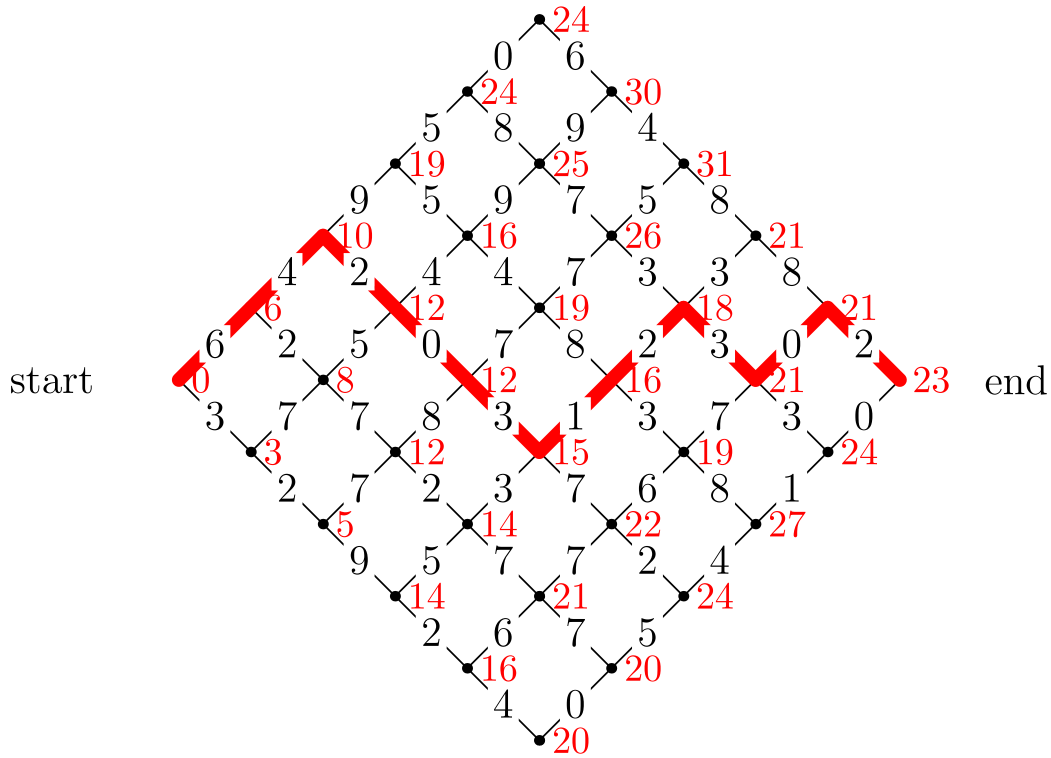
Algorithms and Analysis

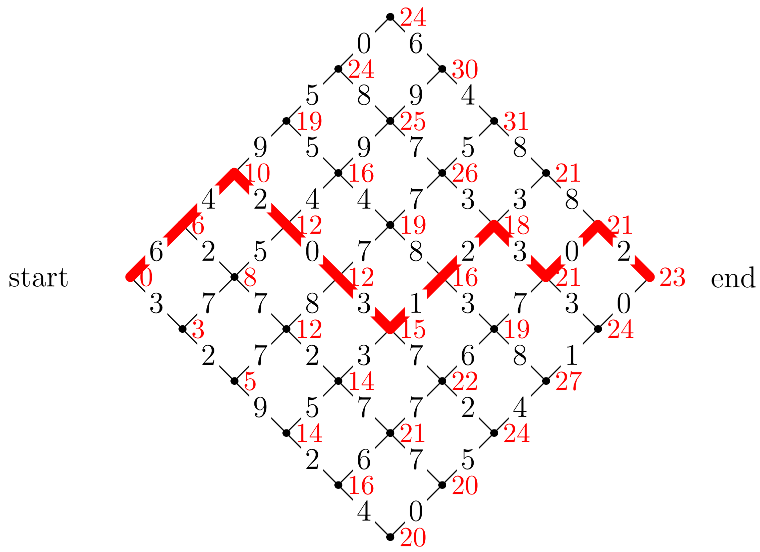
Lesson 23: *Dynamic Programming*



Dynamic programming, line breaking, edit distance, Dijkstra, TSP

Outline

1. **Dynamic Programming**
2. Applications
   - Line Breaks
   - Edit Distance
   - Dijkstra's Algorithm
3. Limitation



Dynamic Programming

- One of the most powerful strategies for solving optimisation problems is **dynamic programming**
  - ⋆ Build a set of optimal partial solutions
  - ⋆ Increase the size of the partial solutions until you have a full solution
  - ⋆ Each step uses the set of optimal partial solutions found in the previous step
- Developed by Richard Bellman in the early 1950's
- The name is unfortunate as it doesn't have much to do with programming

Dynamic Programming

- One of the most powerful strategies for solving optimisation problems is **dynamic programming**
  - ⋆ Build a set of optimal partial solutions
  - ⋆ Increase the size of the partial solutions until you have a full solution
  - ⋆ Each step uses the set of optimal partial solutions found in the previous step
- Developed by Richard Bellman in the early 1950's
- The name is unfortunate as it doesn't have much to do with programming

Dynamic Programming

- One of the most powerful strategies for solving optimisation problems is **dynamic programming**
  - ⋆ Build a set of optimal partial solutions
  - ⋆ Increase the size of the partial solutions until you have a full solution
  - ⋆ Each step uses the set of optimal partial solutions found in the previous step
- Developed by Richard Bellman in the early 1950's
- The name is unfortunate as it doesn't have much to do with programming

Dynamic Programming

- One of the most powerful strategies for solving optimisation problems is **dynamic programming**
  - ⋆ Build a set of optimal partial solutions
  - ⋆ Increase the size of the partial solutions until you have a full solution
  - ⋆ Each step uses the set of optimal partial solutions found in the previous step
- Developed by Richard Bellman in the early 1950's
- The name is unfortunate as it doesn't have much to do with programming

Dynamic Programming

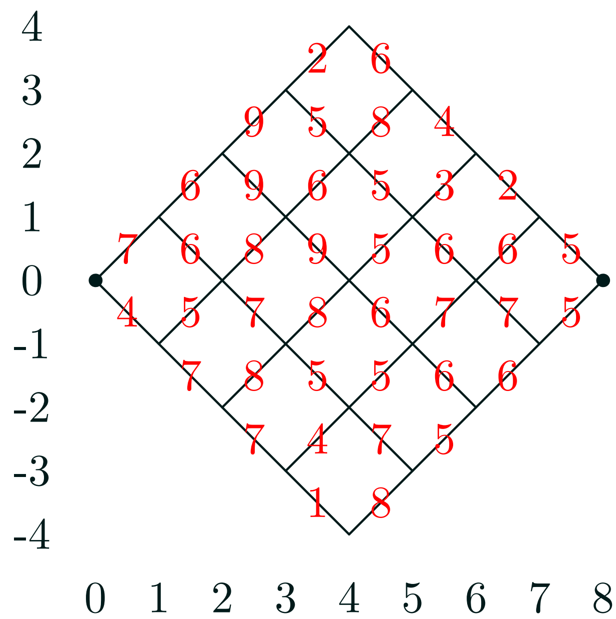
- One of the most powerful strategies for solving optimisation problems is **dynamic programming**
  - ⋆ Build a set of optimal partial solutions
  - ⋆ Increase the size of the partial solutions until you have a full solution
  - ⋆ Each step uses the set of optimal partial solutions found in the previous step
- Developed by Richard Bellman in the early 1950's
- The name is unfortunate as it doesn't have much to do with programming

Dynamic Programming

- One of the most powerful strategies for solving optimisation problems is **dynamic programming**
  - ⋆ Build a set of optimal partial solutions
  - ⋆ Increase the size of the partial solutions until you have a full solution
  - ⋆ Each step uses the set of optimal partial solutions found in the previous step
- Developed by Richard Bellman in the early 1950's
- The name is unfortunate as it doesn't have much to do with programming

A Toy Problem

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

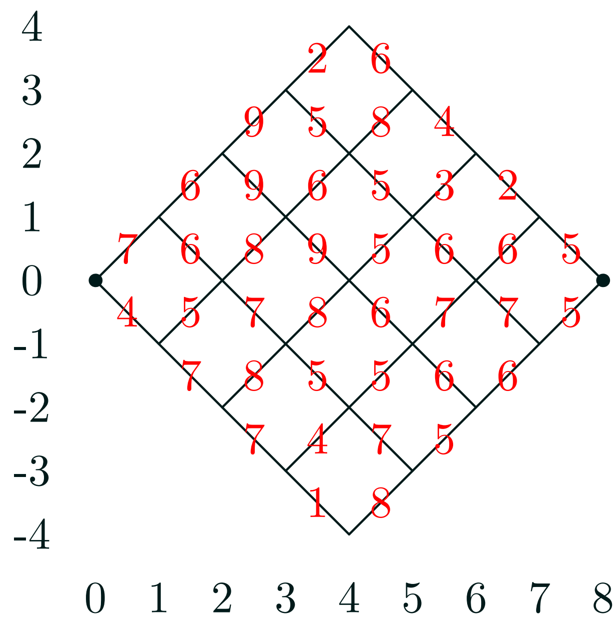


- The costs of traversing each link is shown in red

- 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

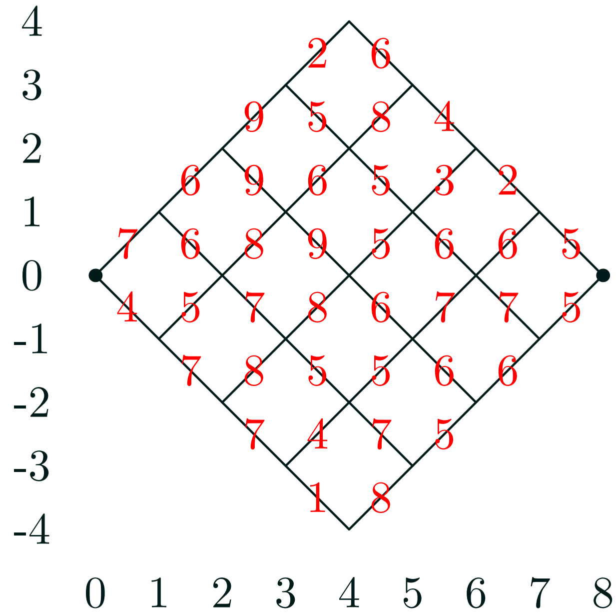


- The costs of traversing each link is shown in red

- 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

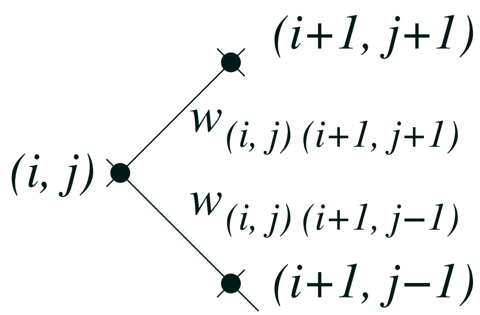
- The obvious brute force strategy is to try every path
- For a problem with n steps we require $n/2$ to be diagonally up and $n/2$ to be diagonally down
- The total number of paths is

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

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



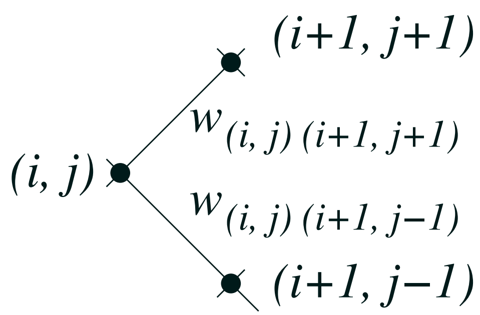
- Clearly $c_{(0,0)} = 0$

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)

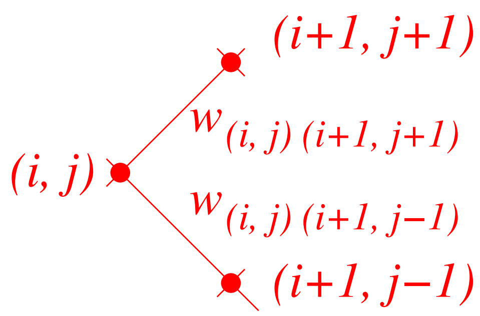
- We denote the weights between two points on the lattice by $w_{(i,j)(i+1,j\pm1)}$



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

Building a Solution

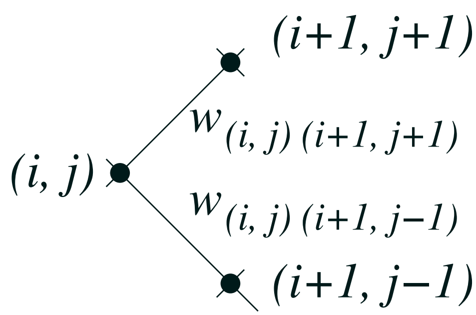
- We can solve this problem efficiently using dynamic programming by considering optimal paths of shorter length
- Let $c_{(i,j)}$ denote the cost of the optimal path to node (i, j)
- We denote the weights between two points on the lattice by $w_{(i,j)(i+1,j\pm 1)}$



- 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

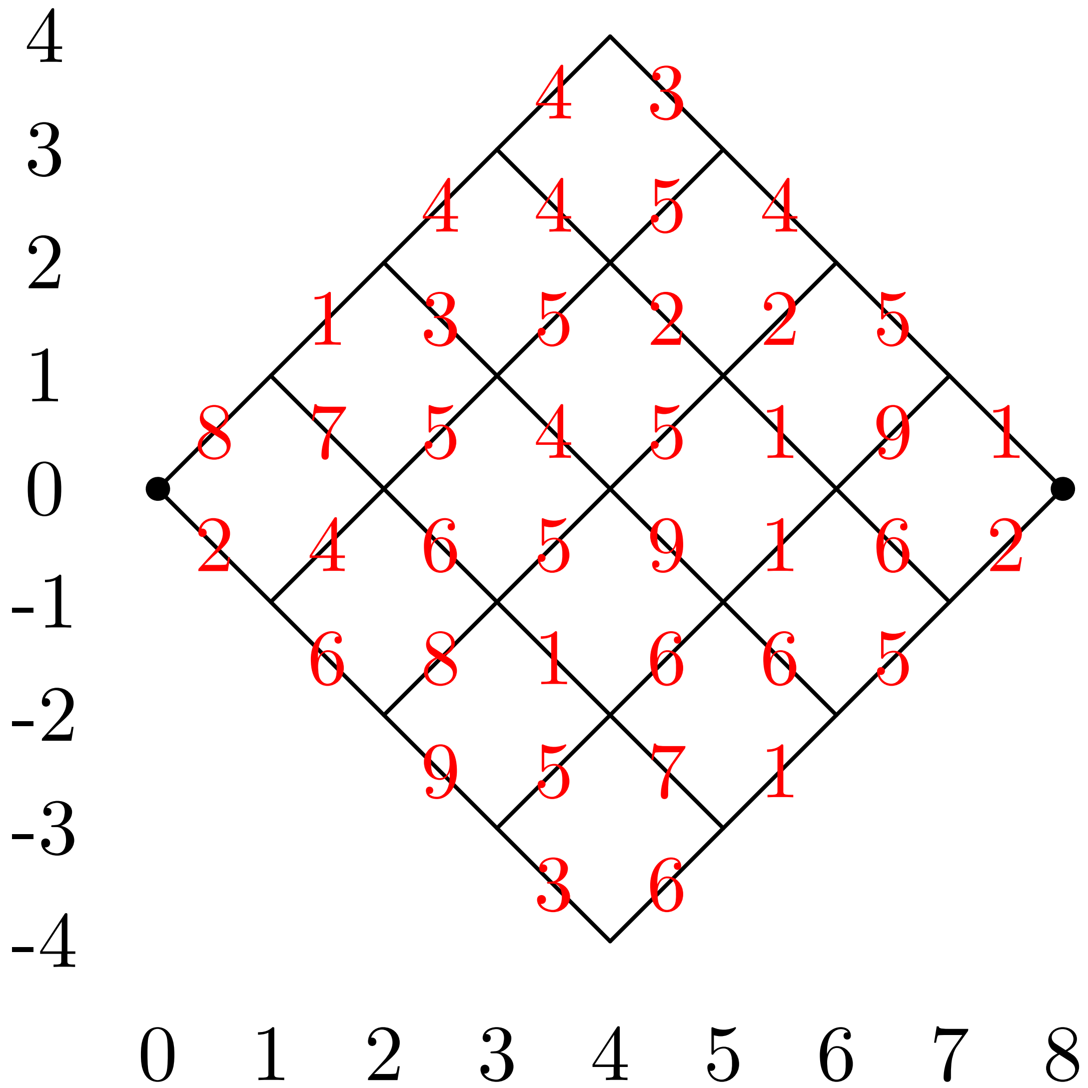
Forward Algorithm

- 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

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

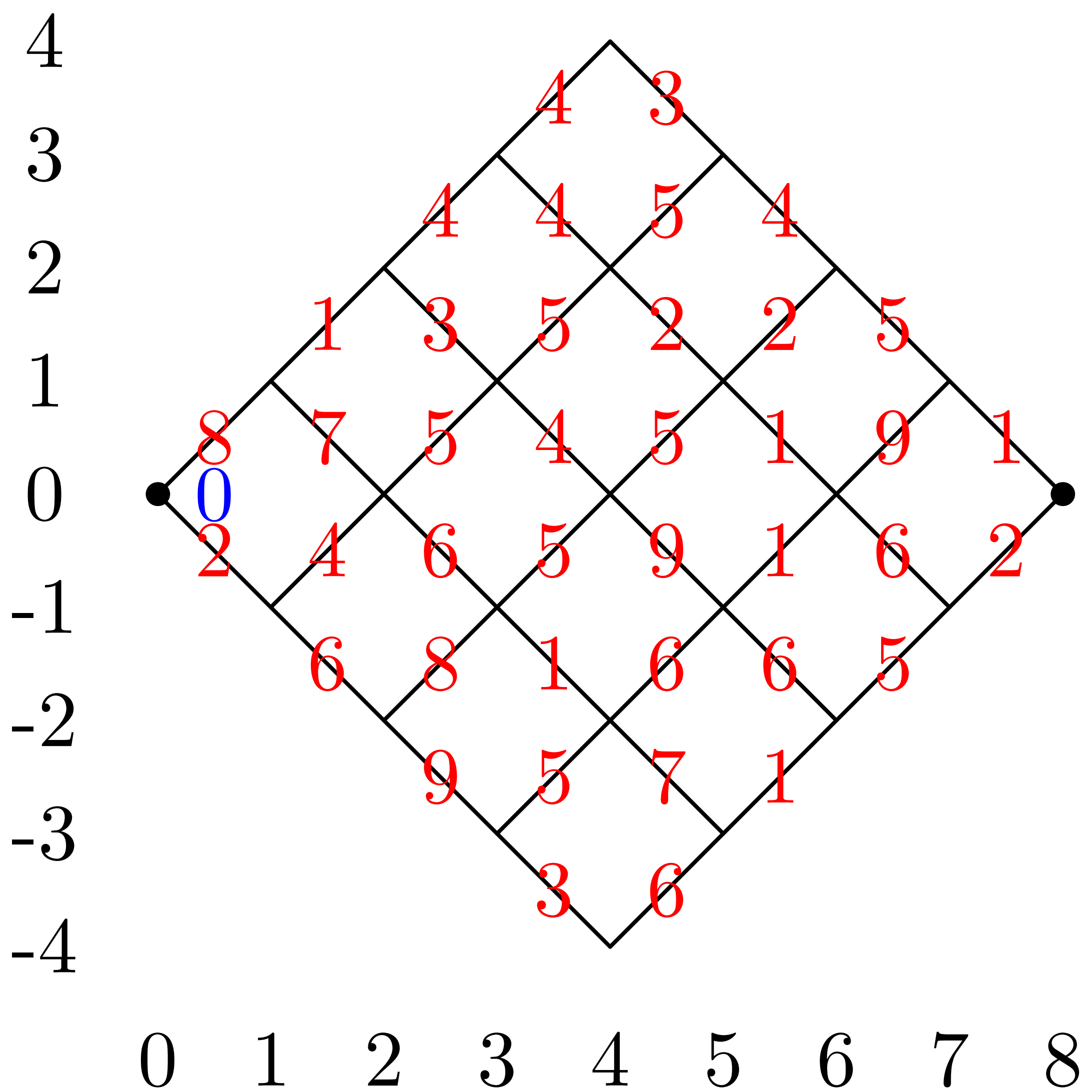
- This is the defining equation in dynamic programming
- We have to treat the boundary sites specially, but this is just book-keeping

