 4 6 5 9 1 6 2
2
6 8 1 6 6 5
9 5 7 1
3 6

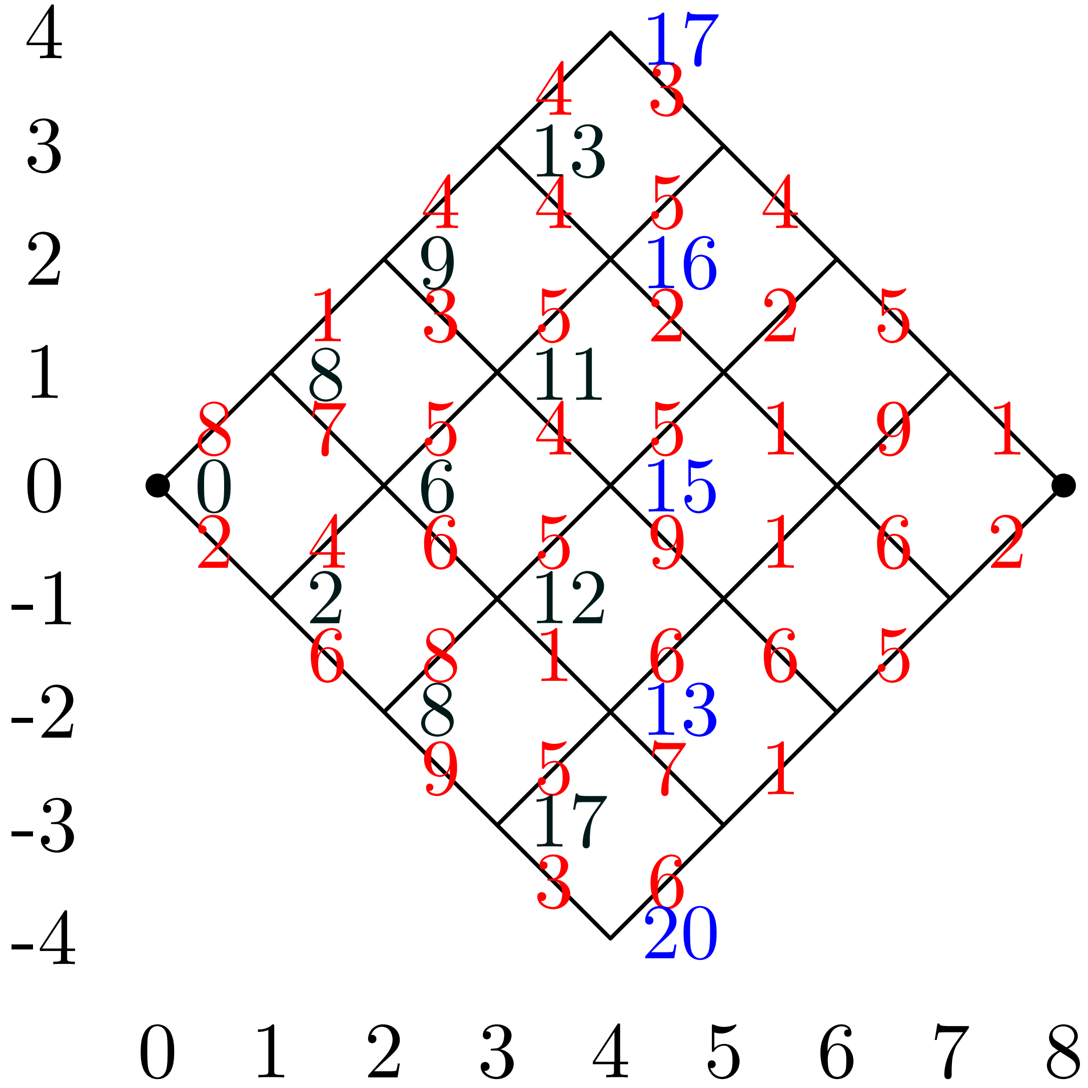
Example



Example

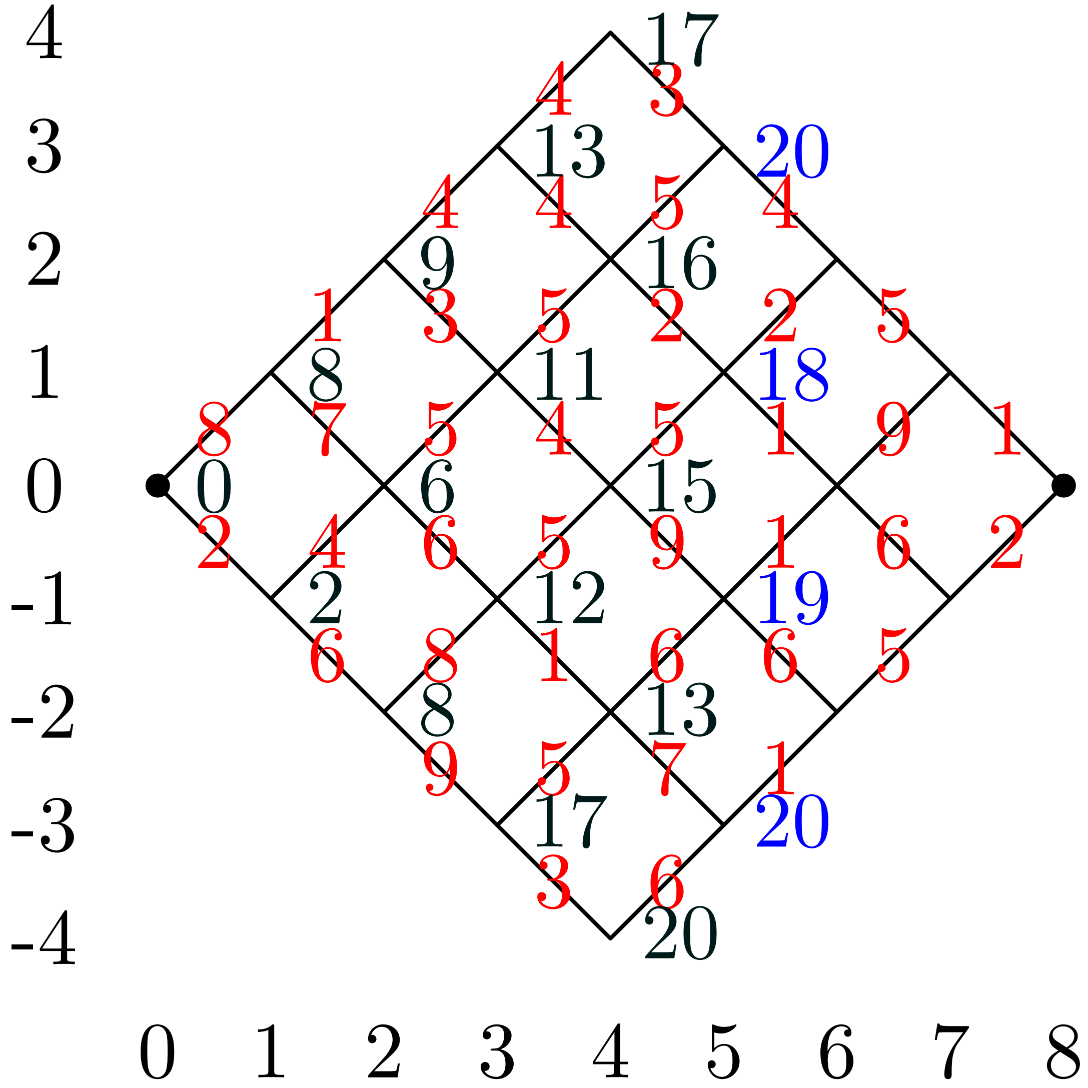


Example



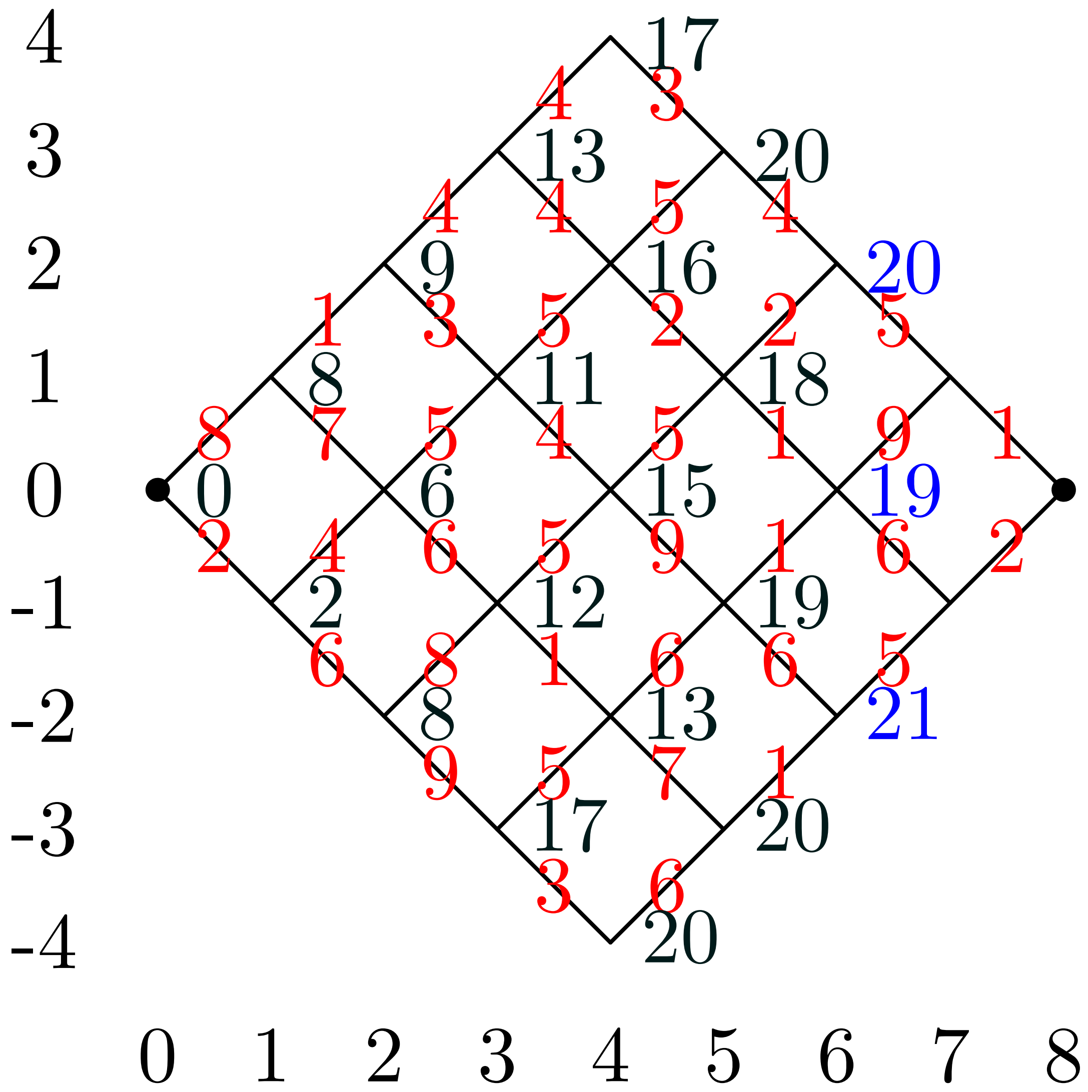
4
3
2
1
0
-1
-2
-3
-4
0 1 2 3 4 5 6 7 8
17
4 3
13