Example



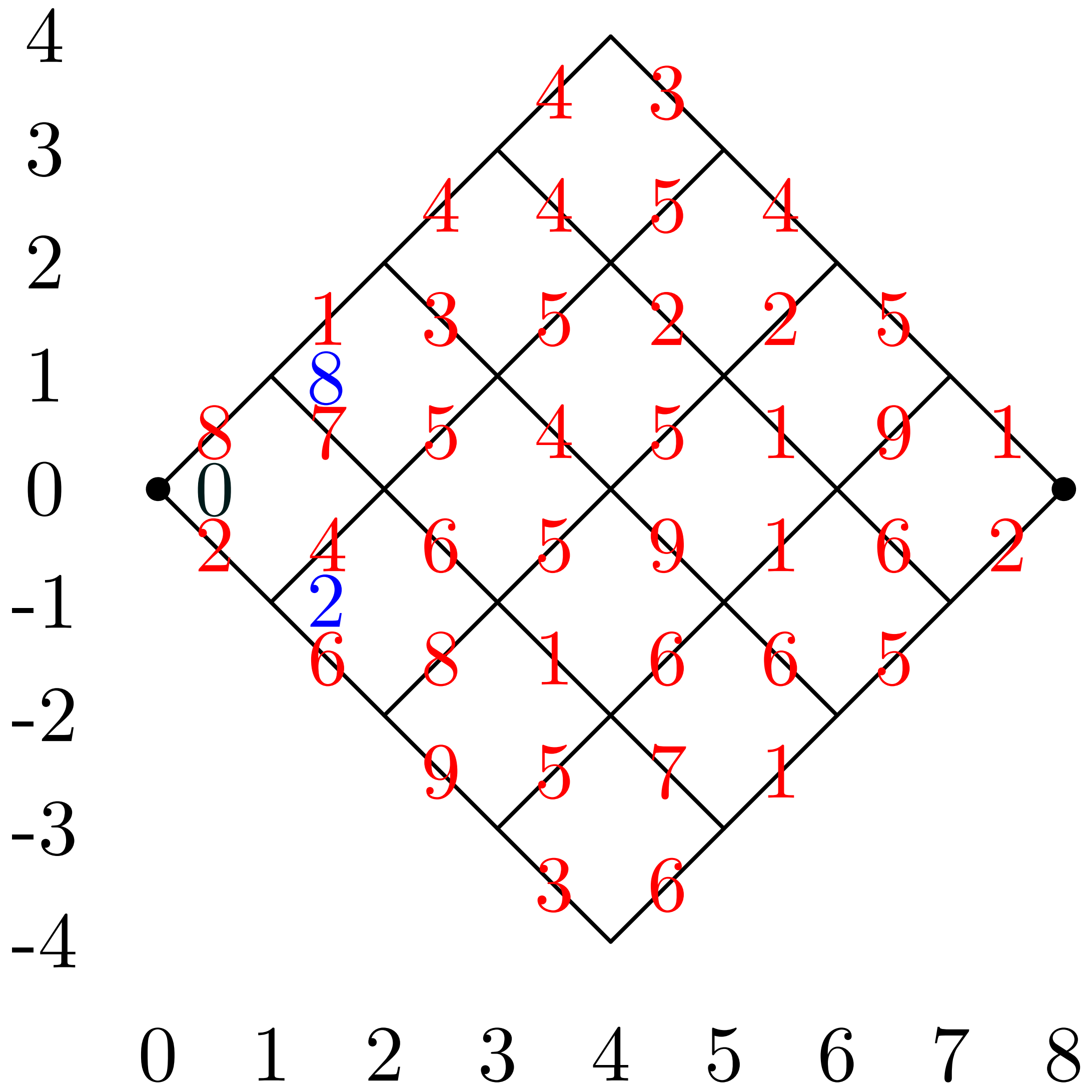
4
3
2
1
0
-1
-2
-3
-4
4 3
4 4 5 4
1 3 5 2 2 5
8 7 5 4 5 1 9 1
2 4 6 5 9 1 6 2
6 8 1 6 6 5
9 5 7 1
3 6
0 1 2 3 4 5 6 7 8

Example

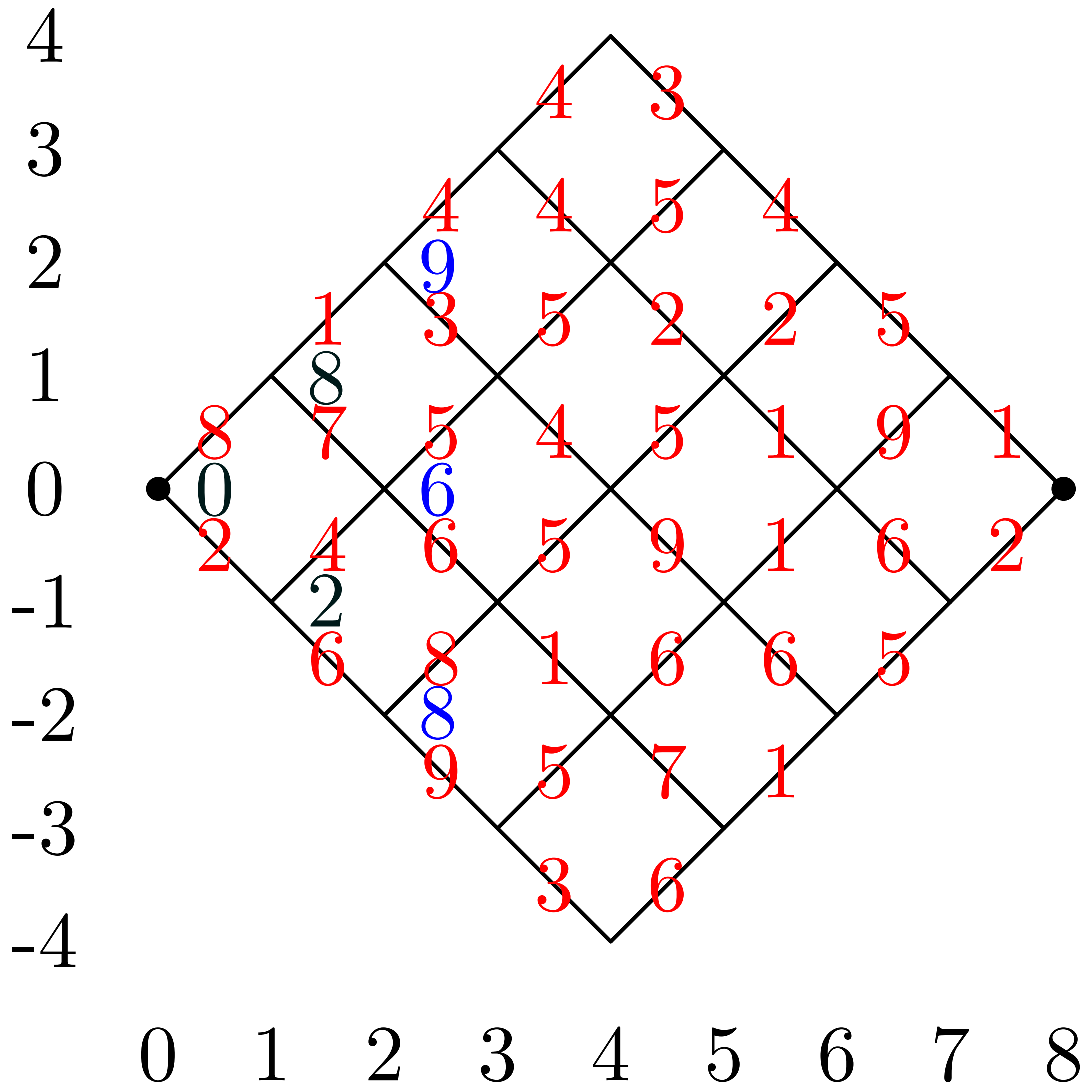


4
3
2
1
0
-1
-2
-3
-4
4 3
4 4 5 4
1 3 5 2 2 5
8 7 5 4 5 1 9 1
0
2 4 6 5 9 1 6 2
6 8 1 6 6 5
9 5 7 1
3 6
0 1 2 3 4 5 6 7 8