4 4 5 4
9 16
1 3 5 2 2 5
8 11
8 7 5 4 5 1 9 1
0 6 15
2 4 6 5 9 1 6 2
2 12
6 8 1 6 6 5
8 13
9 5 7 1
17
3 6
20

Example



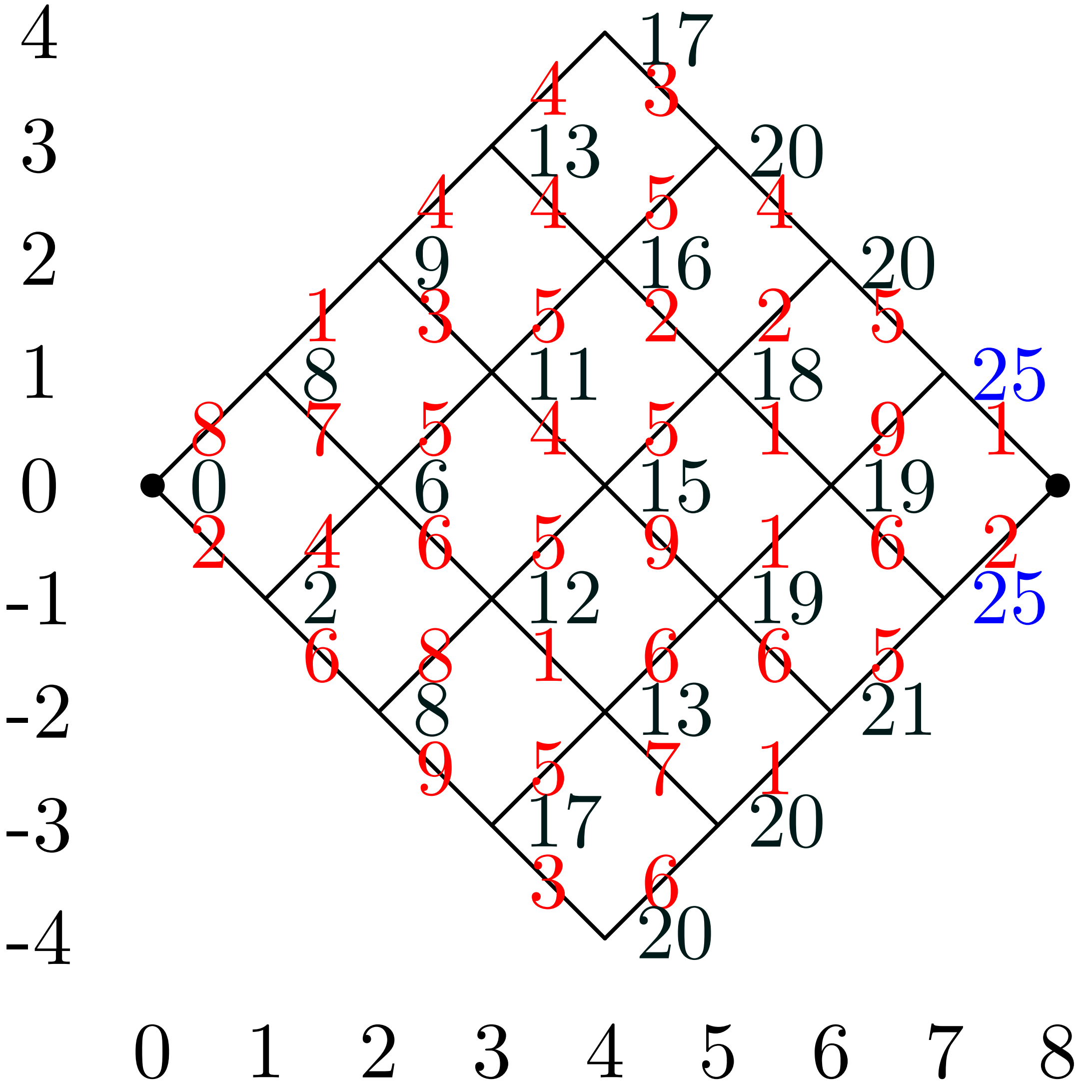
4
3
2
1
0
-1
-2
-3
-4
17
13
20
9
16
8
11
18
0
6
15
2
12
19
8
13
17
20
20
0 1 2 3 4 5 6 7 8