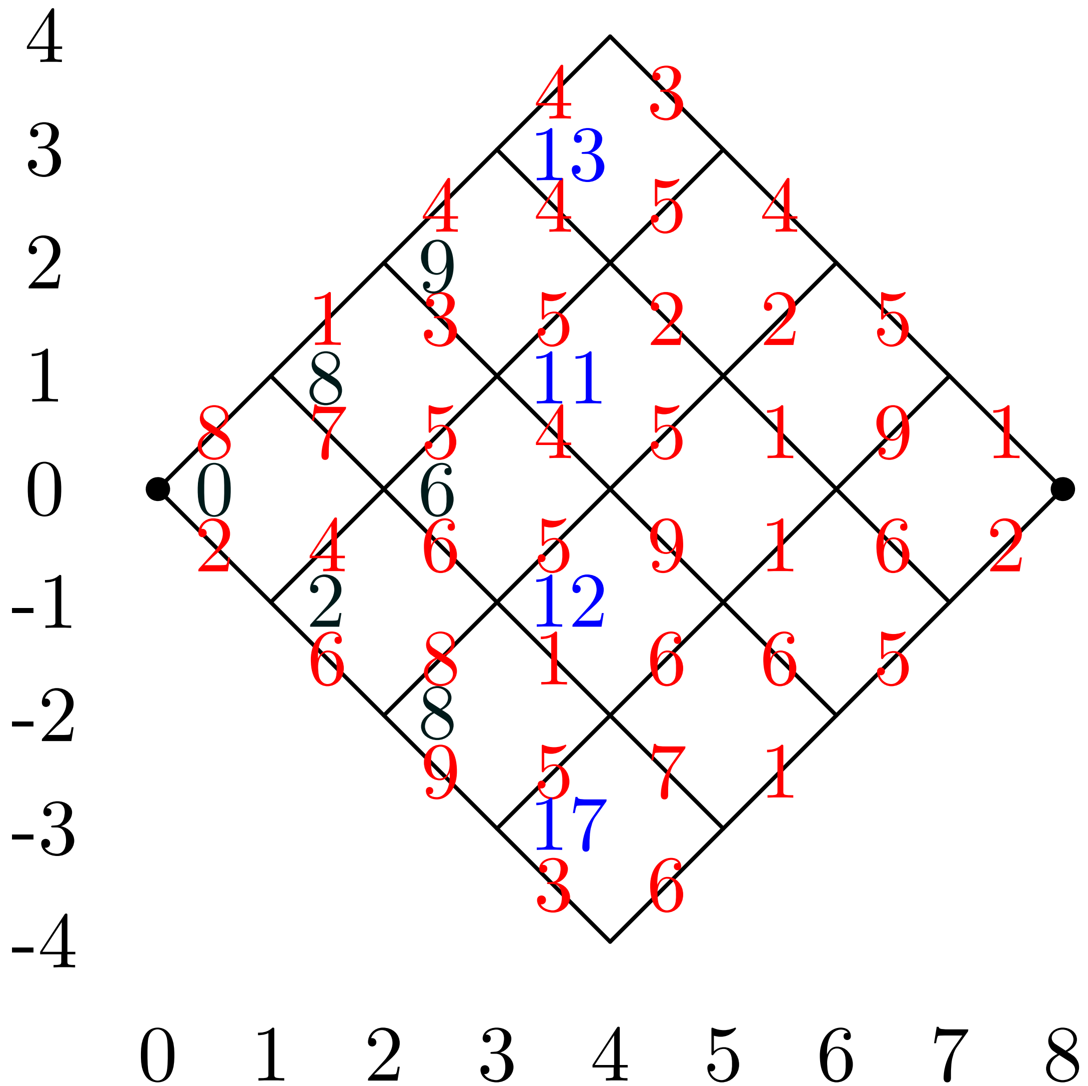
Example



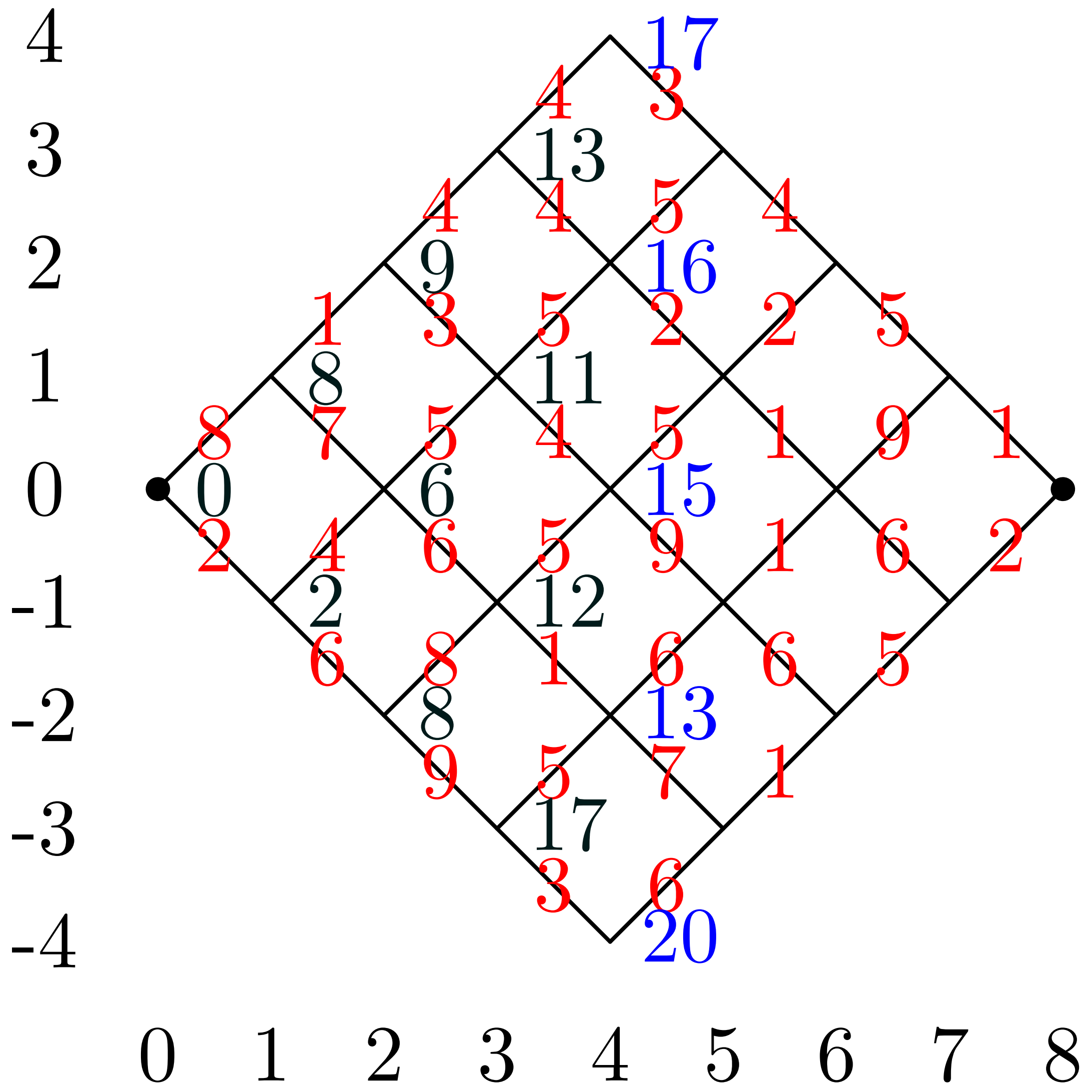
Example



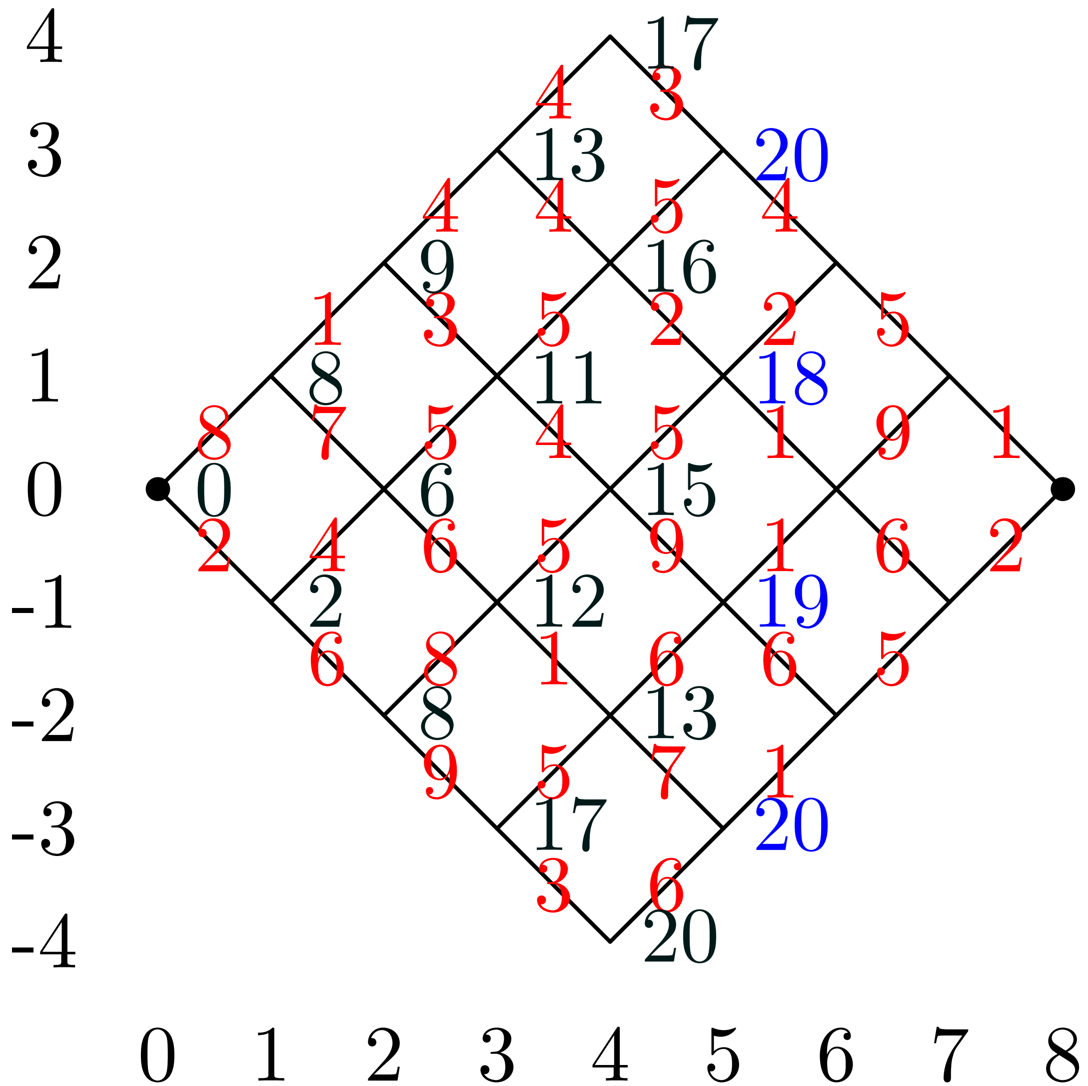
Example



Example

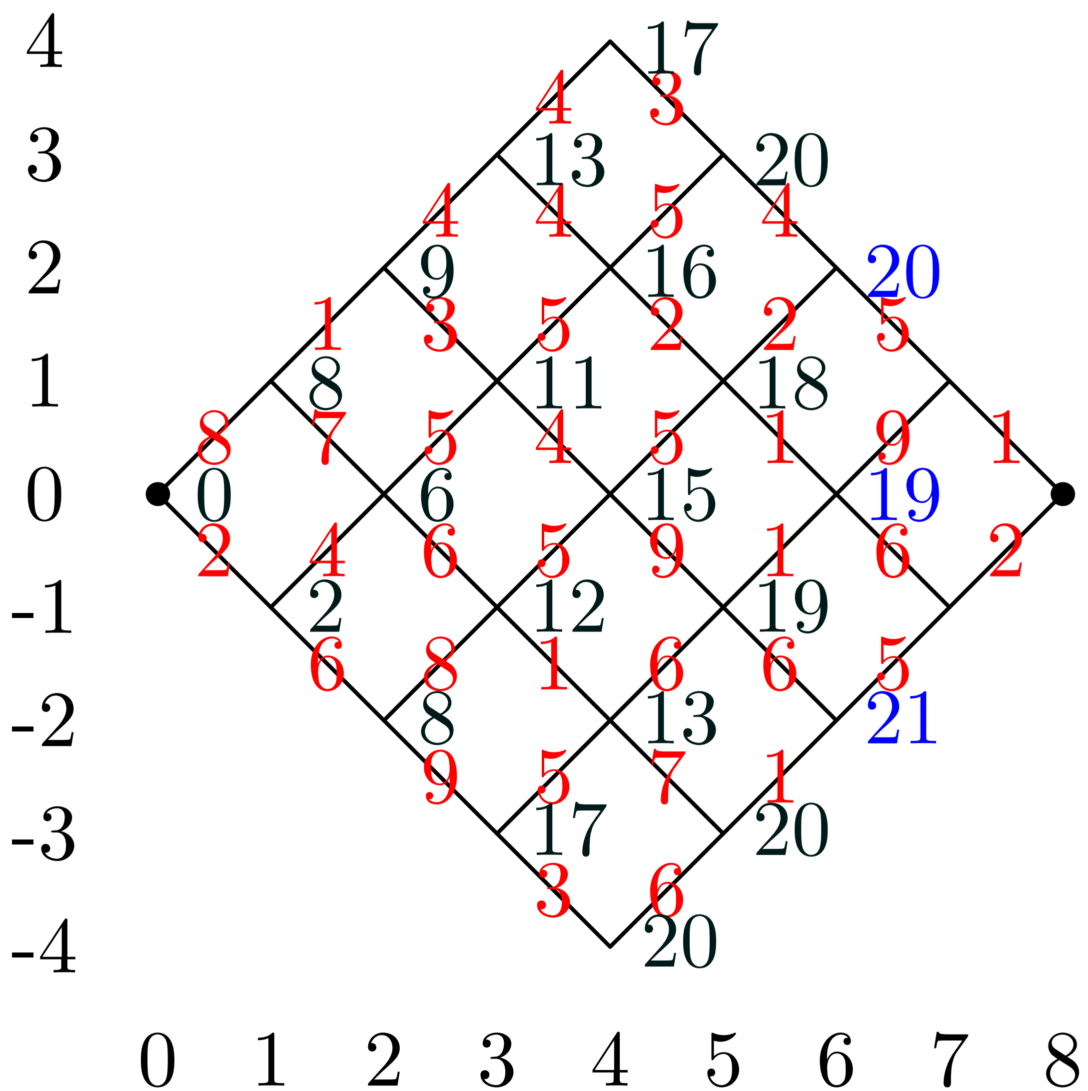


Example



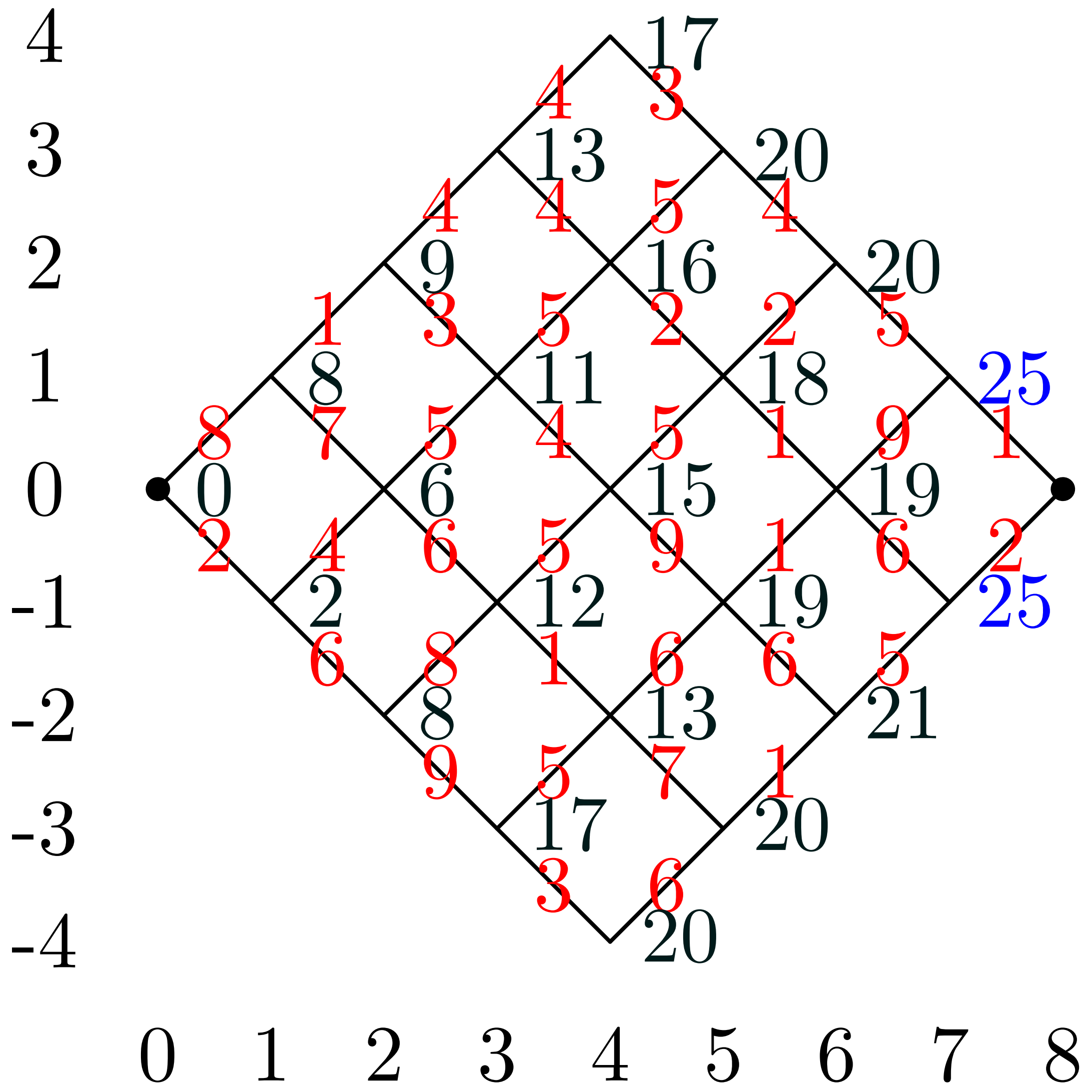
4
3
2
1
0
-1
-2
-3
-4
0 1 2 3 4 5 6 7 8
17
13
20
9
16
8
11
18
0
6
15
2
12
19
8
13
17
20
20
4 3
4 4 5 4
1 3 5 2 2 5
8 7 5 4 5 1 9 1
2 4 6 5 9 1 6 2
6 8 1 6 6 5
9 5 7 1
3 6

Example



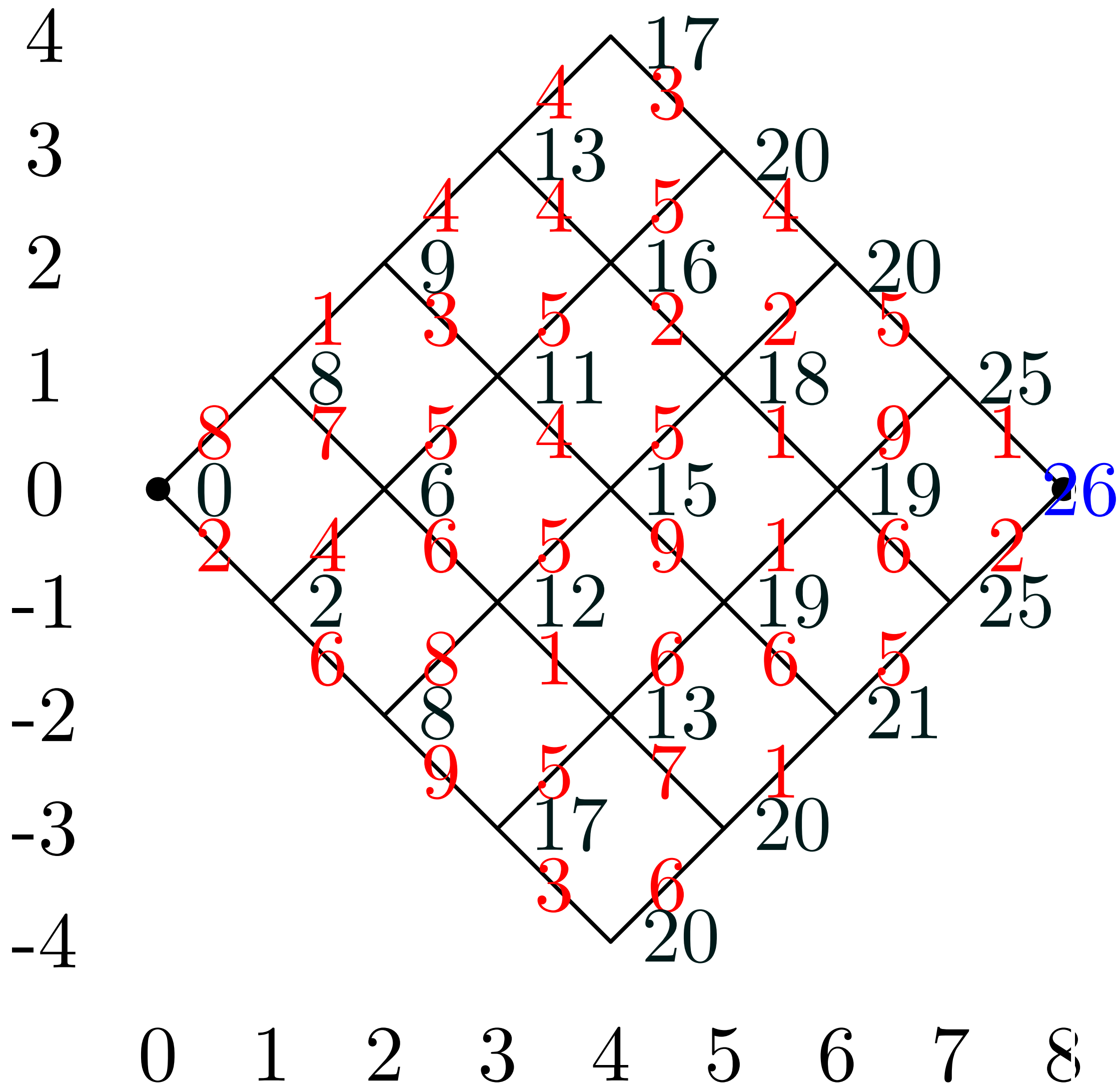
4
3
2
1
0
-1
-2
-3
-4
0 1 2 3 4 5 6 7 8
17
13
20
9
16
20
8
11
18
0
6
15
19
2
12
19
8
13
21
17
20
20

Example



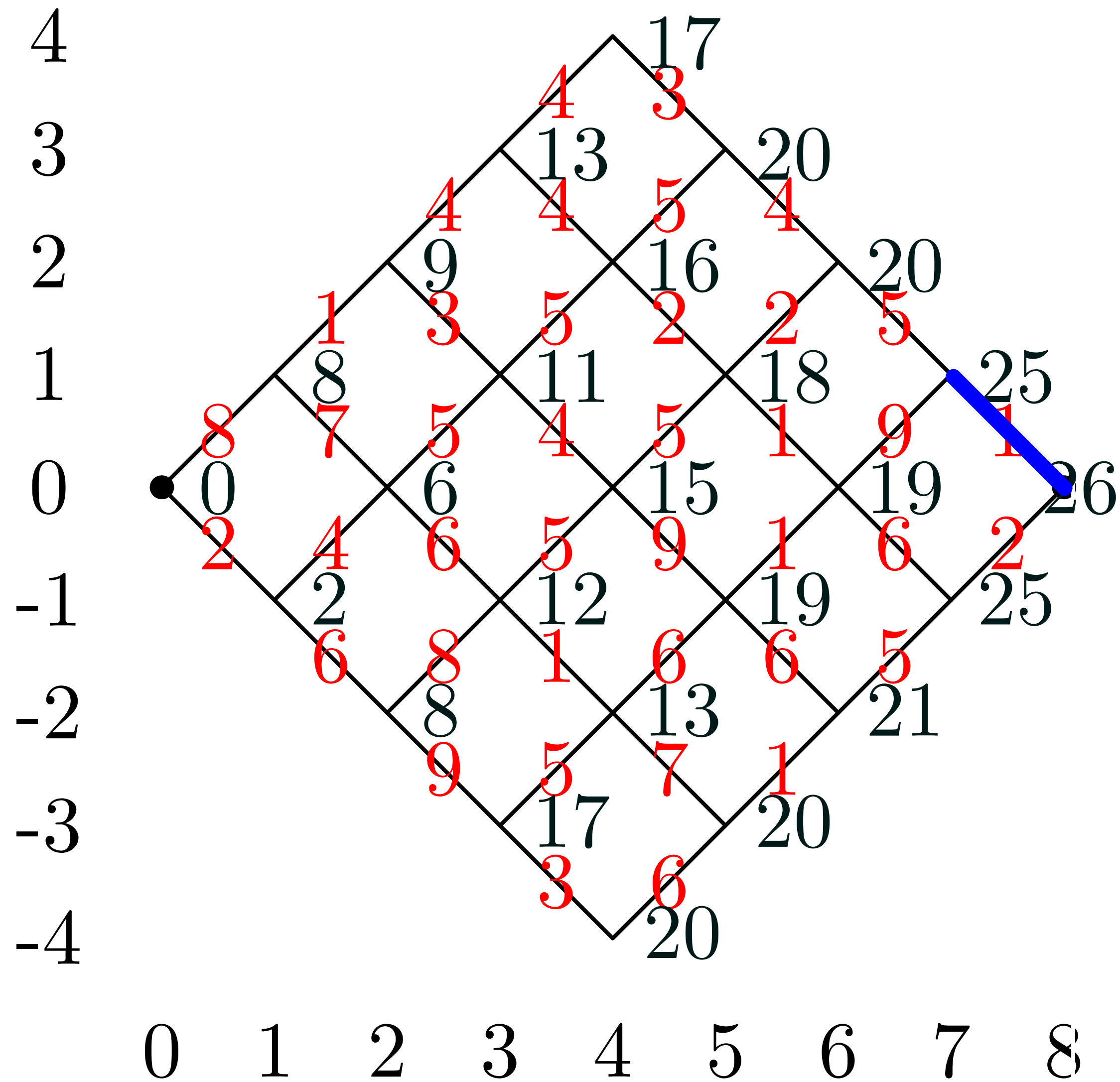
4
3
2
1
0
-1
-2
-3
-4
0 1 2 3 4 5 6 7 8
17
13
20
9
16
20
8
11
18
25
0
6
15
19
2
12
19
25
8
13
21
17
20
20

Example



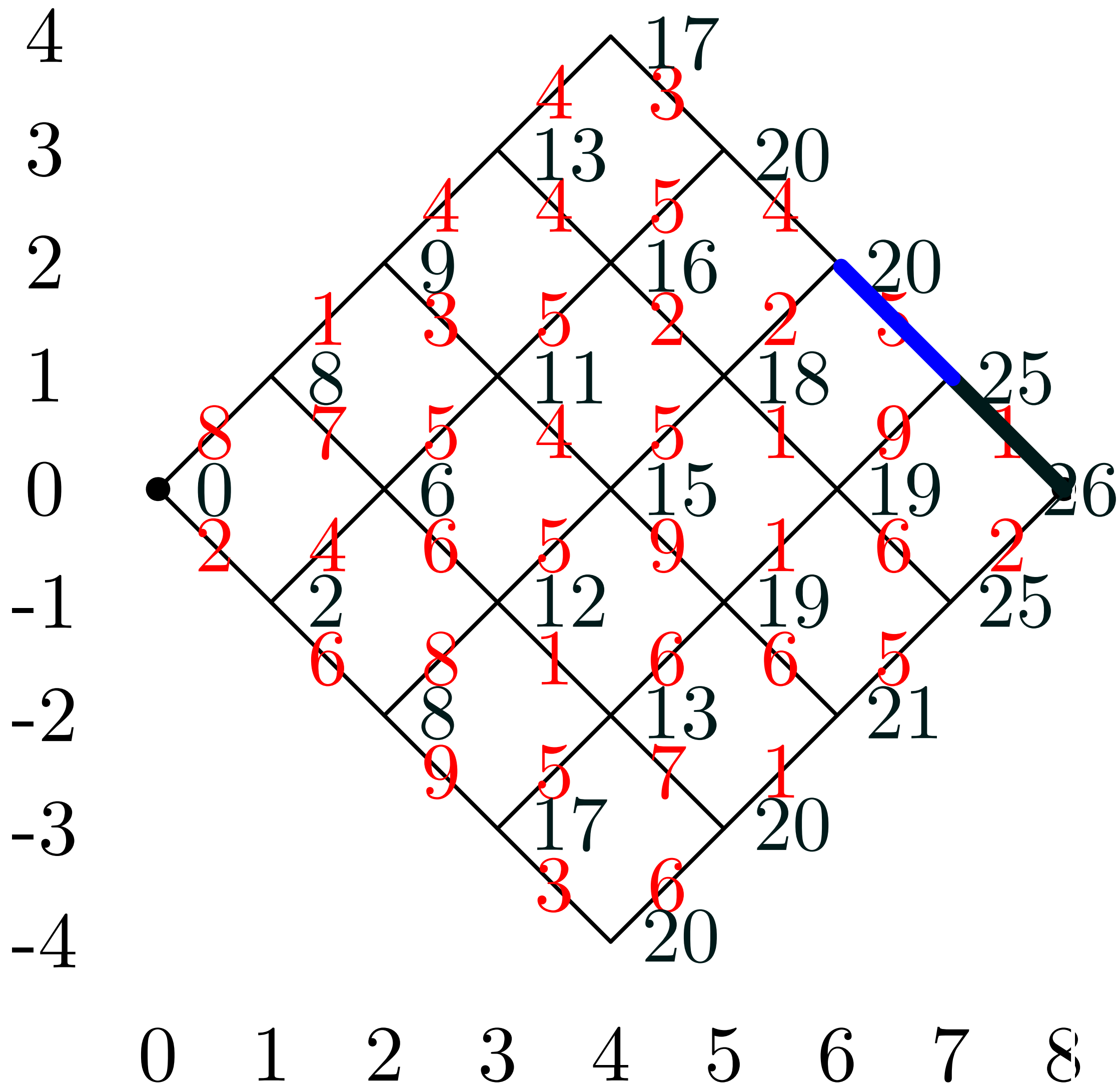
4
3
2
1
0
-1
-2
-3
-4
0 1 2 3 4 5 6 7 8
17
13
20
9
16
20
8
11
18
25
0
6
15
19
26
2
12
19
25
8
13
21
17
20
20
4 3
4 4 5 4
1 3 5 2 2 5
8 7 5 4 5 1 9 1
2 4 6 5 9 1 6 2
6 8 1 6 6 5
9 5 7 1
3 6