Example



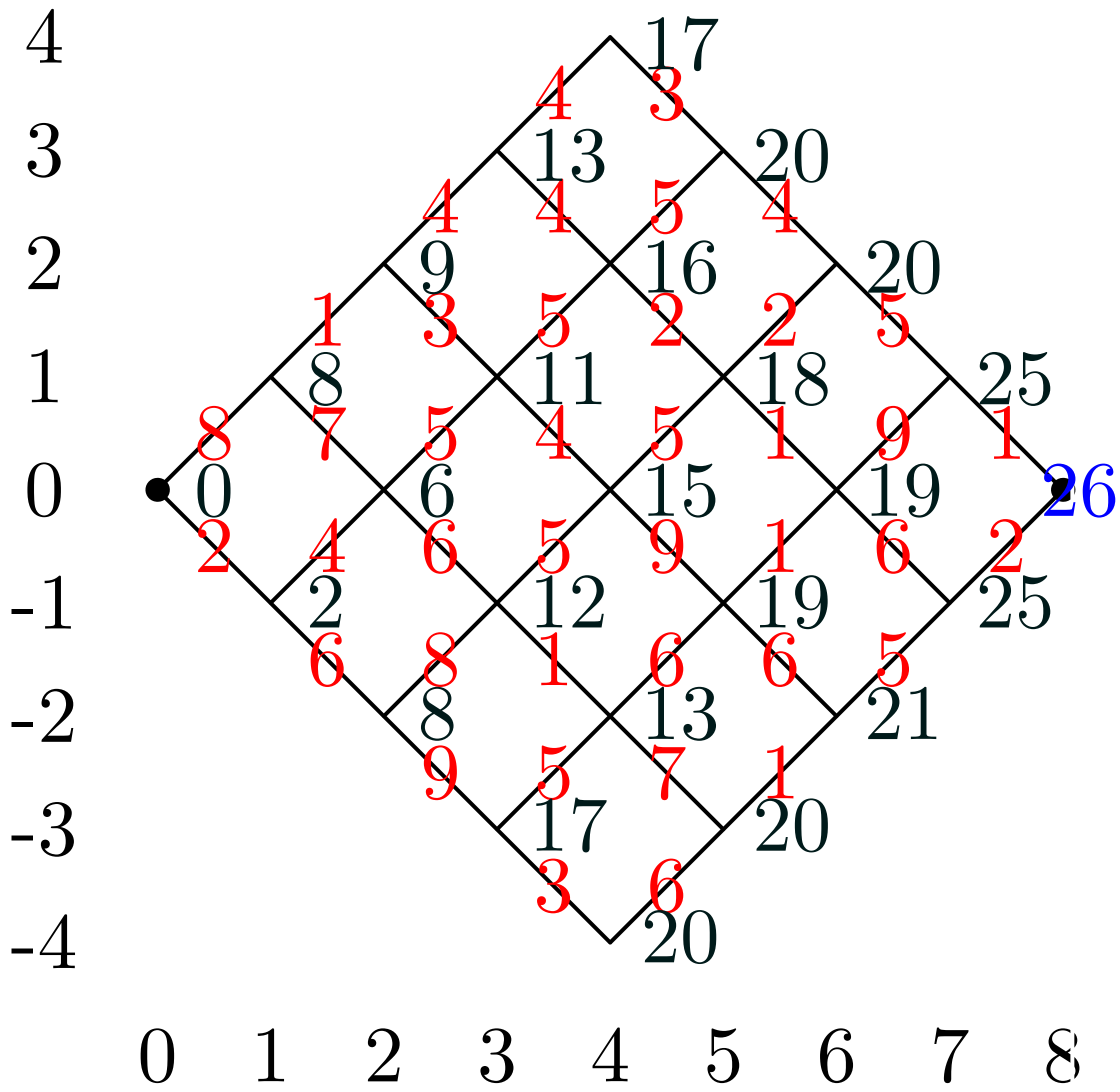
4
3
2
1
0
-1
-2
-3
-4
0 1 2 3 4 5 6 7 8
17
13
20
9
16
20
8
11
18
0
6
15
19
2
12
19
8
13
21
17
20
20

Example

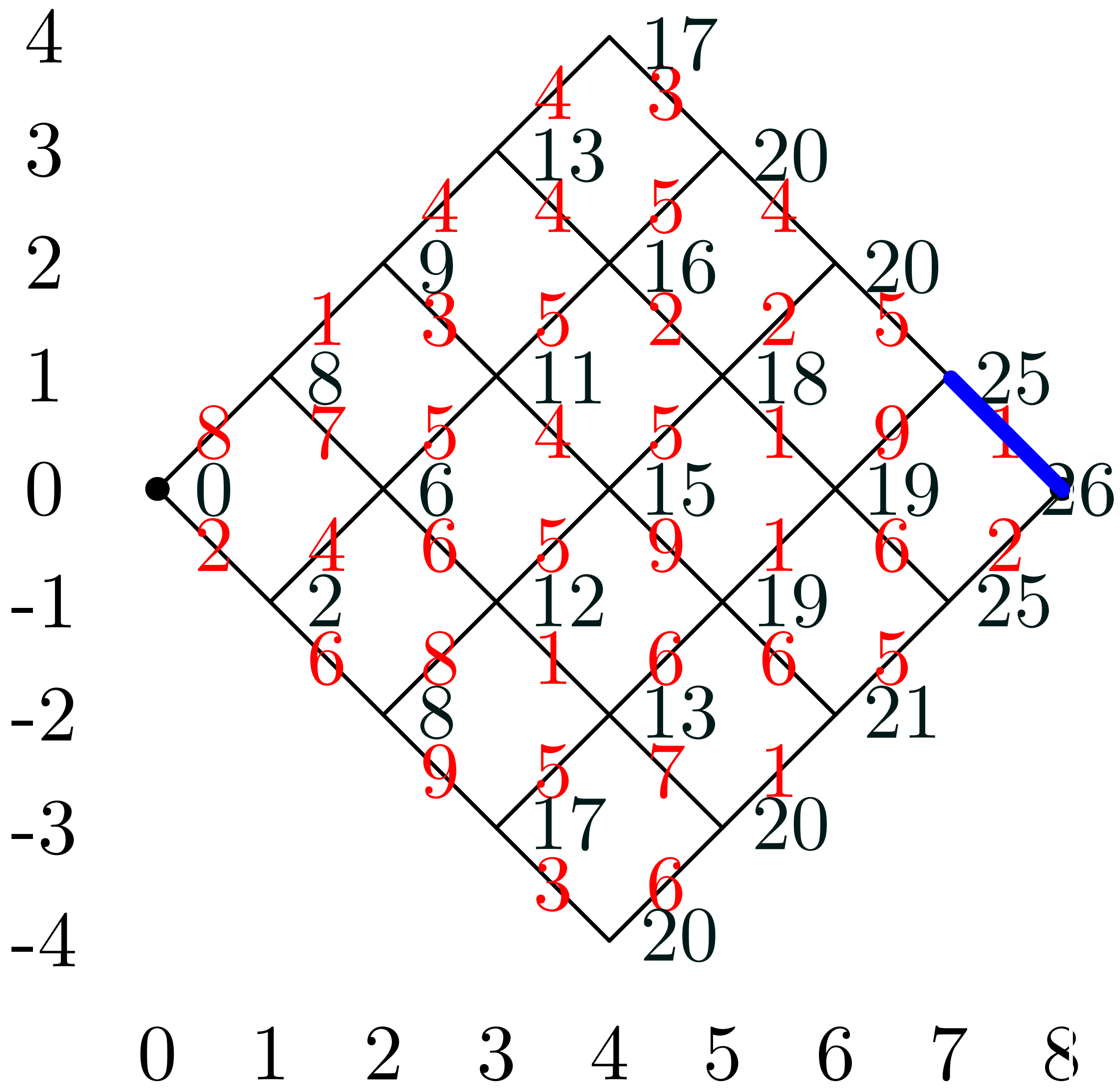


4
3
2
1
0
-1
-2
-3
-4
0 1 2 3 4 5 6 7 8
17
13 20
9 16 20
8 11 18 25
0 6 15 19
2 12 19 25
8 13 21
17 20
20