Example



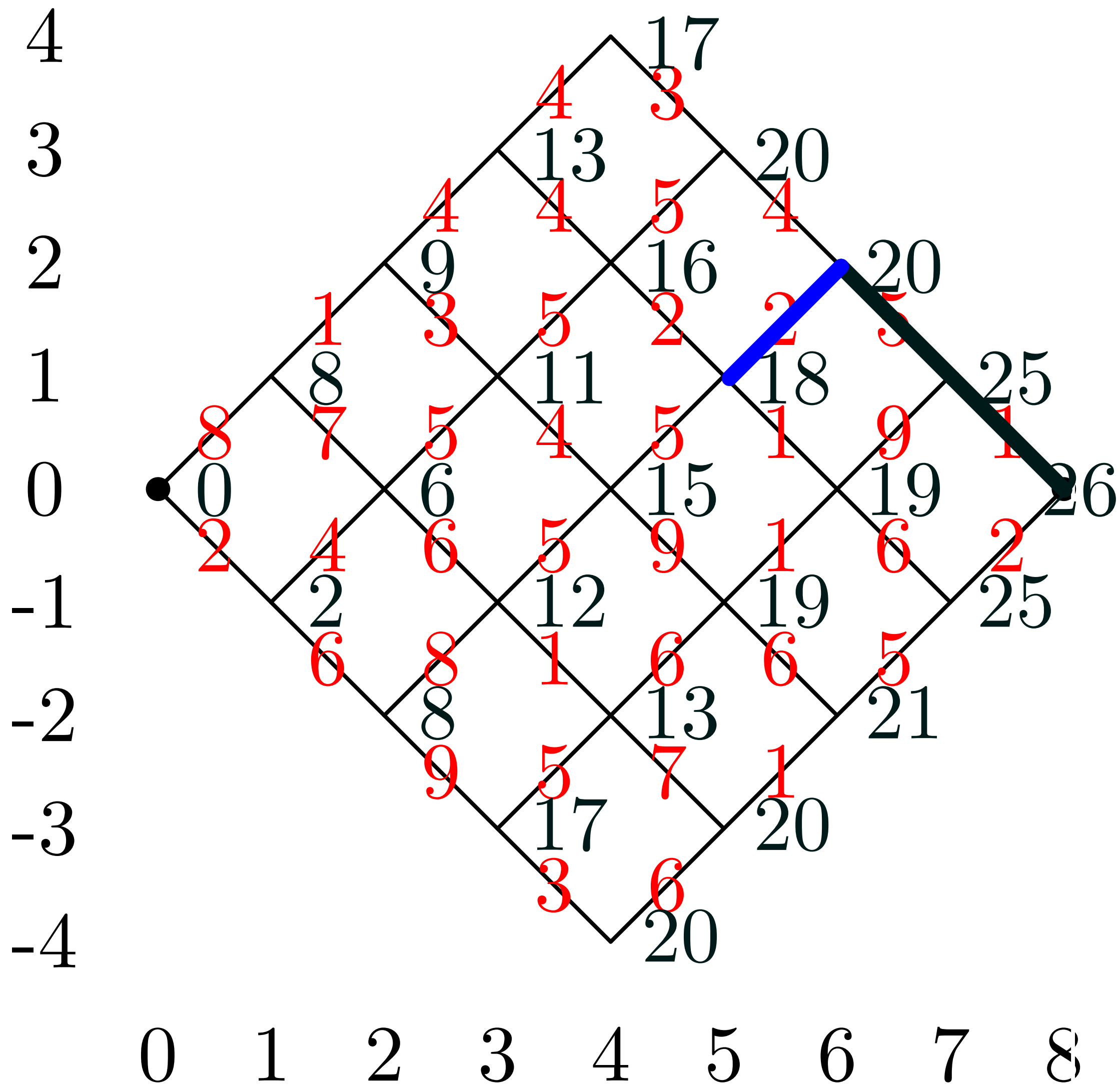
4
3
2
1
0
-1
-2
-3
-4
0 1 2 3 4 5 6 7 8
0
8
2
9
6
13
11
12
8
17
15
16
20
17
18
19
13
20
20
19
21
25
25
26

Example



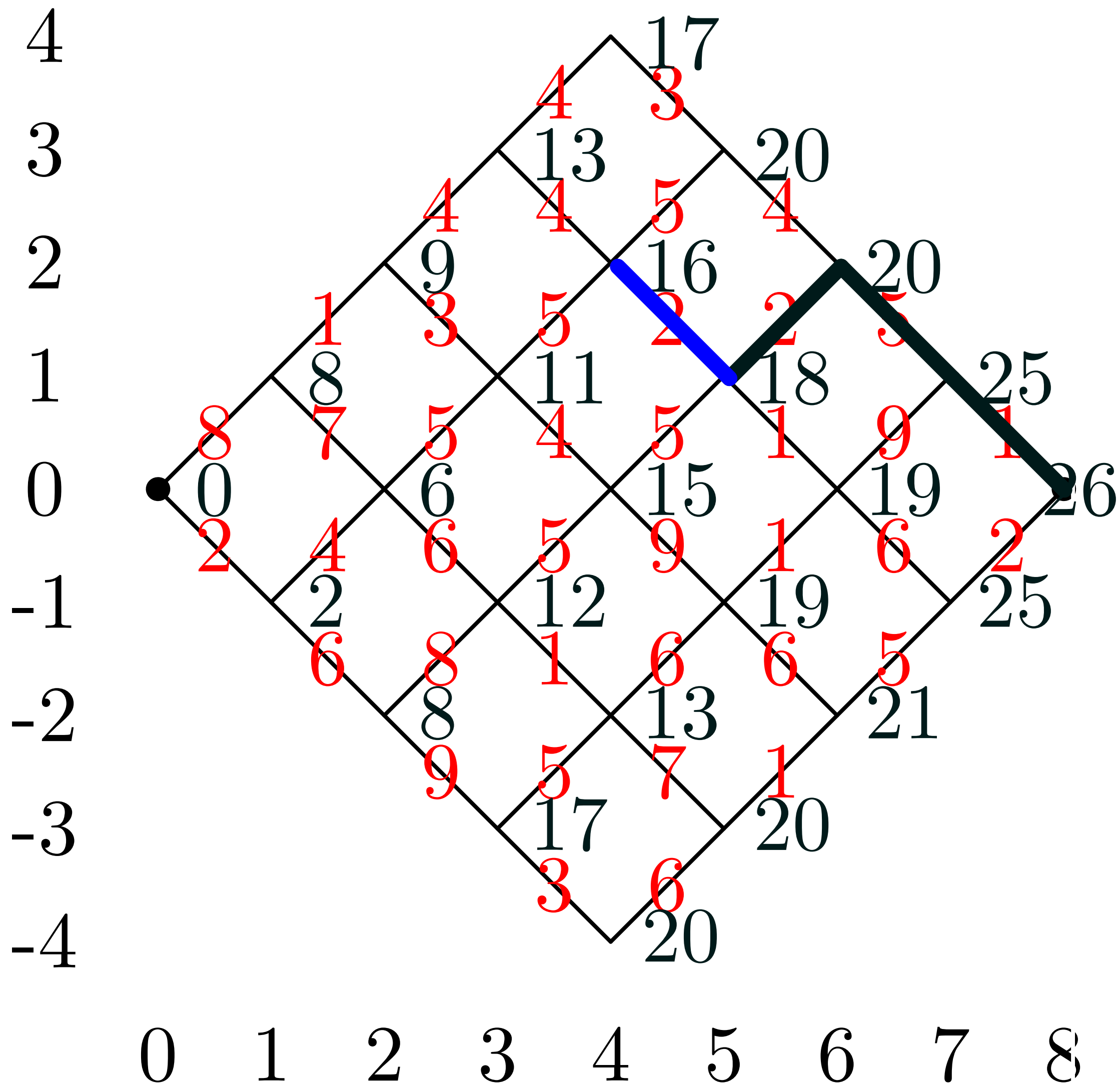
4
3
2
1
0
-1
-2
-3
-4
0 1 2 3 4 5 6 7 8
17
13
20
9
16
20
8
11
18
25
0
6
15
19
26
2
12
19
25
8
13
21
17
20
20

Example



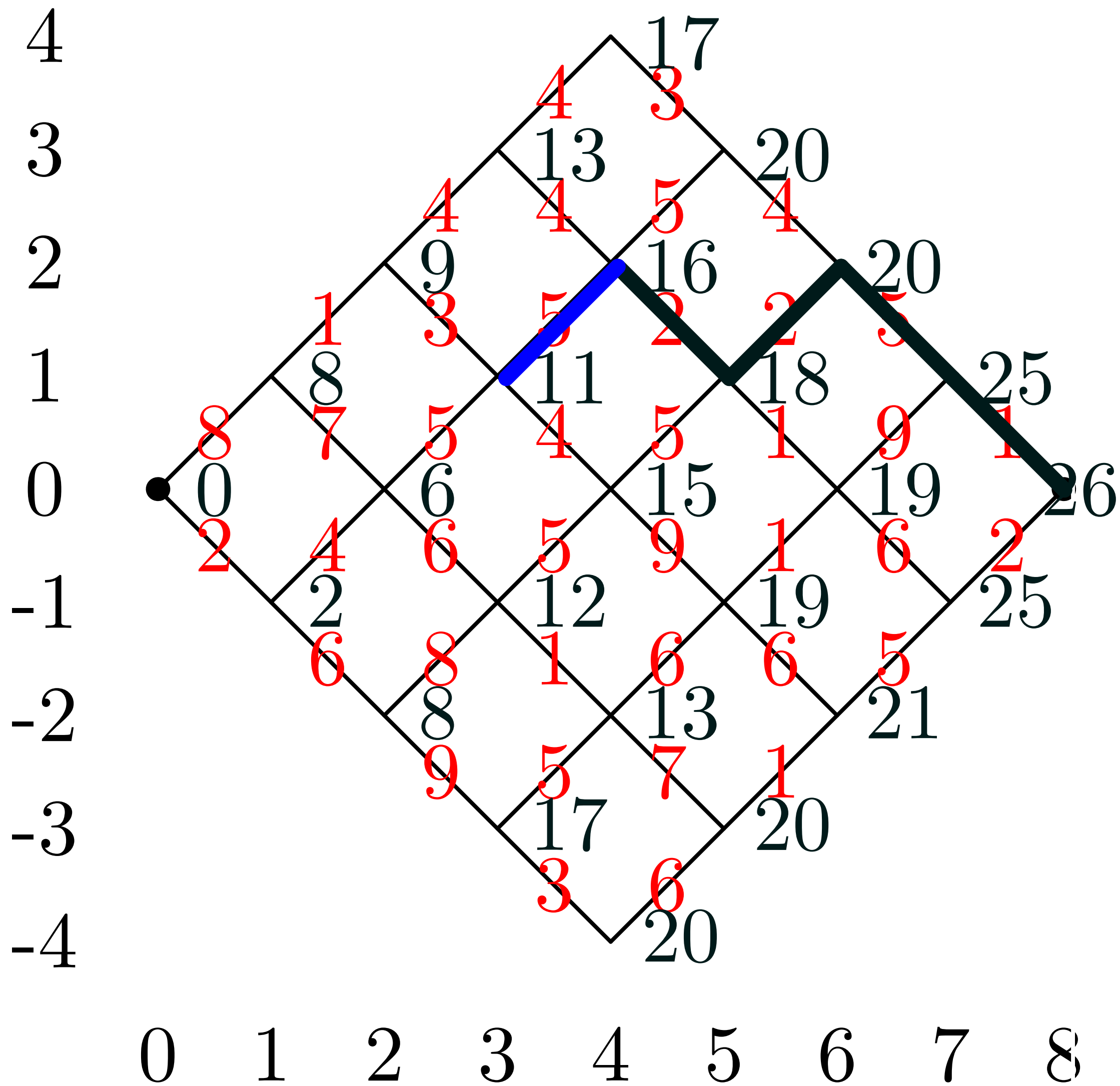
4
3
2
1
0
-1
-2
-3
-4
0 1 2 3 4 5 6 7 8
17
13
20
9
16
20
8
11
18
25
0
6
15
19
26
2
12
19
25
8
13
21
17
20
20

Example



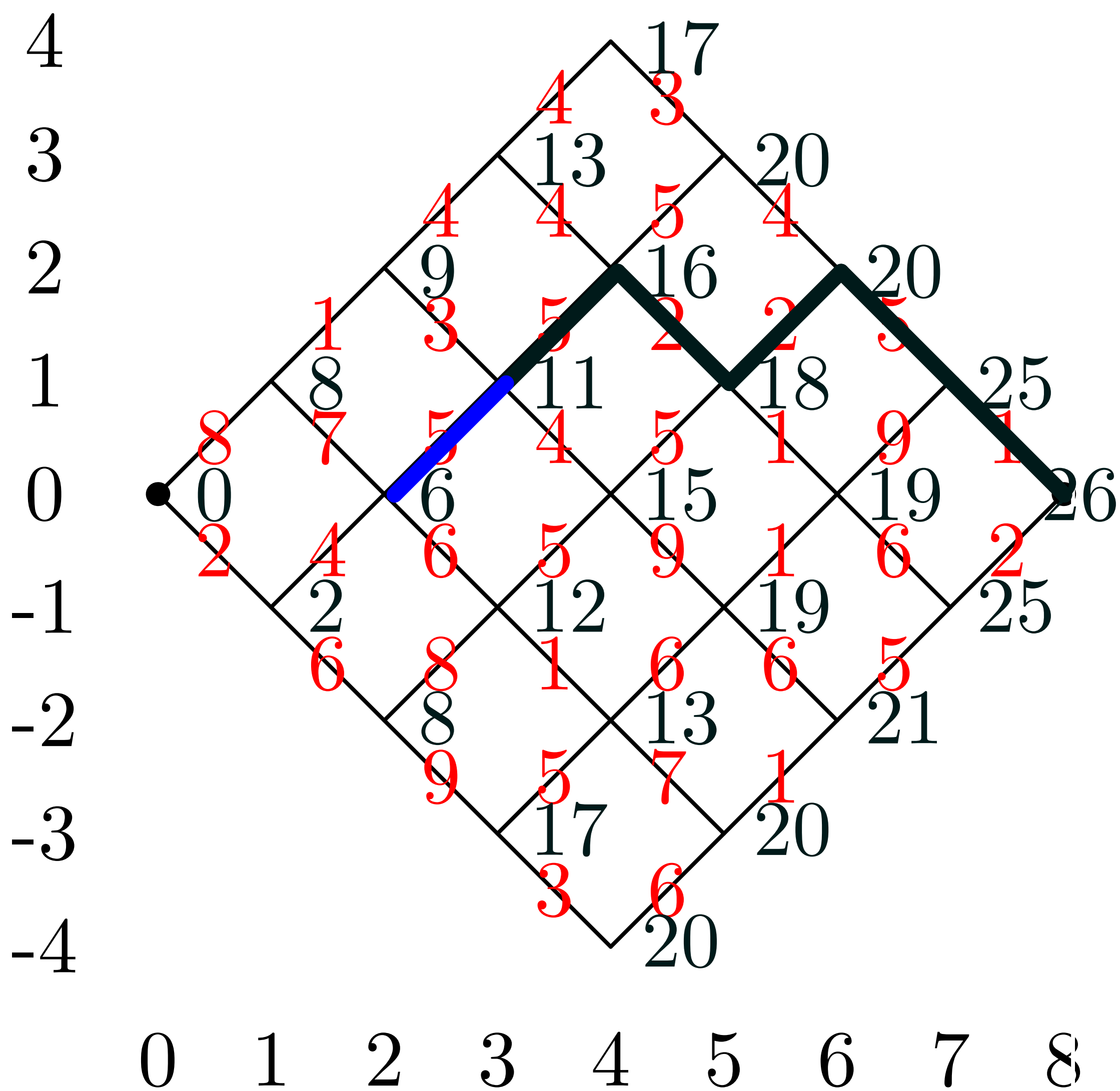
4
3
2
1
0
-1
-2
-3
-4
0 1 2 3 4 5 6 7 8
17
13
20
9
16
20
8
11
18
25
0
6
15
19
26
2
12
19
25
8
13
21
17
20
20