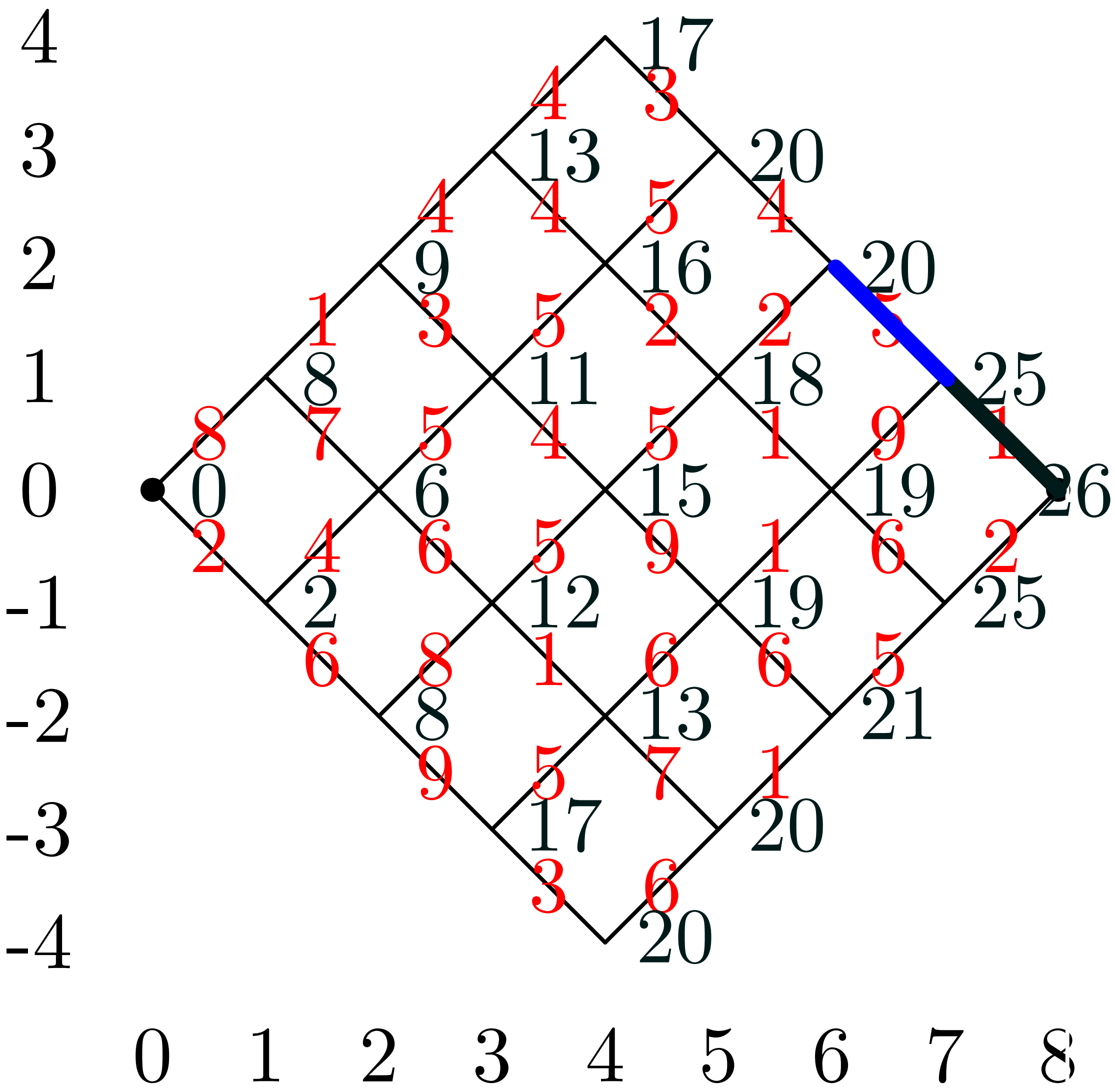
Example



Example

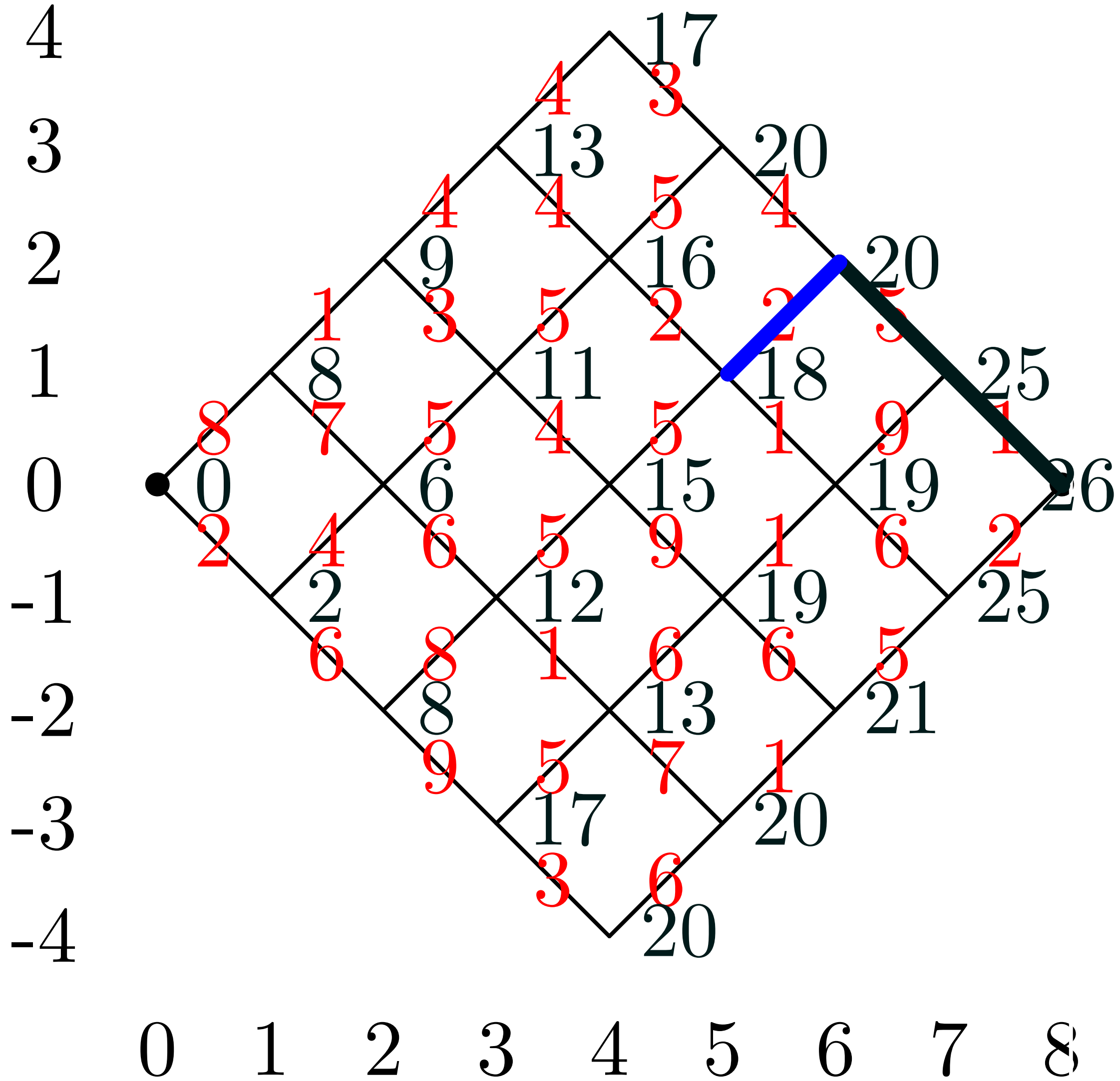


Example



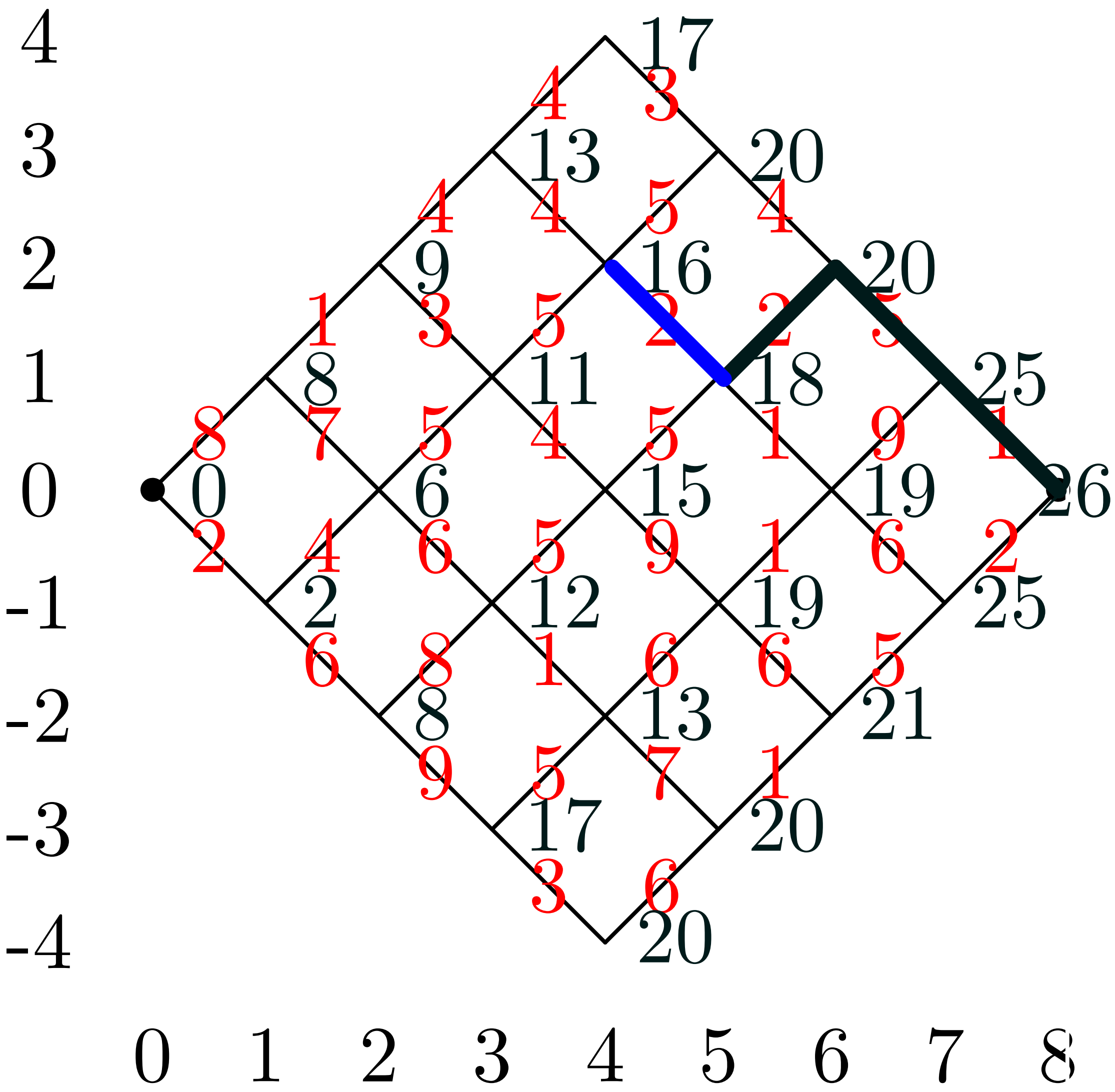
4
3
2
1
0
-1
-2
-3
-4
0 1 2 3 4 5 6 7 8
17
13
20
9
16
20
8
11
18
25
0
6
15
19
26
2
12
19
25
8
13
21
17
20
20

Example



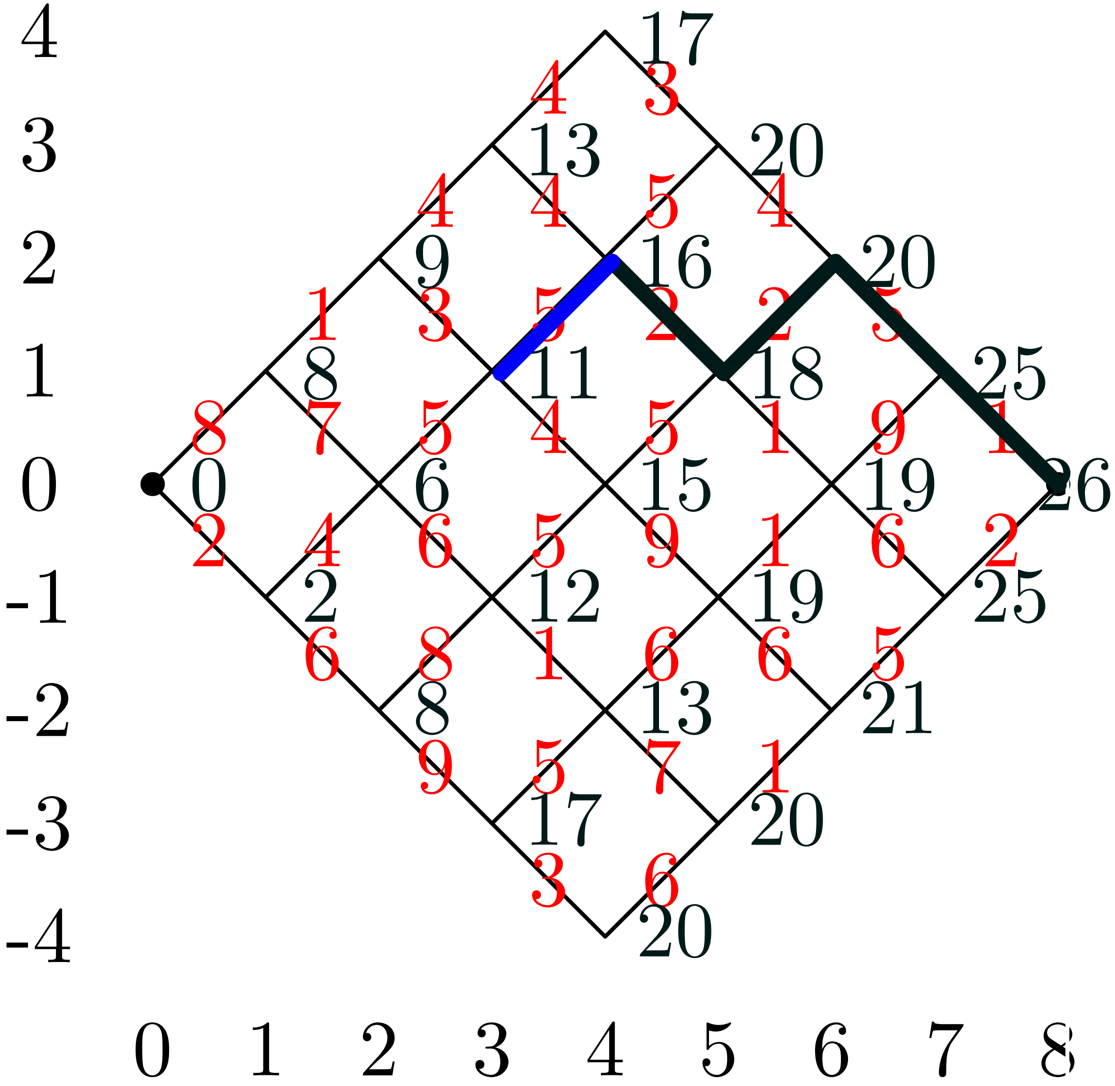
4
3
2
1
0
-1
-2
-3
-4
17
13
20
9
16
20
8
11
18
25
0
6
15
19
26
2
12
19
25
8
13
21
17
20
20
0 1 2 3 4 5 6 7 8

Example



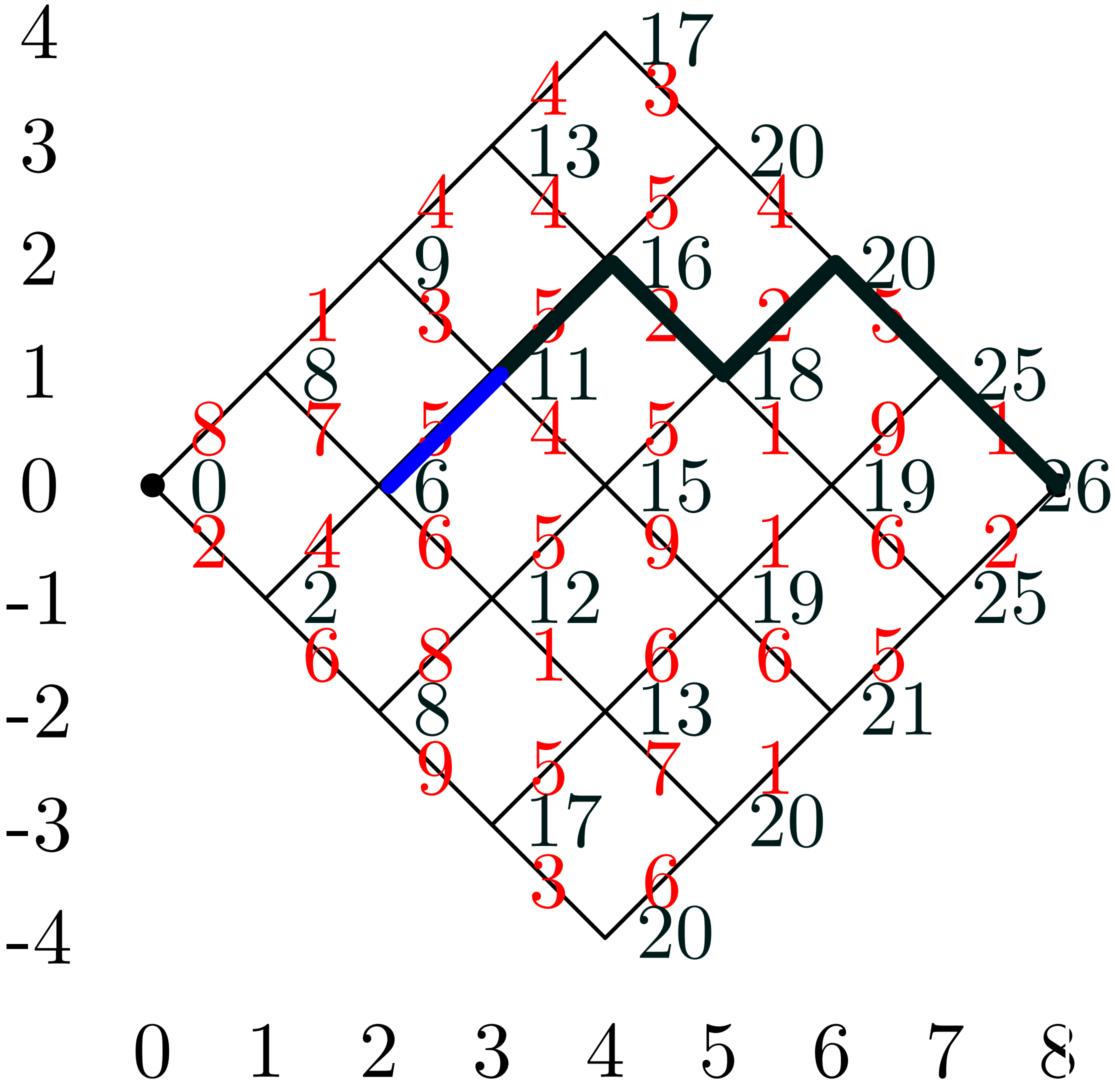
4
3
2
1
0
-1
-2
-3
-4
0 1 2 3 4 5 6 7 8
0
8
2
13
17
20
9
16
20
11
18
25
6
15
19
26
12
19
25
8
13
21
17
20
20