Example

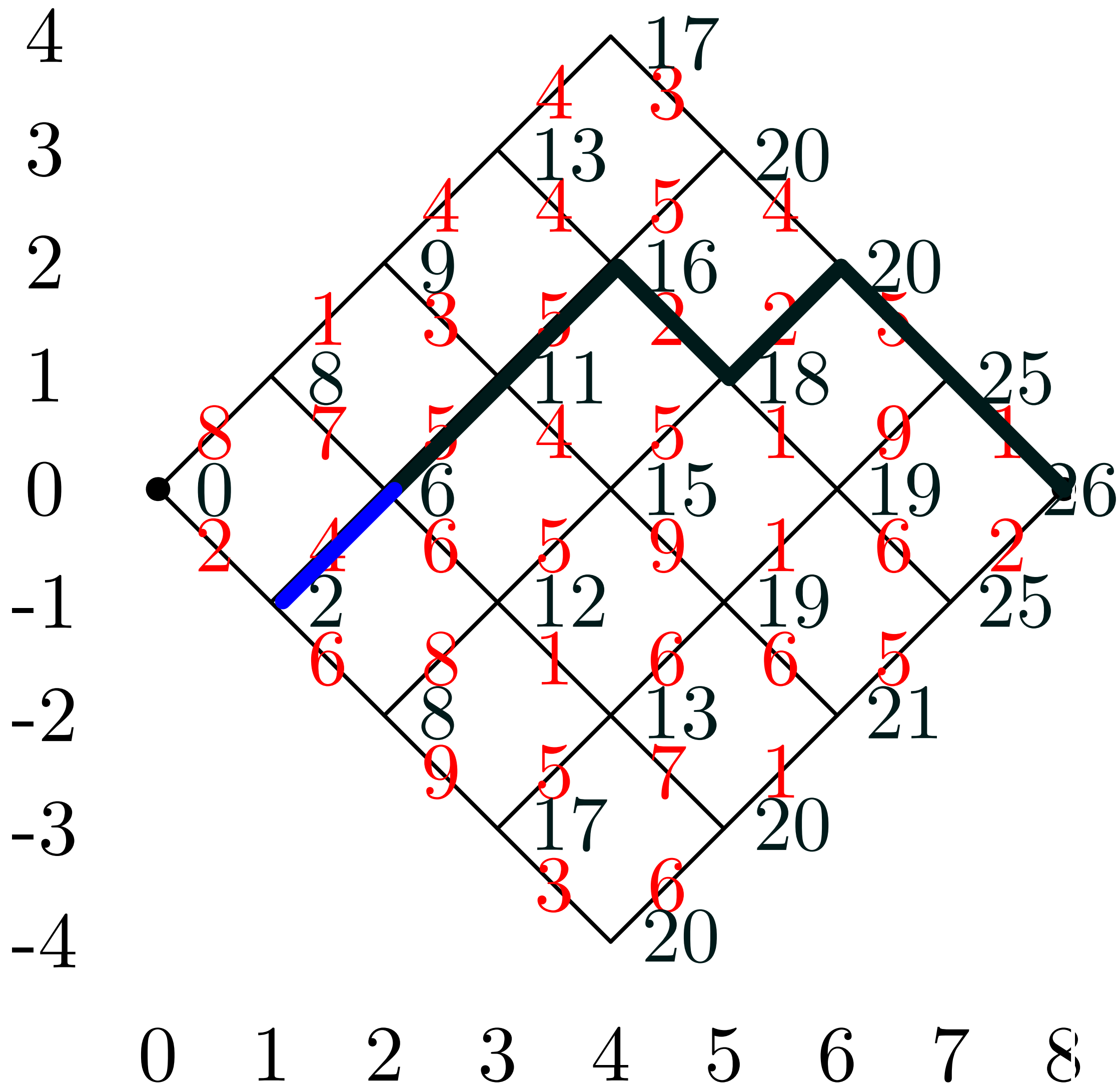


4
3
2
1
0
-1
-2
-3
-4
0 1 2 3 4 5 6 7 8
17
13
20
9
16
20
8
11
18
25
0
6
15
19
26
2
12
19
25
8
13
21
17
20
20

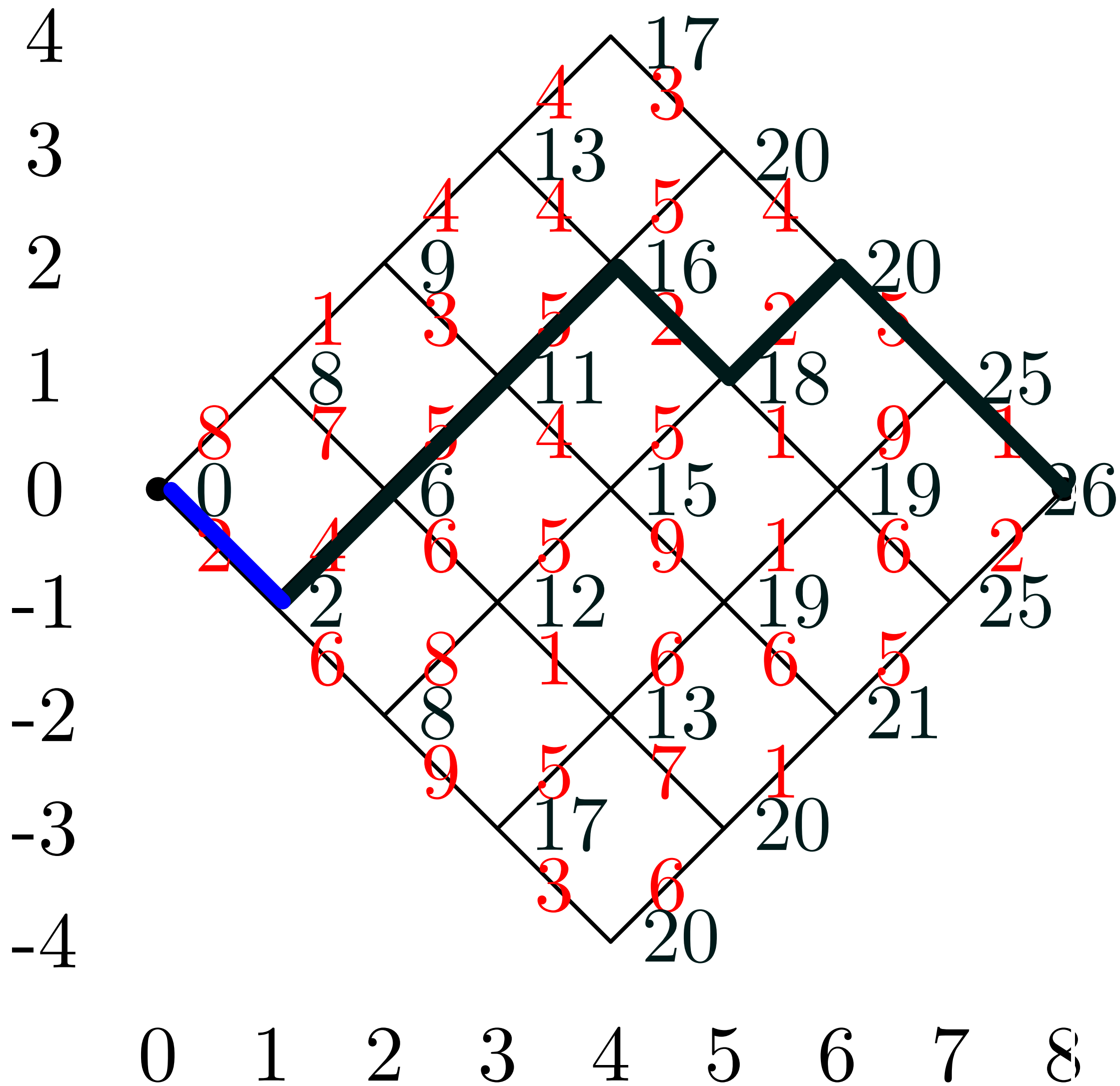
Example


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



Example



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

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

- 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

Time Complexity

- In our dynamic programming solution we had to compute the cost $c_{(i,j)}$ at each lattice point
- There were $(\frac{n+1}{2})^2$ 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 $(\frac{n+1}{2})^2$ 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 $(\frac{n+1}{2})^2$ 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

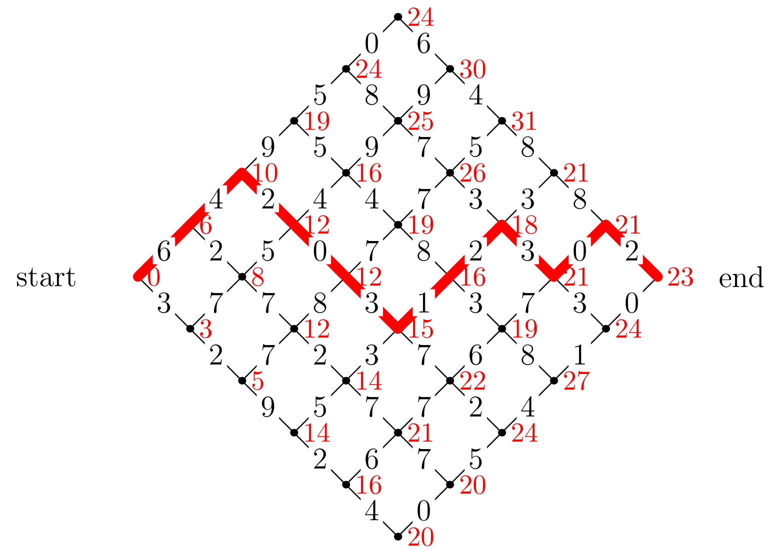
- In our dynamic programming solution we had to compute the cost $c_{(i,j)}$ at each lattice point
- There were $(\frac{n+1}{2})^2$ lattice point
- It took constant time to compute each cost so the total time to perform the forward algorithm was $\Theta(n^2)$
- The time complexity of the backward algorithm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

Outline

1. Dynamic Programming
2. **Applications**
   - Line Breaks
   - Edit Distance
   - Dijkstra's Algorithm
3. Limitation



Applications of Dynamic Programming

- Dynamic programming is used in a vast number of applications
  - ⋆ String matching algorithms
  - ⋆ Shape matching in images
  - ⋆ Dynamical time-warping in speech
  - ⋆ Hidden Markov Models in machine learning
- Unlike greedy algorithms the idea is readily extended to many different applications

Applications of Dynamic Programming

- Dynamic programming is used in a vast number of applications
  - ⋆ String matching algorithms
  - ⋆ Shape matching in images
  - ⋆ Dynamical time-warping in speech
  - ⋆ Hidden Markov Models in machine learning
- Unlike greedy algorithms the idea is readily extended to many different applications

Using Dynamic Programming

- The challenge is recognising that you can use dynamic programming and representing the problem right

- Learn this from examples

- Consider writing a word processor that splits paragraphs up into lines

- You want to choose the line breaks so that the lines are all roughly the same length

- This is a global optimisation task

  minimise the total number of spaces left at the end of each line

Using Dynamic Programming

- The challenge is recognising that you can use dynamic programming and representing the problem right
- Learn this from examples
- Consider writing a word processor that splits paragraphs up into lines
- You want to choose the line breaks so that the lines are all roughly the same length
- This is a global optimisation task

  minimise the total number of spaces left at the end of each line

Using Dynamic Programming

- The challenge is recognising that you can use dynamic programming and representing the problem right
- Learn this from examples
- Consider writing a word processor that splits paragraphs up into lines
- You want to choose the line breaks so that the lines are all roughly the same length
- This is a global optimisation task

  minimise the total number of spaces left at the end of each line

Using Dynamic Programming

- The challenge is recognising that you can use dynamic programming and representing the problem right
- Learn this from examples
- Consider writing a word processor that splits paragraphs up into lines
- You want to choose the line breaks so that the lines are all roughly the same length
- This is a global optimisation task

  minimise the total number of spaces left at the end of each line

Using Dynamic Programming

- The challenge is recognising that you can use dynamic programming and representing the problem right
- Learn this from examples
- Consider writing a word processor that splits paragraphs up into lines
- You want to choose the line breaks so that the lines are all roughly the same length
- This is a global optimisation task

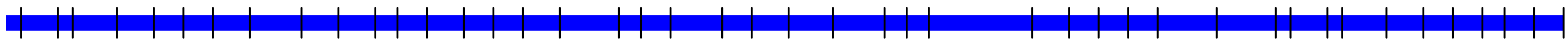
  minimise the total number of spaces left at the end of each line

Using Dynamic Programming

- I have a dream that one day this nation will rise up and live out the. . .

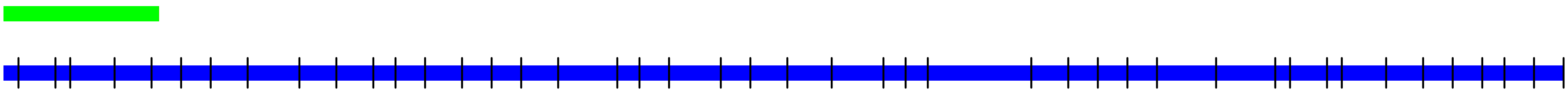
Using Dynamic Programming

- I have a dream that one day this nation will rise up and live out the. . .



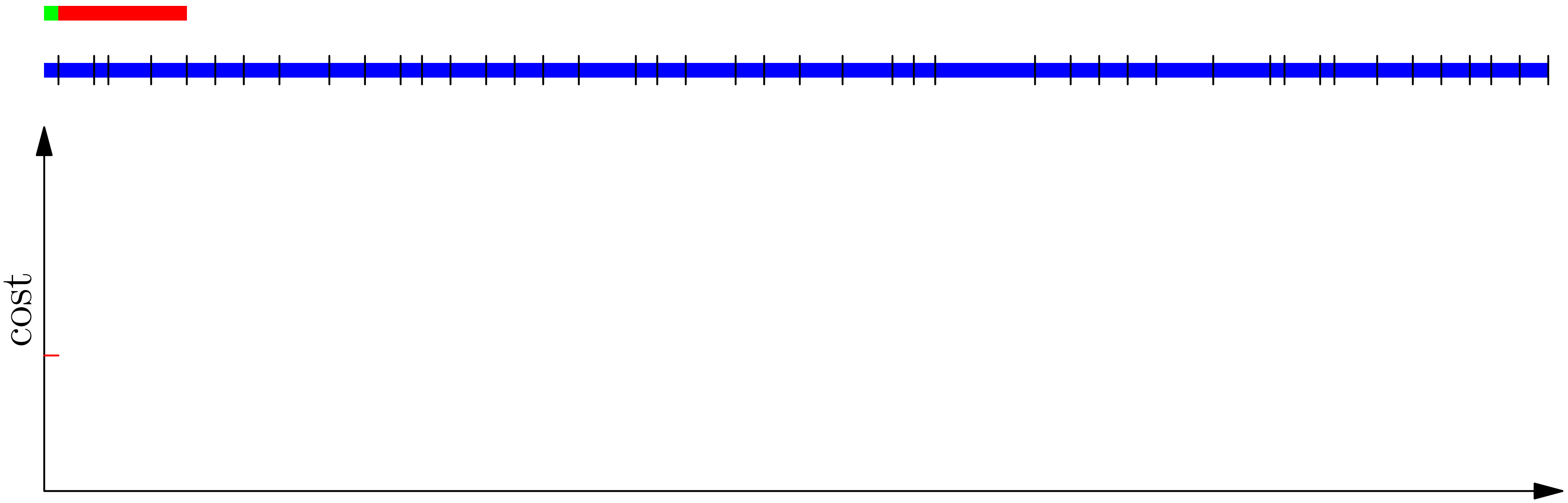
Using Dynamic Programming

- I have a dream that one day this nation will rise up and live out the. . .



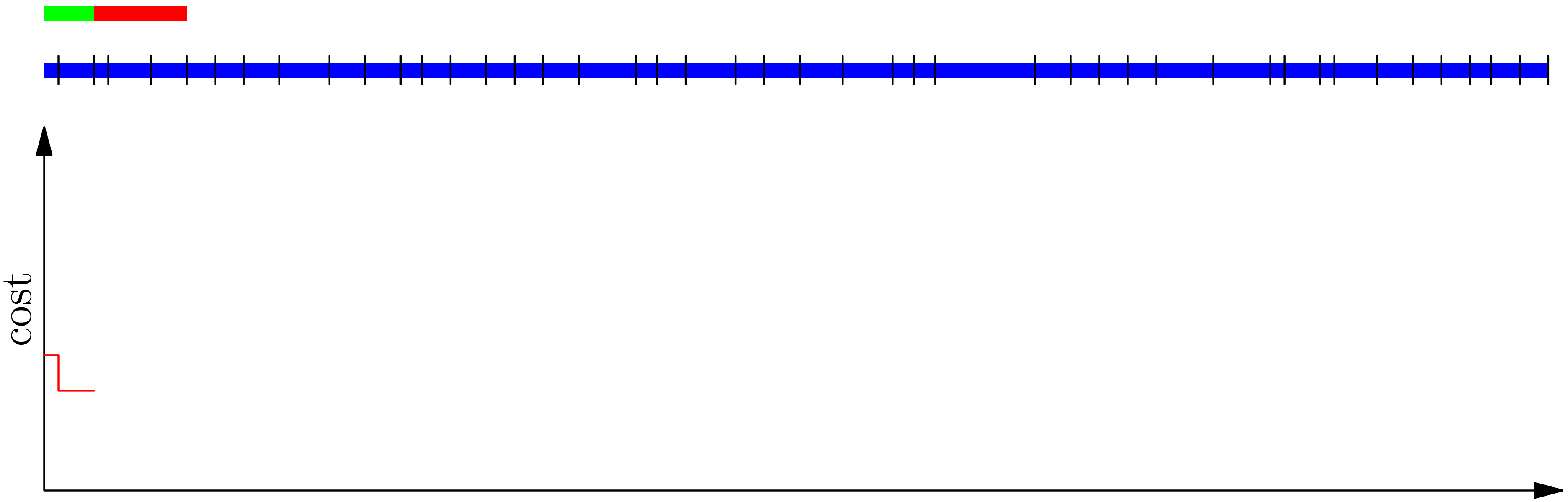
Using Dynamic Programming

- I have a dream that one day this nation will rise up and live out the. . .



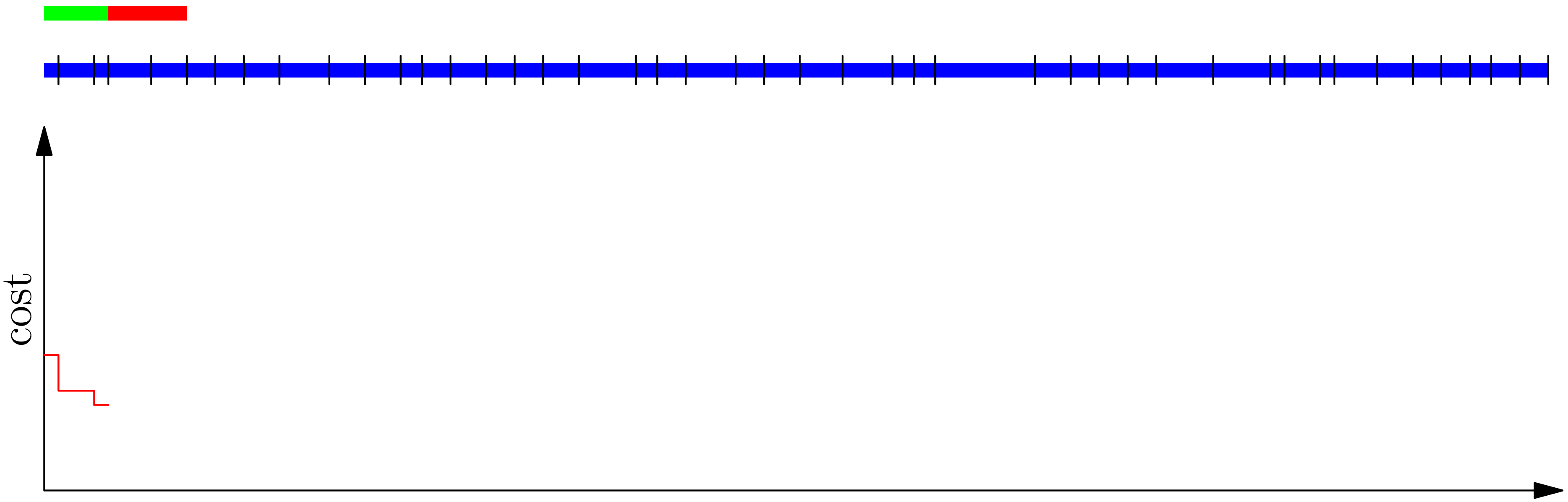
Using Dynamic Programming

- I have a dream that one day this nation will rise up and live out the. . .



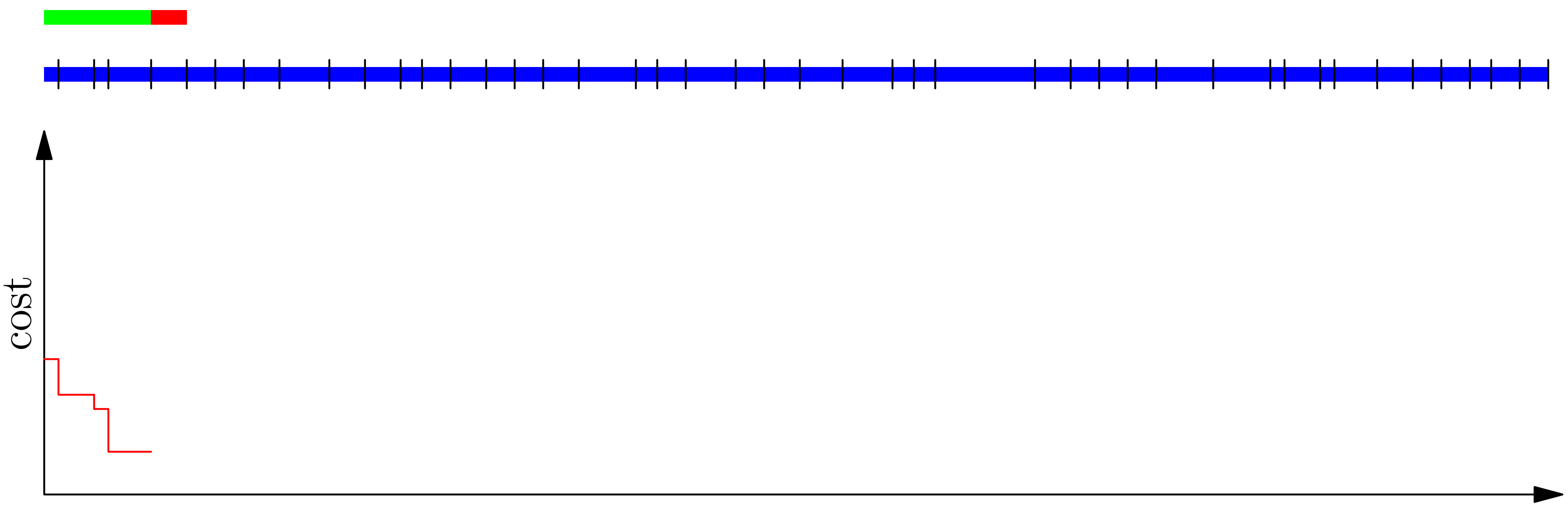
Using Dynamic Programming

- I have a dream that one day this nation will rise up and live out the. . .



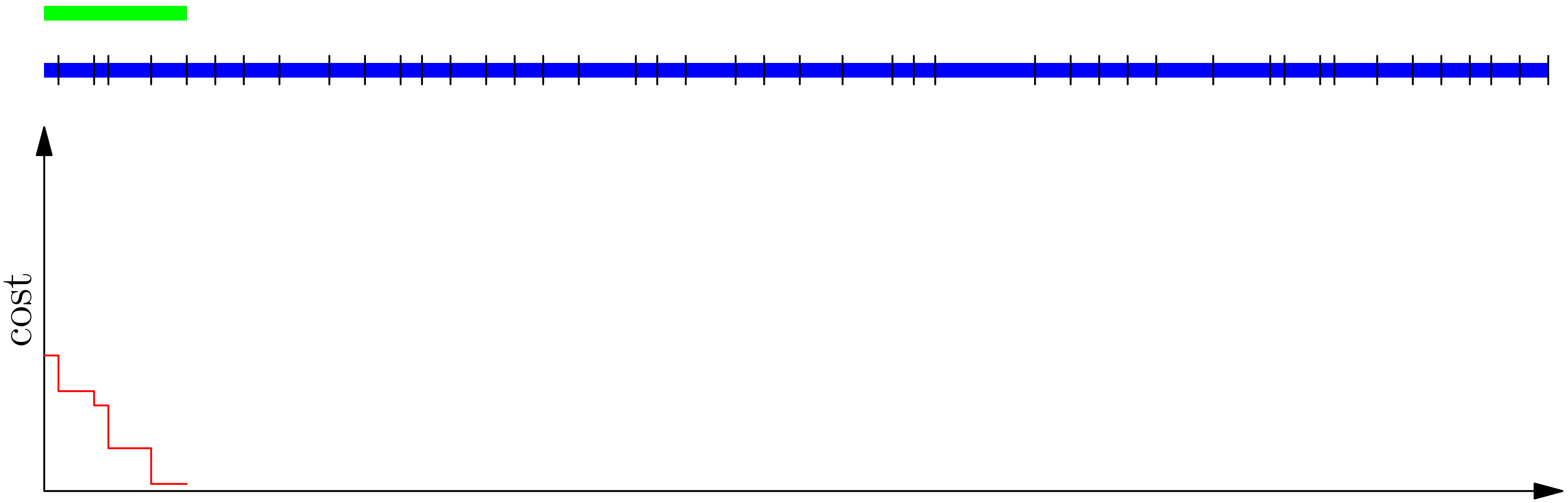
Using Dynamic Programming

- I have a dream that one day this nation will rise up and live out the. . .



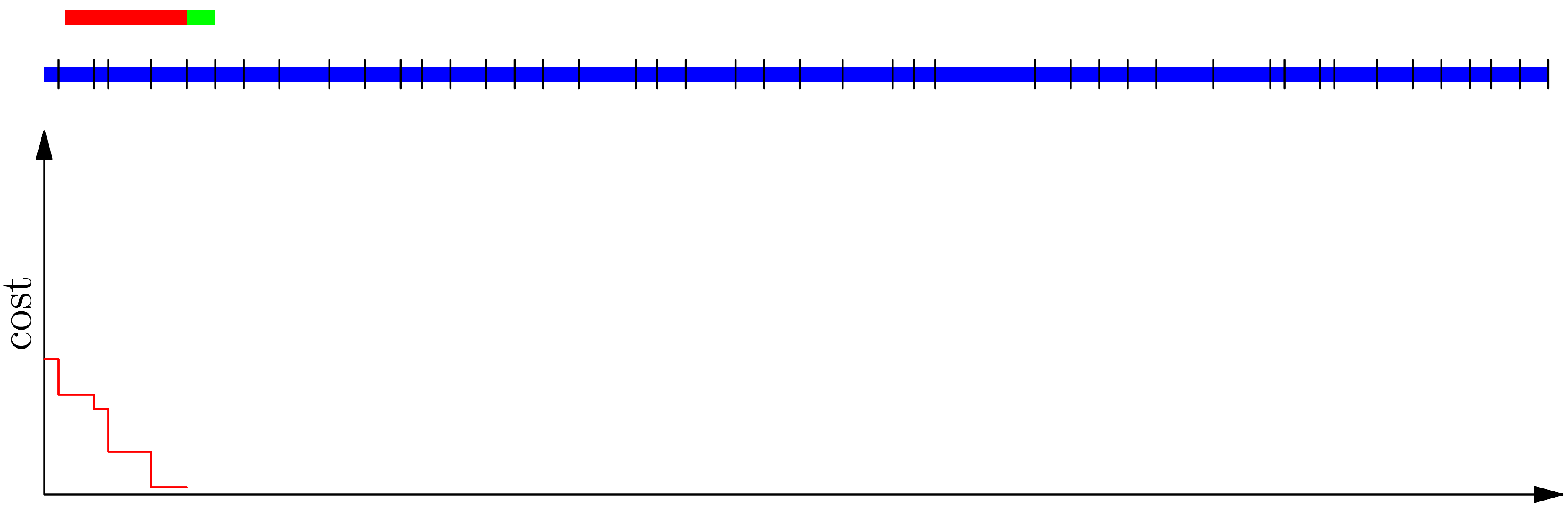
Using Dynamic Programming

- I have a dream that one day this nation will rise up and live out the. . .



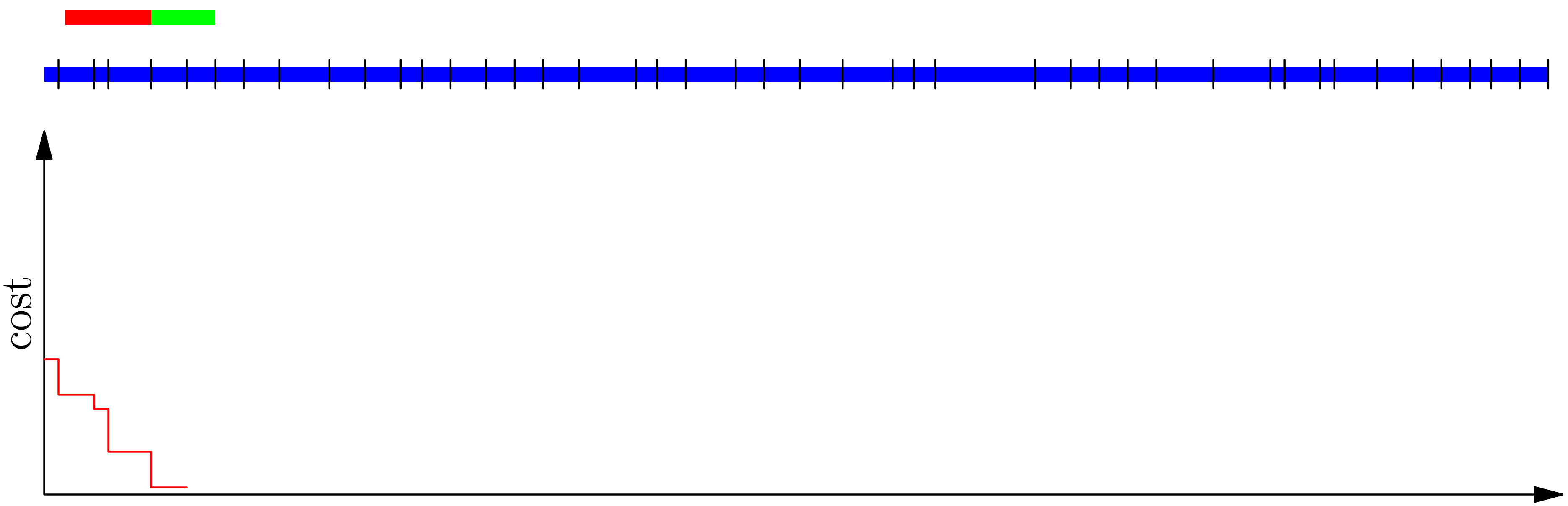
Using Dynamic Programming

- I have a dream that one day this nation will rise up and live out the. . .



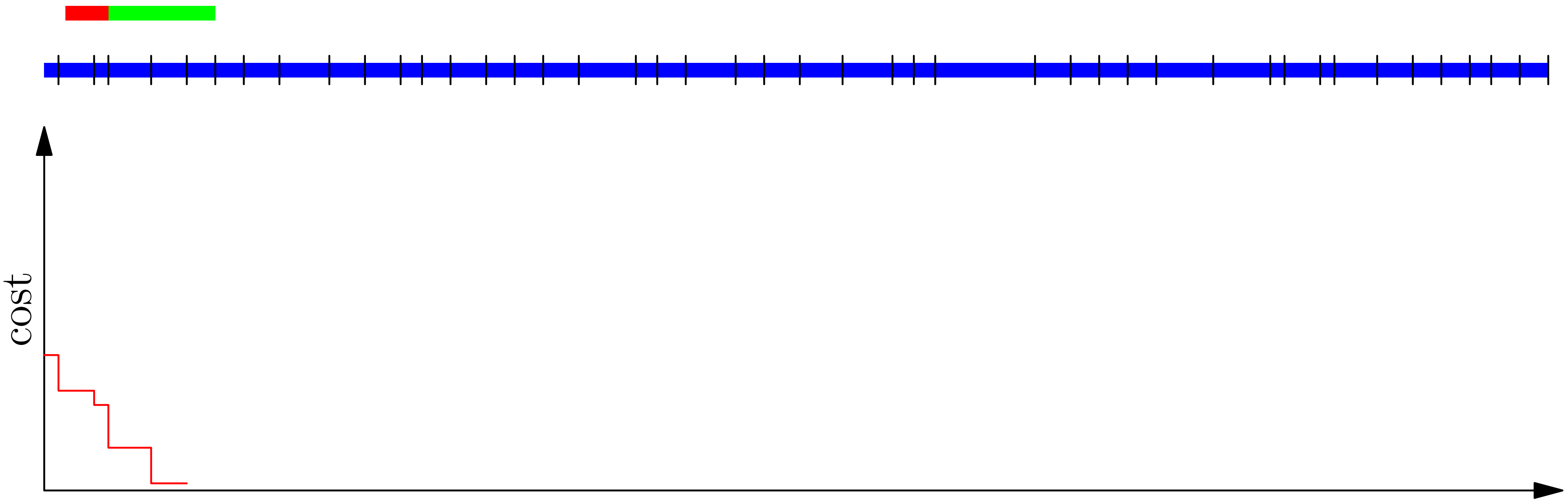
Using Dynamic Programming

- I have a dream that one day this nation will rise up and live out the. . .



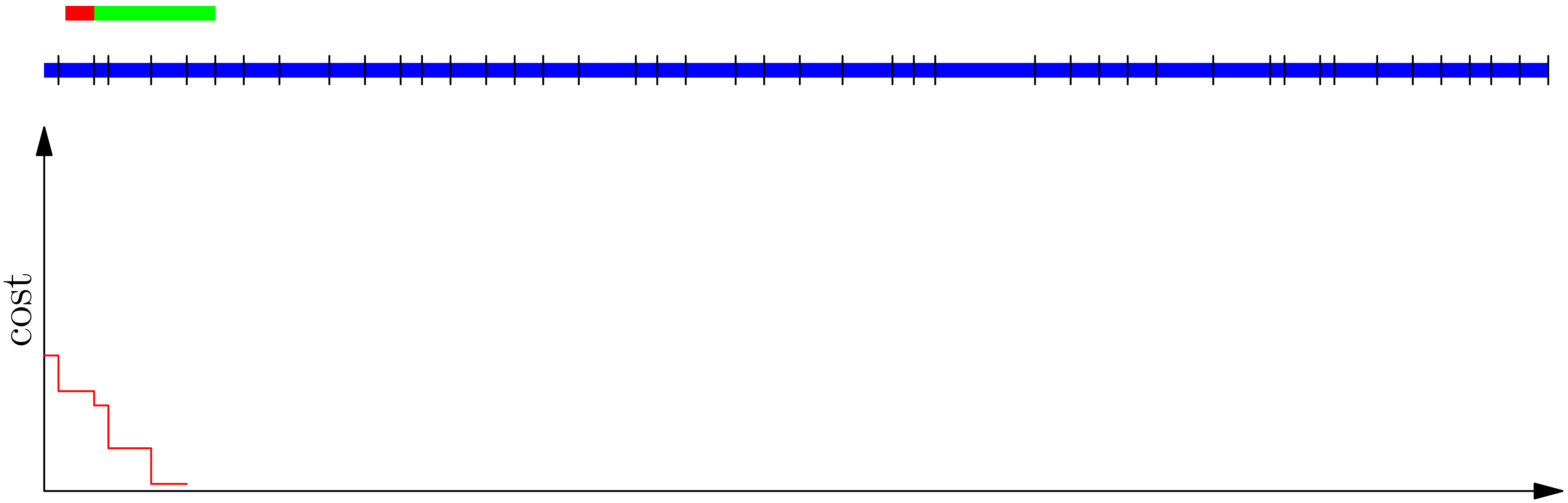
Using Dynamic Programming

- I have a dream that one day this nation will rise up and live out the. . .



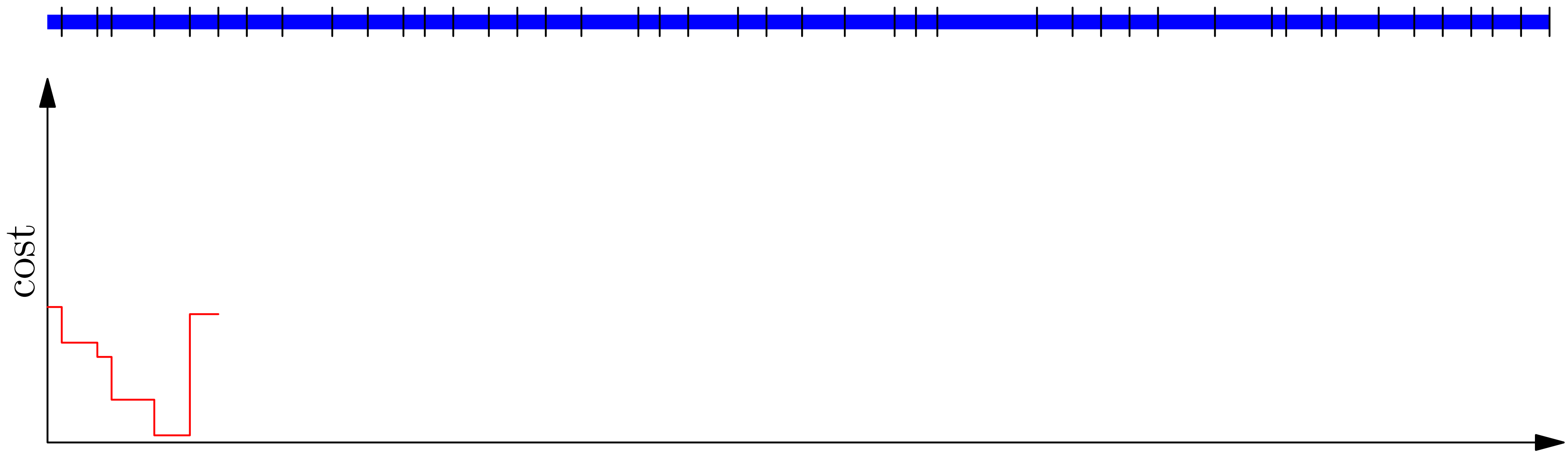
Using Dynamic Programming

- I have a dream that one day this nation will rise up and live out the. . .