Example



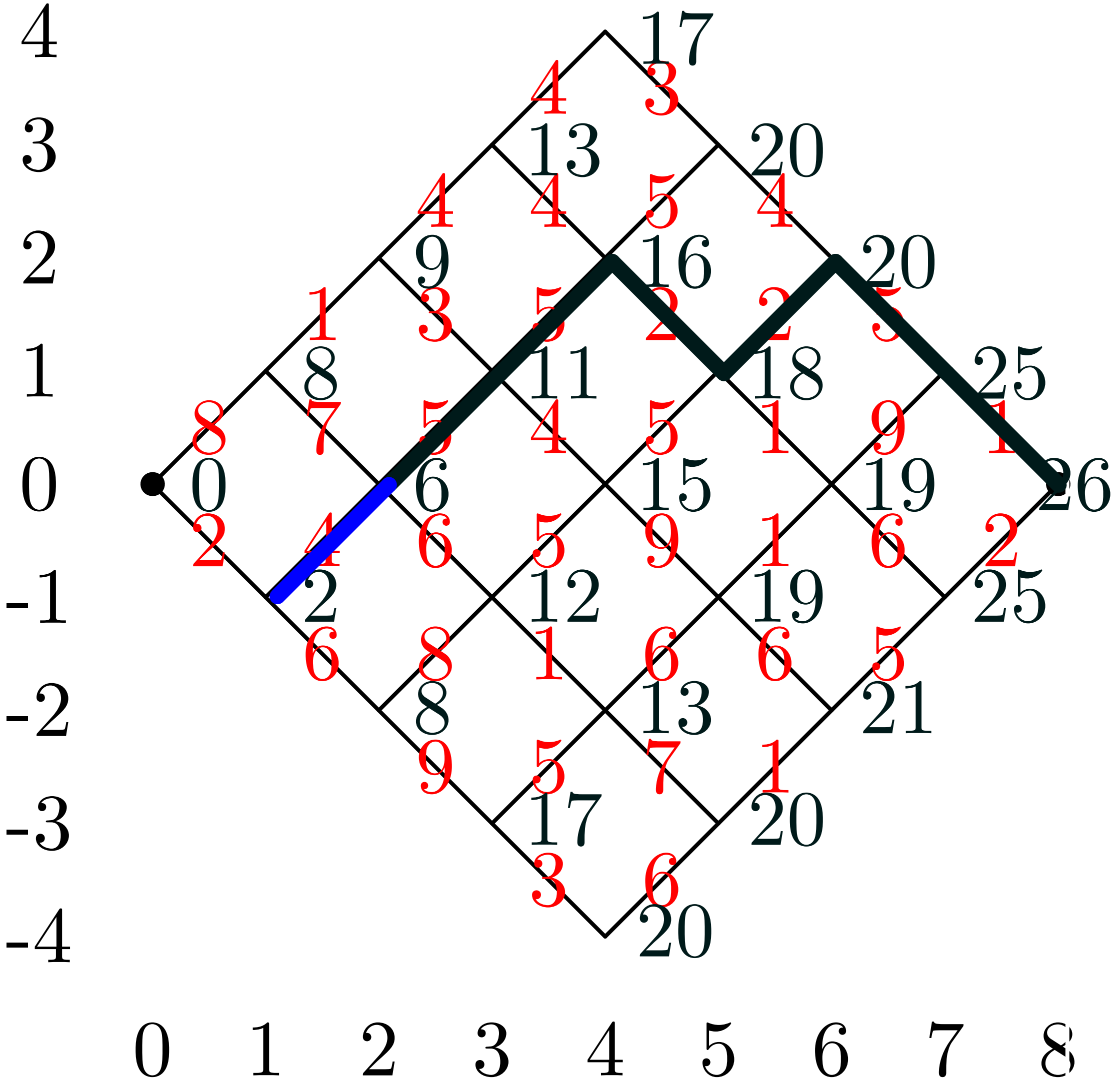
4
3
2
1
0
-1
-2
-3
-4
0 1 2 3 4 5 6 7 8
17
13
20
9
16
20
8
11
18
25
0
6
15
19
26
2
12
19
25
8
13
21
17
20
20