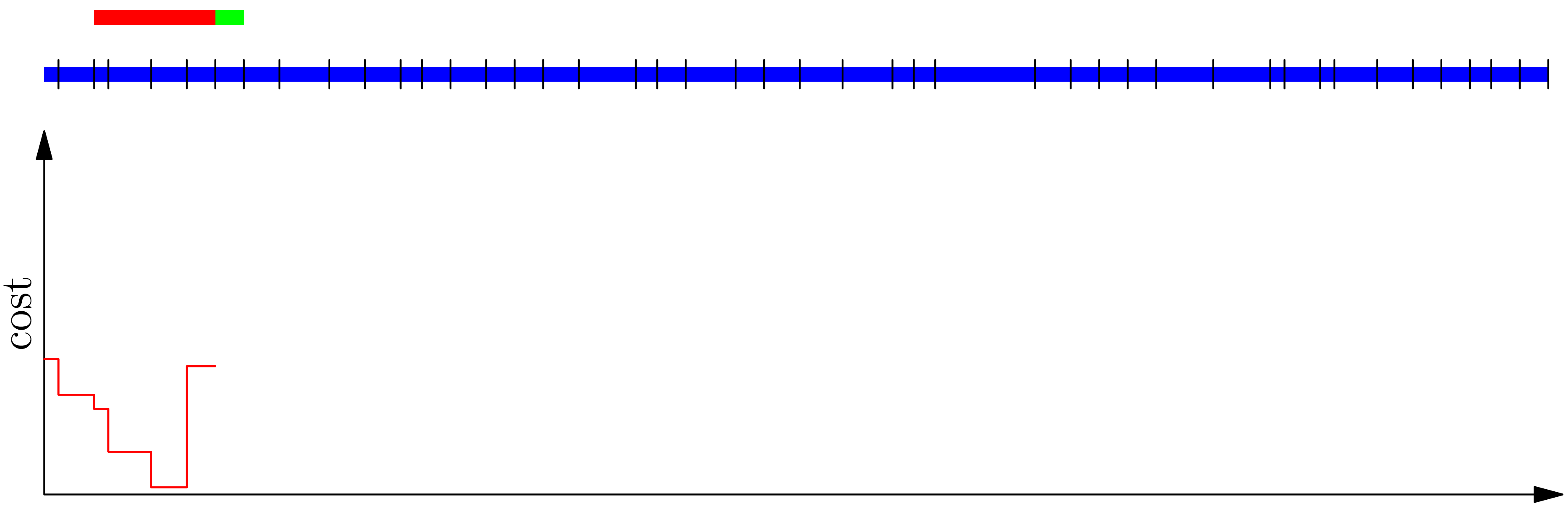
Using Dynamic Programming

- I have a dream that one day this nation will rise up and live out the. . .



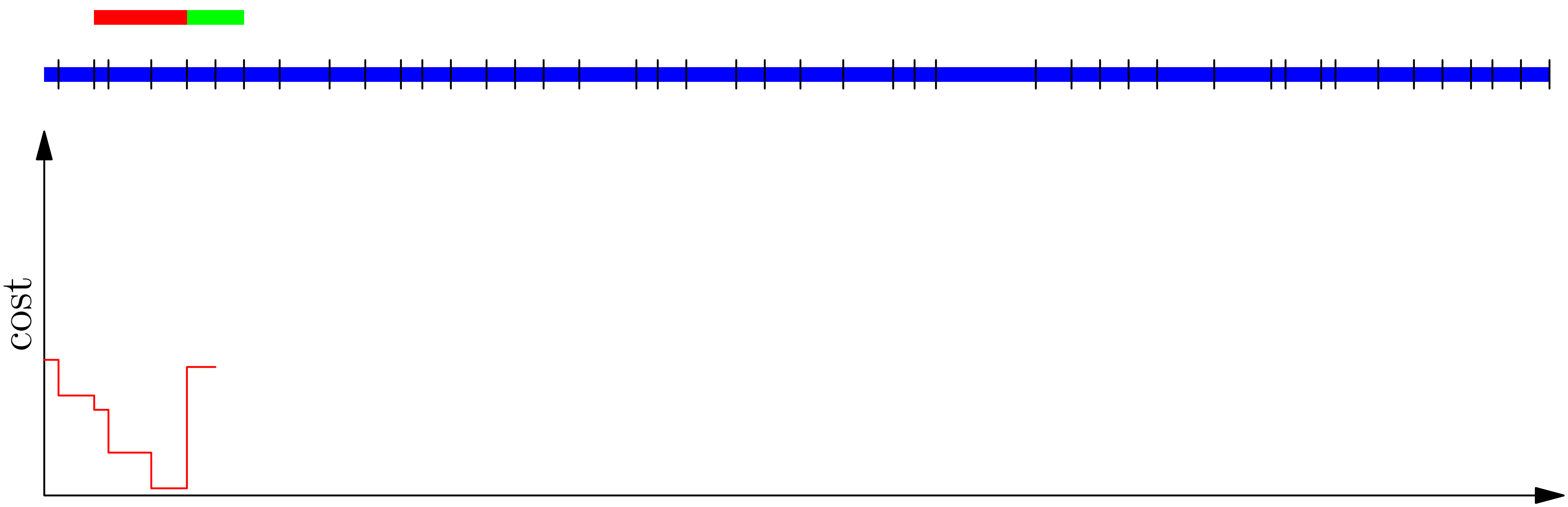
Using Dynamic Programming

- I have a dream that one day this nation will rise up and live out the. . .



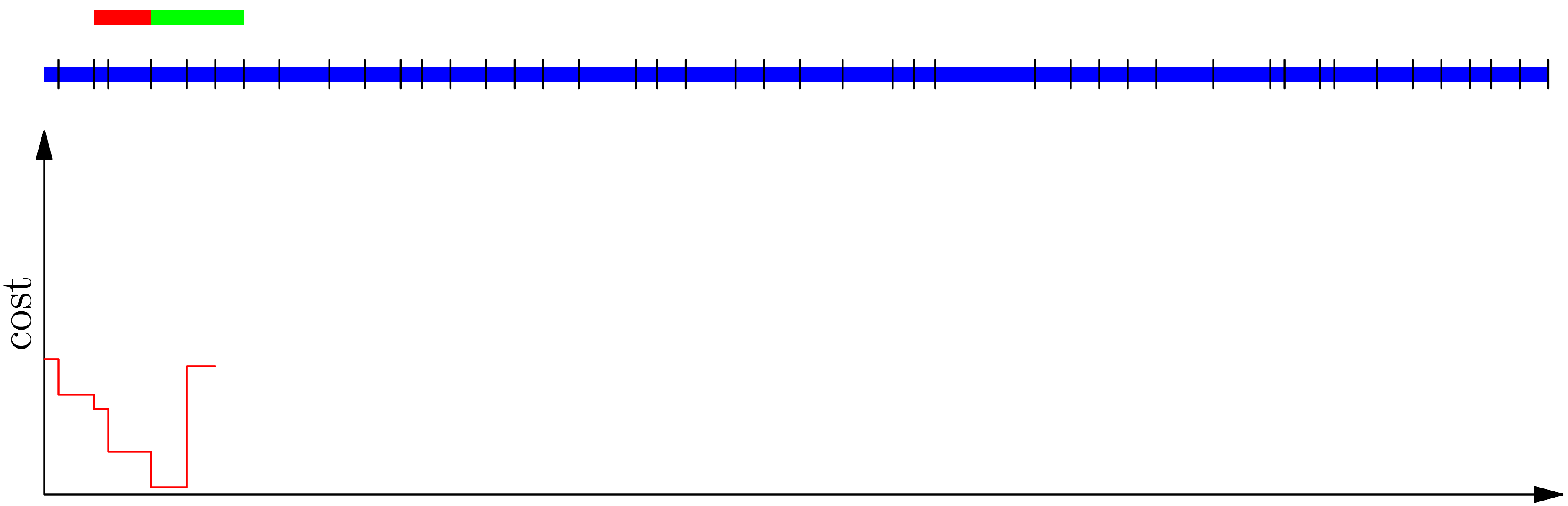
Using Dynamic Programming

- I have a dream that one day this nation will rise up and live out the. . .



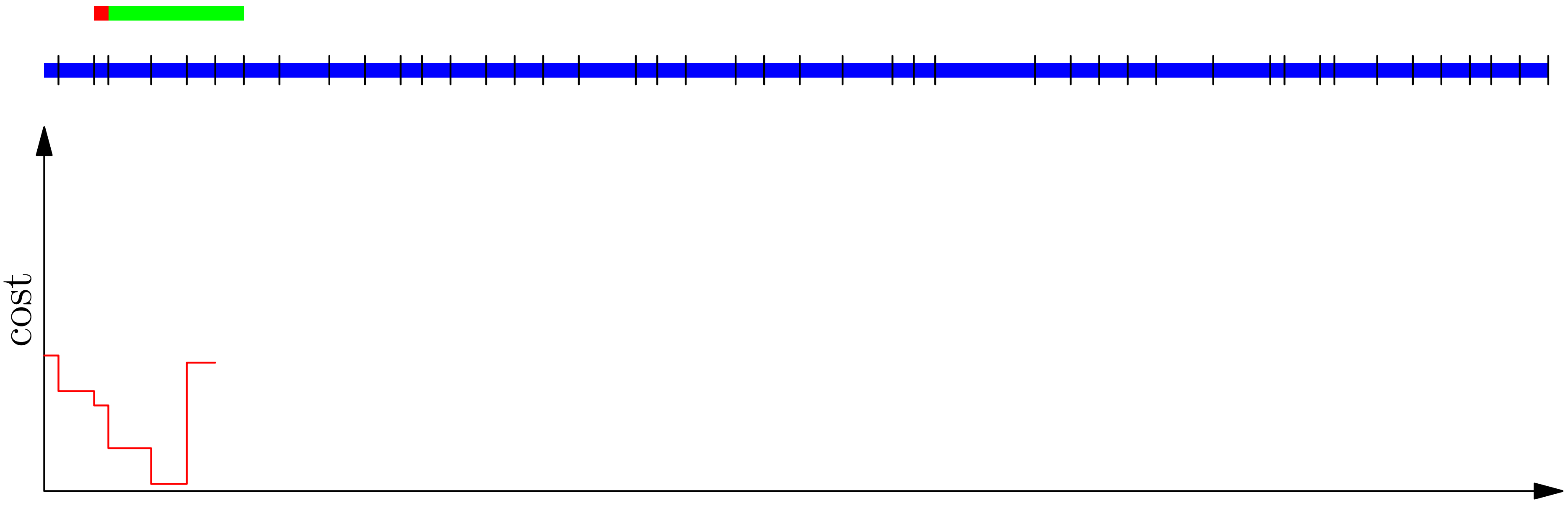
Using Dynamic Programming

- I have a dream that one day this nation will rise up and live out the. . .



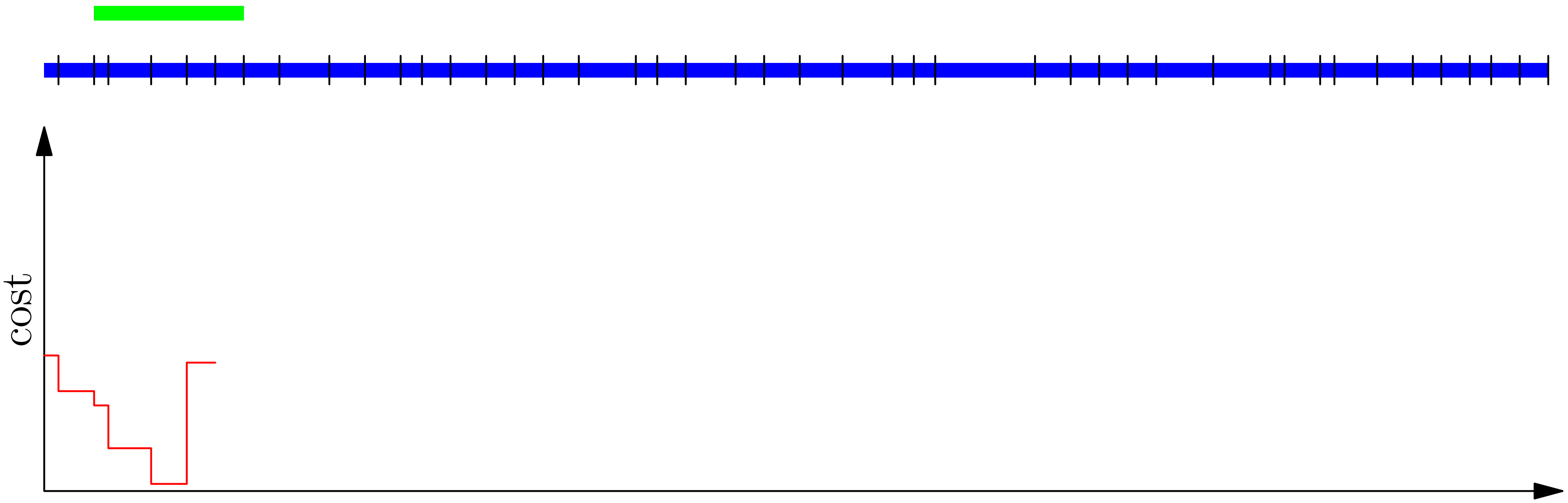
Using Dynamic Programming

- I have a dream that one day this nation will rise up and live out the. . .



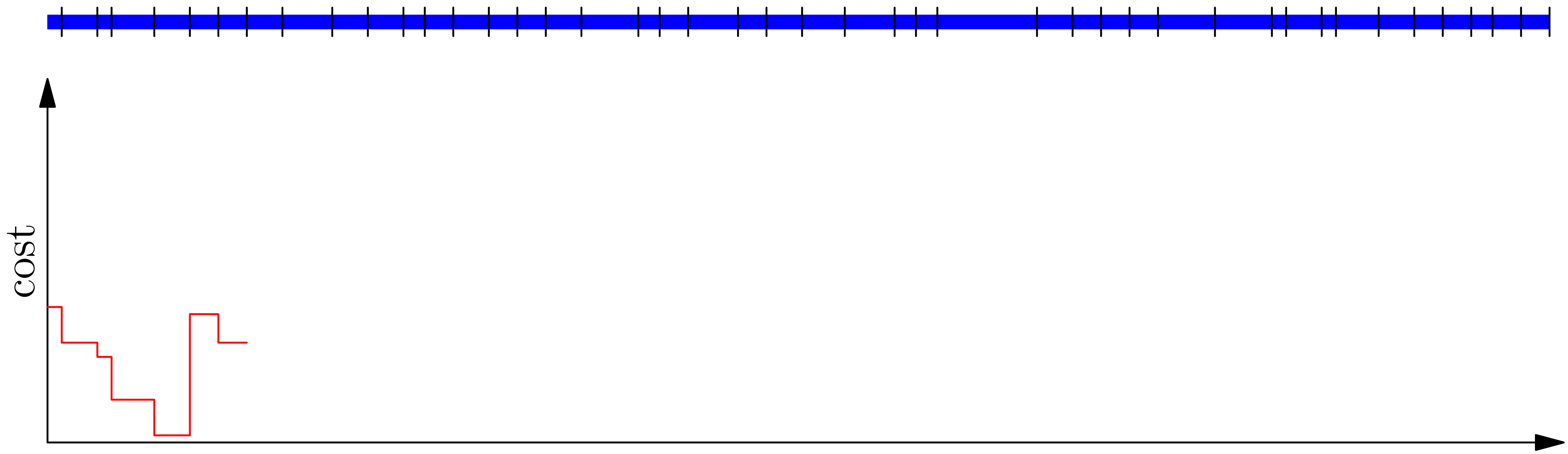
Using Dynamic Programming

- I have a dream that one day this nation will rise up and live out the. . .



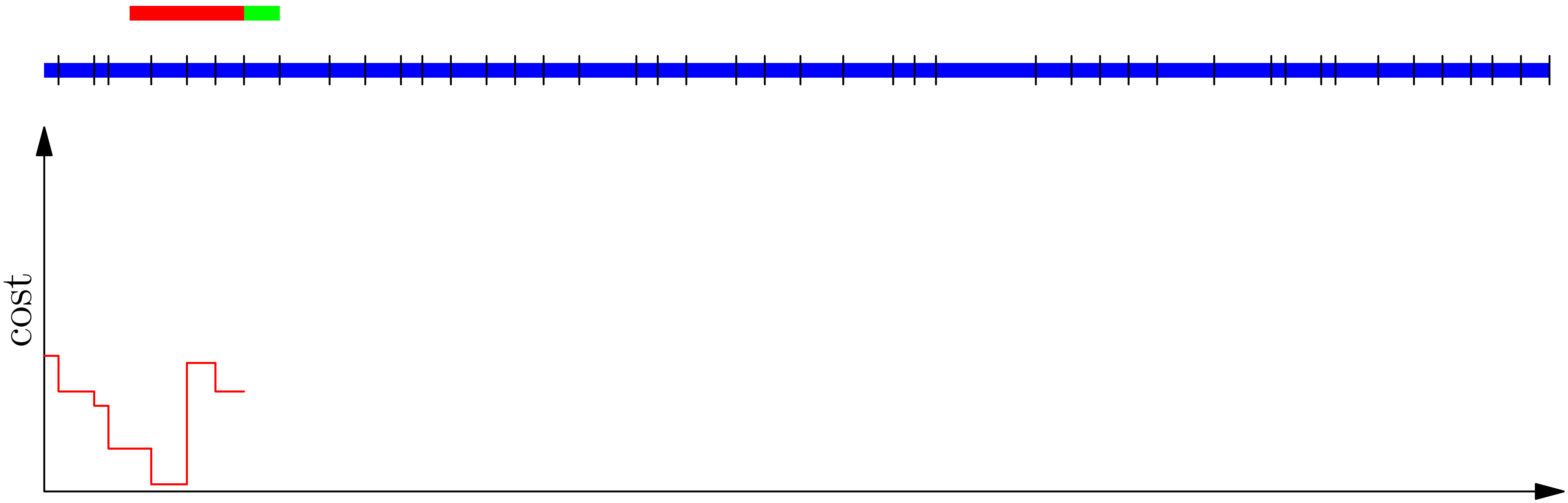
Using Dynamic Programming

- I have a dream that one day this nation will rise up and live out the. . .



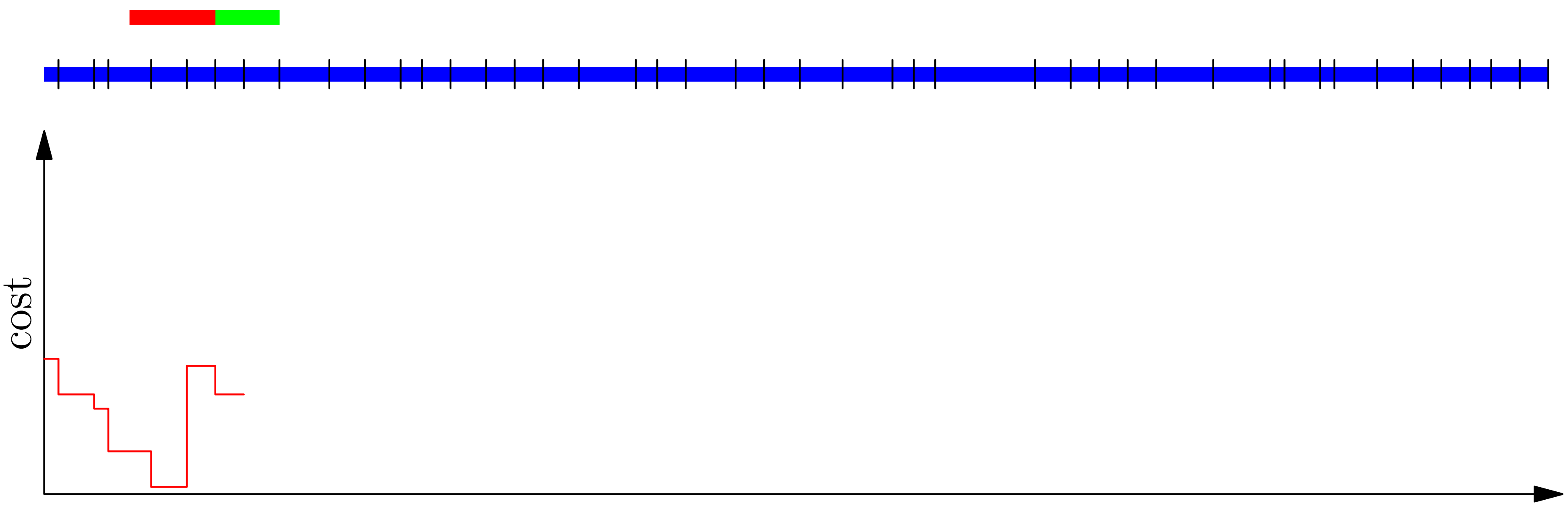
Using Dynamic Programming

- I have a dream that one day this nation will rise up and live out the. . .