Example

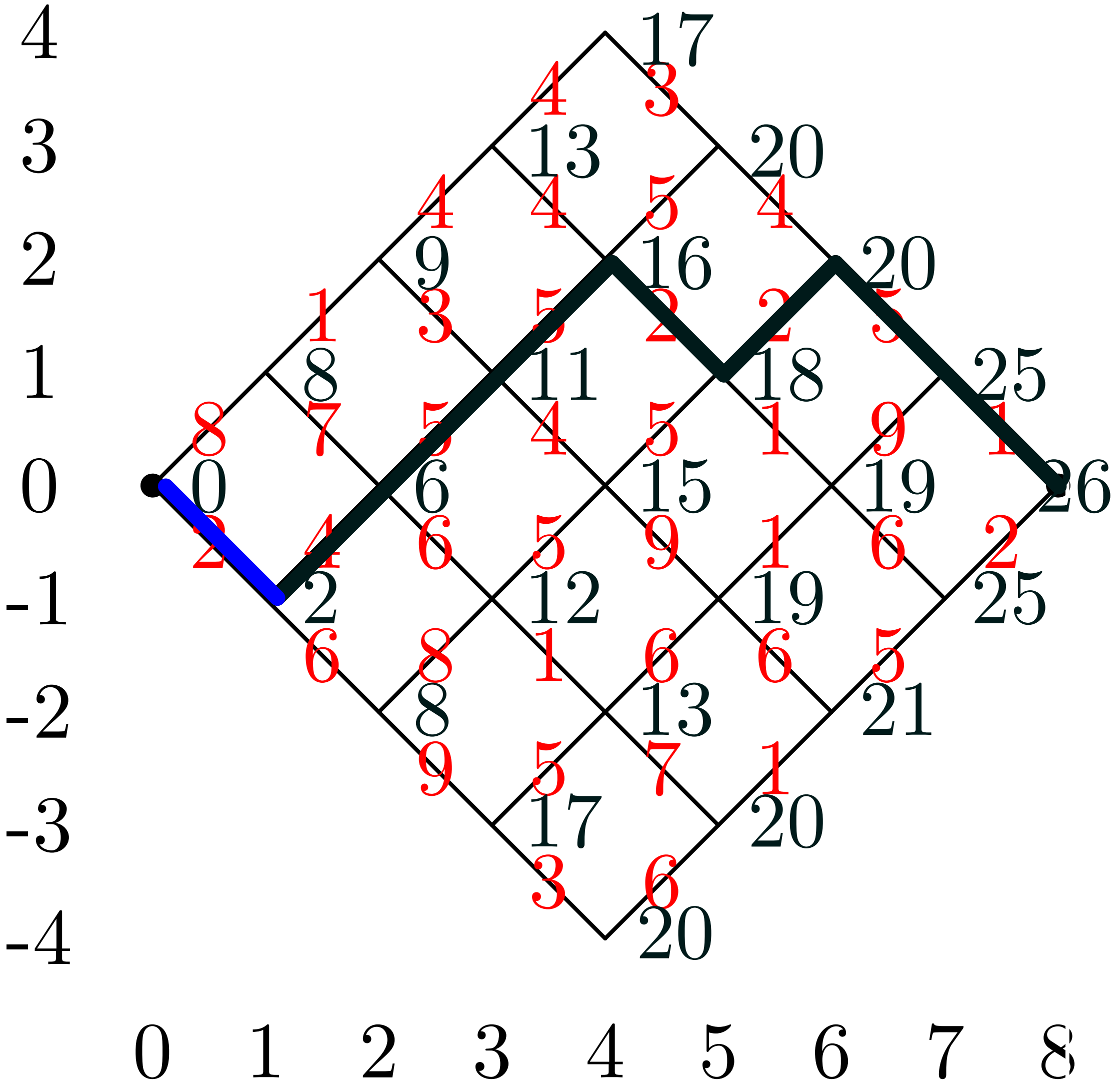


4
3
2
1
0
-1
-2
-3
-4
0 1 2 3 4 5 6 7 8
0
8
2
8
7
4
2
6
9
3
1
6
8
4
4
13
4
3
17
20
5
4
16
20
2
2
5
11
5
4
18
1
9
25
1
26
15
19
25
5
9
1
6
2
12
19
6
6
5
21
1
6
8
13
5
7
1
17
20
3
6
20

Example



Example



4
3
2
1
0
-1
-2
-3
-4
0 1 2 3 4 5 6 7 8
17
13
20
9
16
20
8
11
18
25
0
6
15
19
26
2
12
19
25
8
13
21
17
20
20

Backward Algorithm

- Having found the optimal costs $c_{(i,j)}$ we can find the optimal path starting from $(n, 0)$
- At each step we have a choice of going up or down
- We choose the direction which satisfies the constraint

$$c_{(i,j)} = c_{(i-1,j\pm1)} + w_{(i-1,j\pm1)(i,j)}$$

- If both directions satisfy the constraint we have more than one optimal path

Backward Algorithm

- Having found the optimal costs $c_{(i,j)}$ we can find the optimal path starting from $(n, 0)$
- At each step we have a choice of going up or down
- We choose the direction which satisfies the constraint

$$c_{(i,j)} = c_{(i-1,j\pm1)} + w_{(i-1,j\pm1)(i,j)}$$

- If both directions satisfy the constraint we have more than one optimal path

Backward Algorithm

- Having found the optimal costs $c_{(i,j)}$ we can find the optimal path starting from $(n, 0)$
- At each step we have a choice of going up or down
- We choose the direction which satisfies the constraint

$$c_{(i,j)} = c_{(i-1,j\pm 1)} + w_{(i-1,j\pm 1)(i,j)}$$

- If both directions satisfy the constraint we have more than one optimal path

Backward Algorithm

- Having found the optimal costs $c_{(i,j)}$ we can find the optimal path starting from $(n, 0)$
- At each step we have a choice of going up or down
- We choose the direction which satisfies the constraint

$$c_{(i,j)} = c_{(i-1,j\pm1)} + w_{(i-1,j\pm1)(i,j)}$$

- If both directions satisfy the constraint we have more than one optimal path

Time Complexity

- In our dynamic programming solution we had to compute the cost $c_{(i,j)}$ at each lattice point
- There were $(n+1)^2$ lattice point
- It took constant time to compute each cost so the total time to perform the forward algorithm was $\Theta(n^2)$
- The time complexity of the backward algorithm was $\Theta(n)$
- This compares with $\exp(\Theta(n))$ for the brute force algorithm

Time Complexity

- In our dynamic programming solution we had to compute the cost $c_{(i,j)}$ at each lattice point
- There were $(n+1)^2$ lattice point
- It took constant time to compute each cost so the total time to perform the forward algorithm was $\Theta(n^2)$
- The time complexity of the backward algorithm was $\Theta(n)$
- This compares with $\exp(\Theta(n))$ for the brute force algorithm

Time Complexity

- In our dynamic programming solution we had to compute the cost $c_{(i,j)}$ at each lattice point
- There were $(n+1)^2$ lattice point
- It took constant time to compute each cost so the total time to perform the forward algorithm was $\Theta(n^2)$
- The time complexity of the backward algorithm was $\Theta(n)$
- This compares with $\exp(\Theta(n))$ for the brute force algorithm

Time Complexity

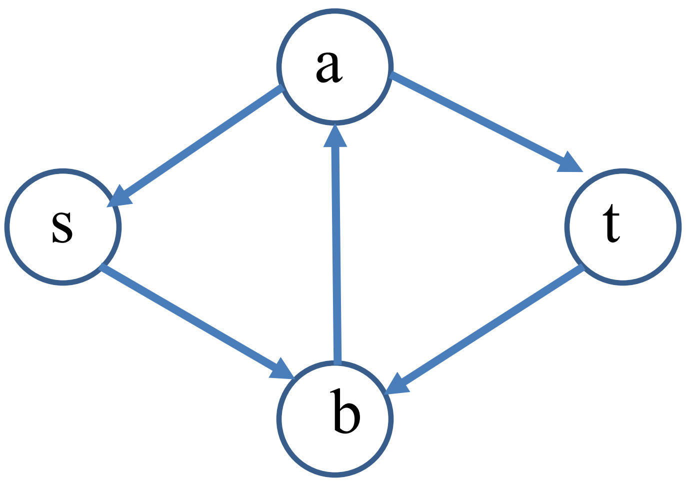
- In our dynamic programming solution we had to compute the cost $c_{(i,j)}$ at each lattice point
- There were $(n+1)^2$ lattice point
- It took constant time to compute each cost so the total time to perform the forward algorithm was $\Theta(n^2)$
- The time complexity of the backward algorithm was $\Theta(n)$
- This compares with $\exp(\Theta(n))$ for the brute force algorithm

Time Complexity

- In our dynamic programming solution we had to compute the cost $c_{(i,j)}$ at each lattice point
- There were $(n+1)^2$ lattice point
- It took constant time to compute each cost so the total time to perform the forward algorithm was $\Theta(n^2)$
- The time complexity of the backward algorithm was $\Theta(n)$
- This compares with $\exp(\Theta(n))$ for the brute force algorithm

Does DP always work?

- Cycle matters!



$$c(s, t) = c(s, a) + c(a, t)$$

$$c(s, a) = c(s, b) + c(b, a)$$

$$c(s, b) = c(s, b) \text{ or } c(s, t) + c(t, b)$$

- Directed acyclic graphs (DAGs) are fine.
- Still, not all problems can be split neatly to sub-problems.
- And, very often, it needs a careful design.