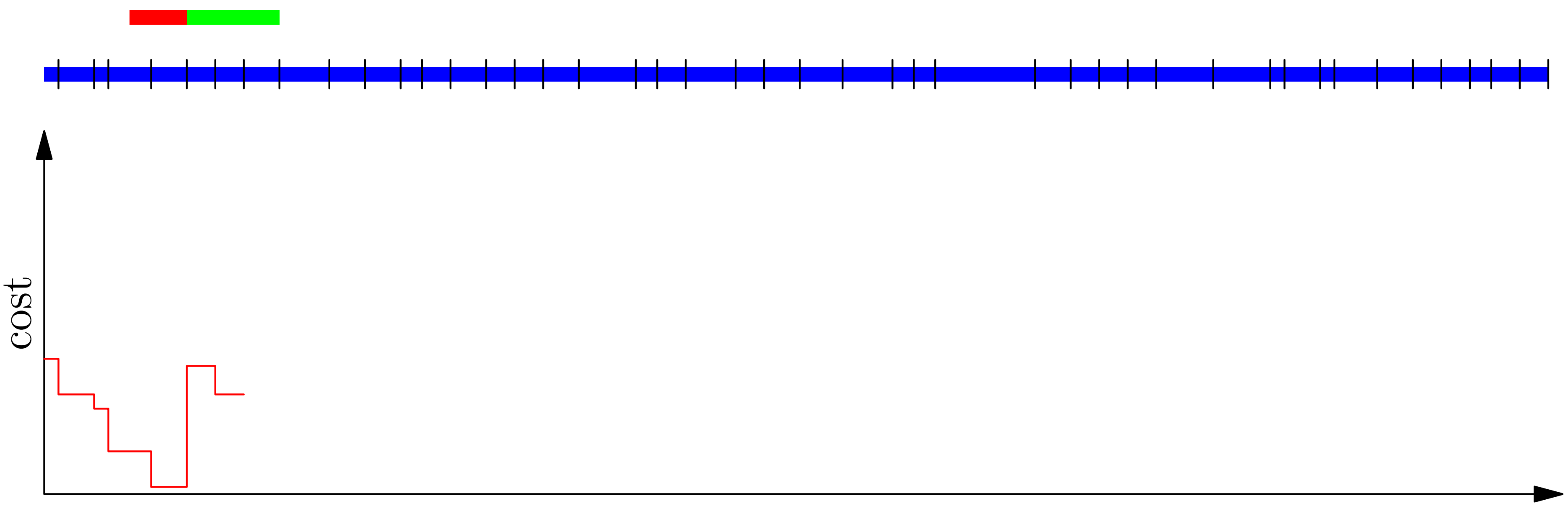
Using Dynamic Programming

- I have a dream that one day this nation will rise up and live out the. . .



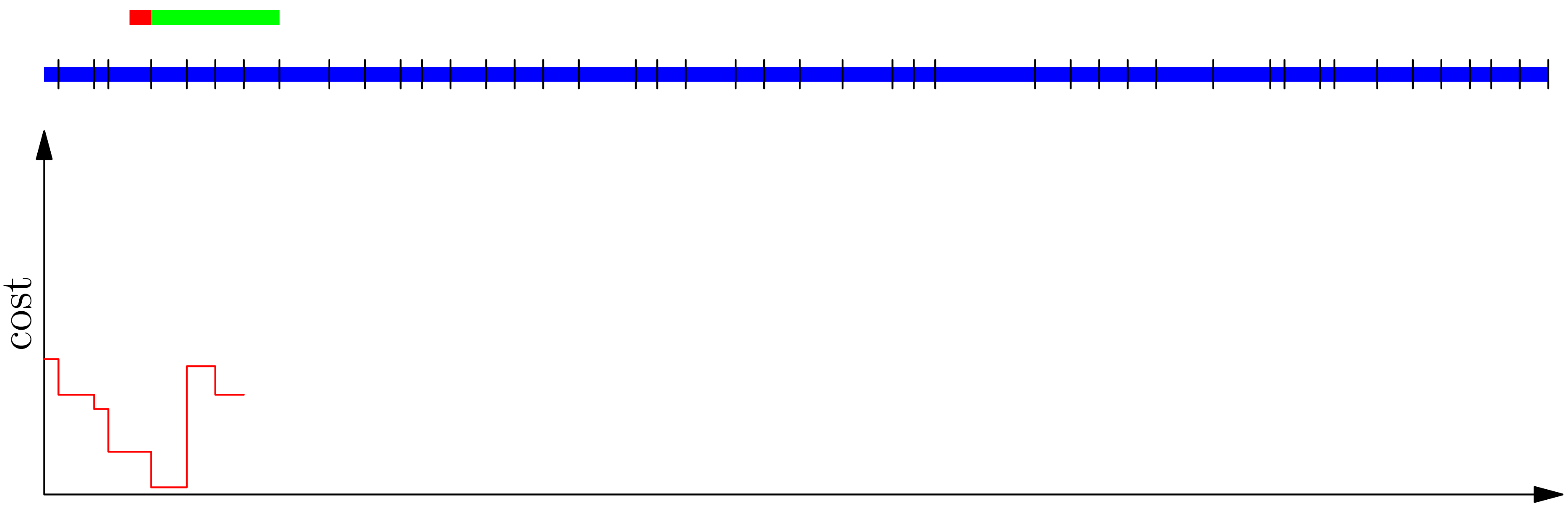
Using Dynamic Programming

- I have a dream that one day this nation will rise up and live out the. . .



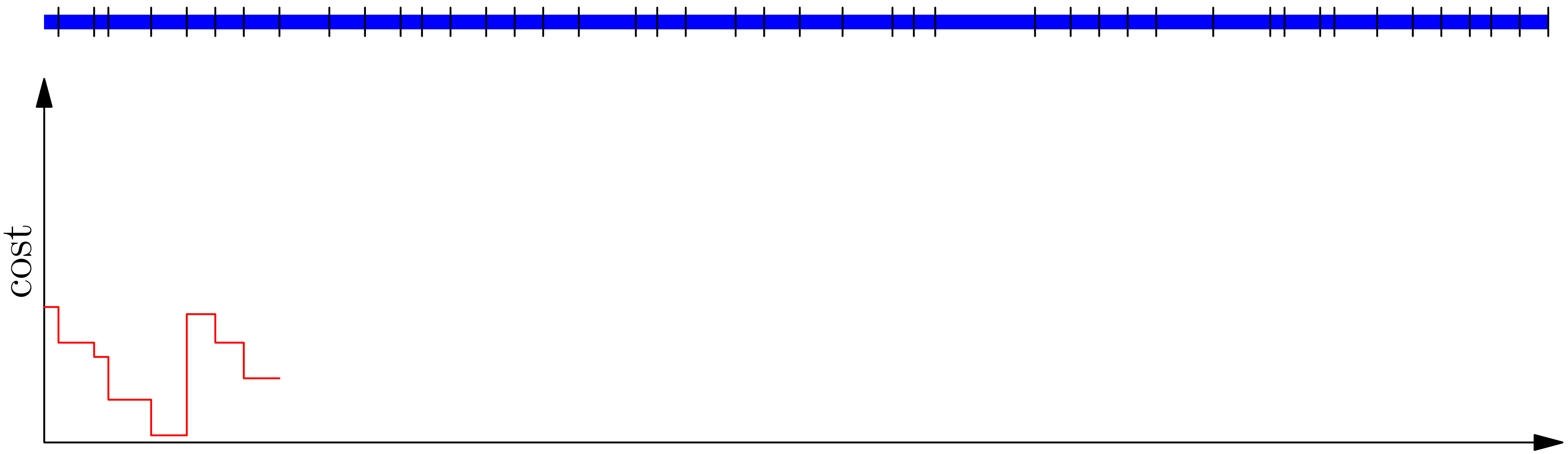
Using Dynamic Programming

- I have a dream that one day this nation will rise up and live out the. . .



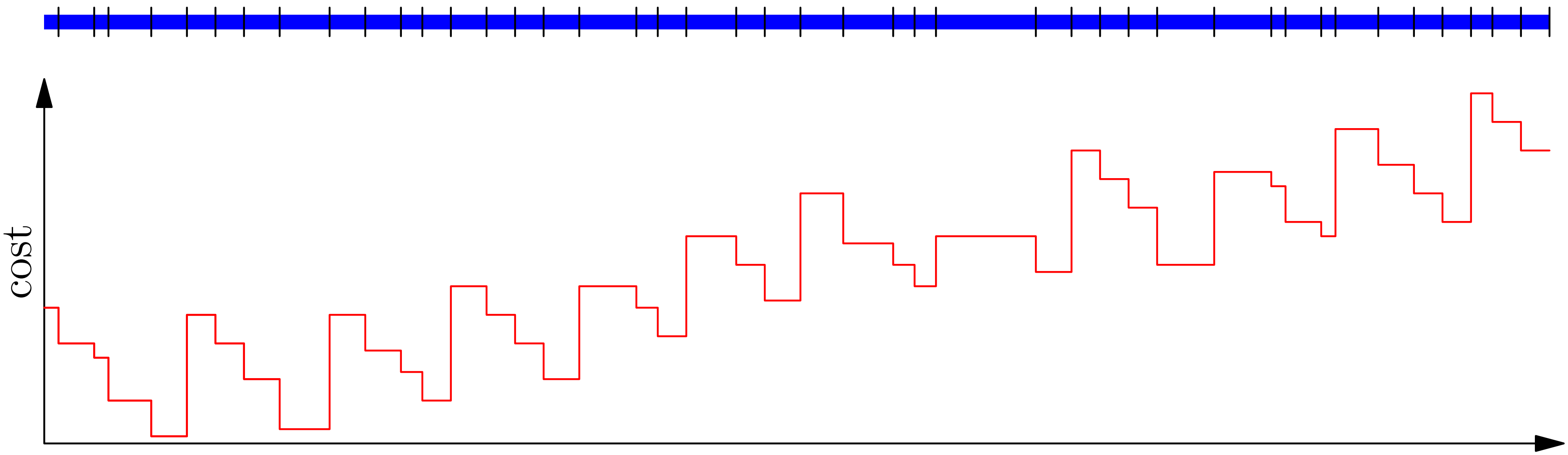
Using Dynamic Programming

- I have a dream that one day this nation will rise up and live out the. . .



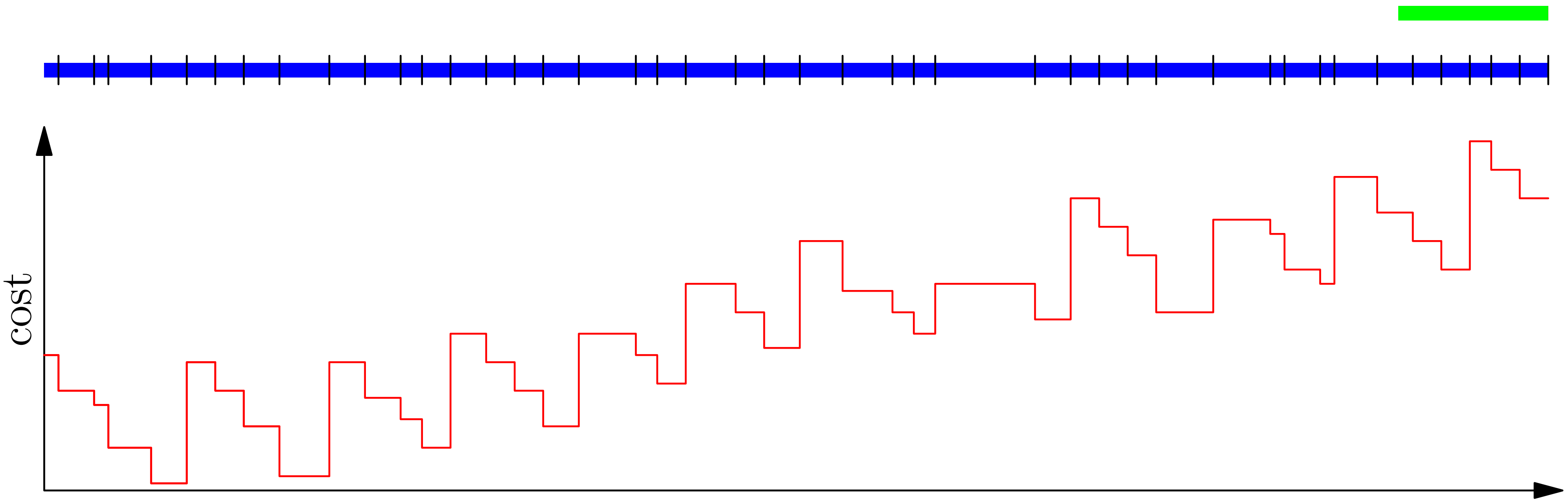
Using Dynamic Programming

- I have a dream that one day this nation will rise up and live out the. . .



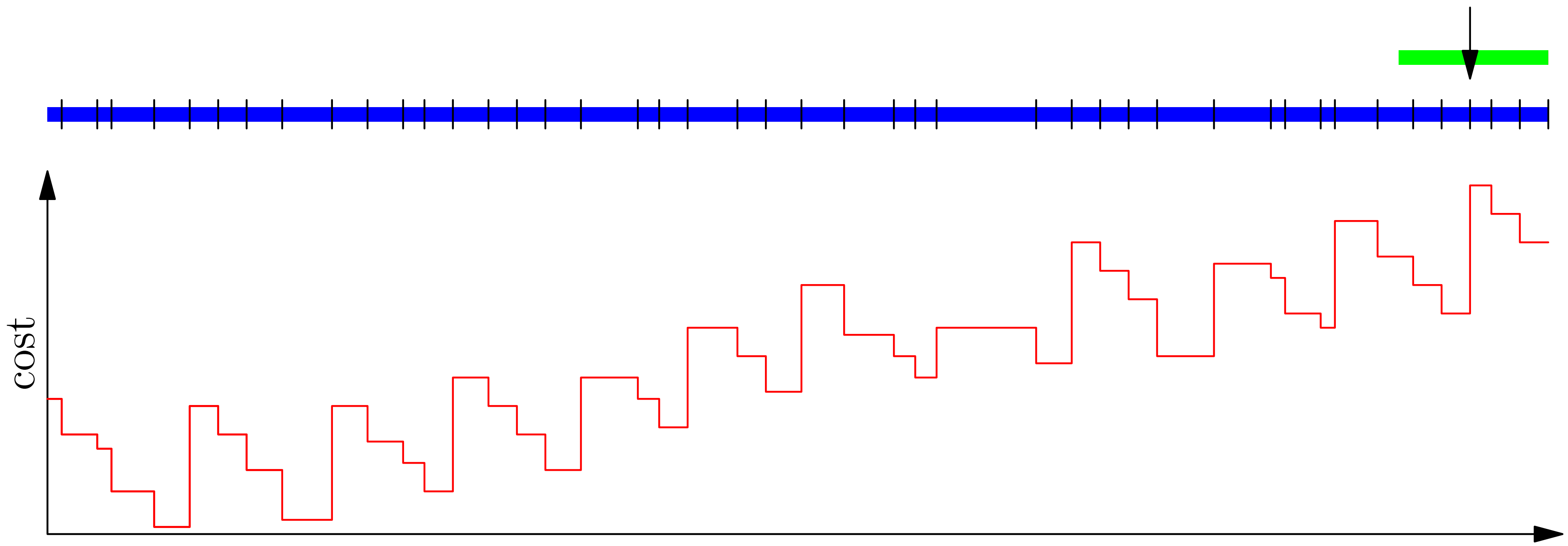
Using Dynamic Programming

- I have a dream that one day this nation will rise up and live out the. . .



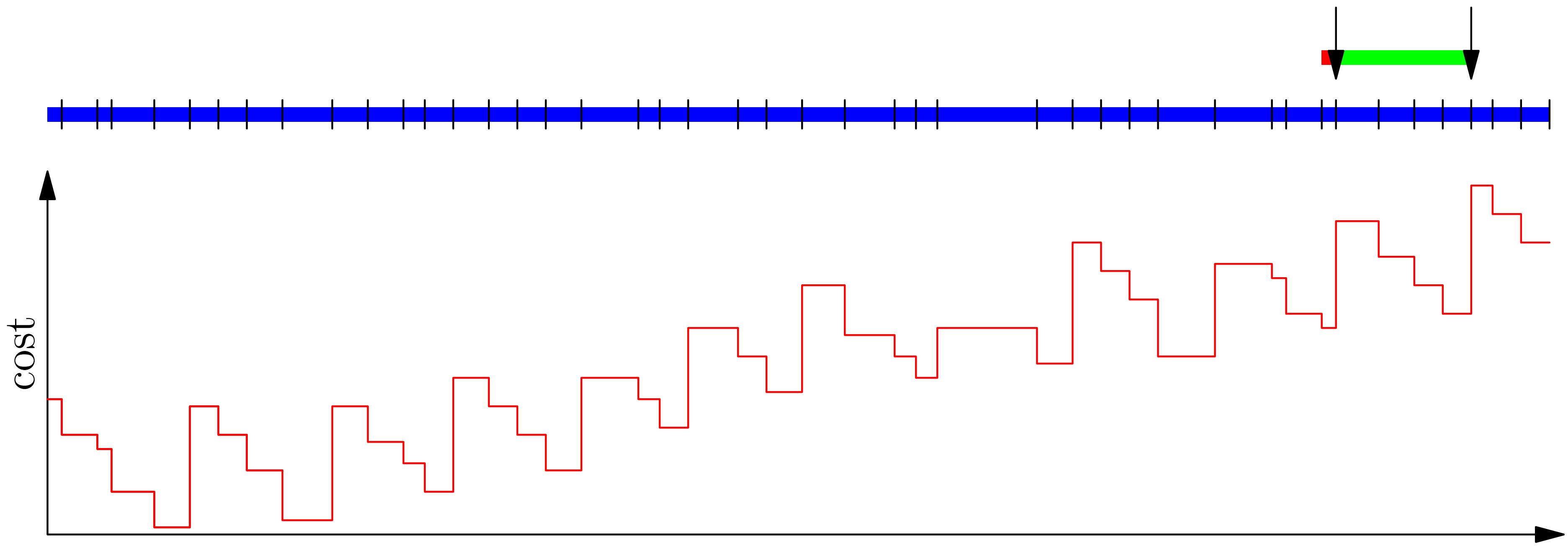
Using Dynamic Programming

- I have a dream that one day this nation will rise up and live out the. . .



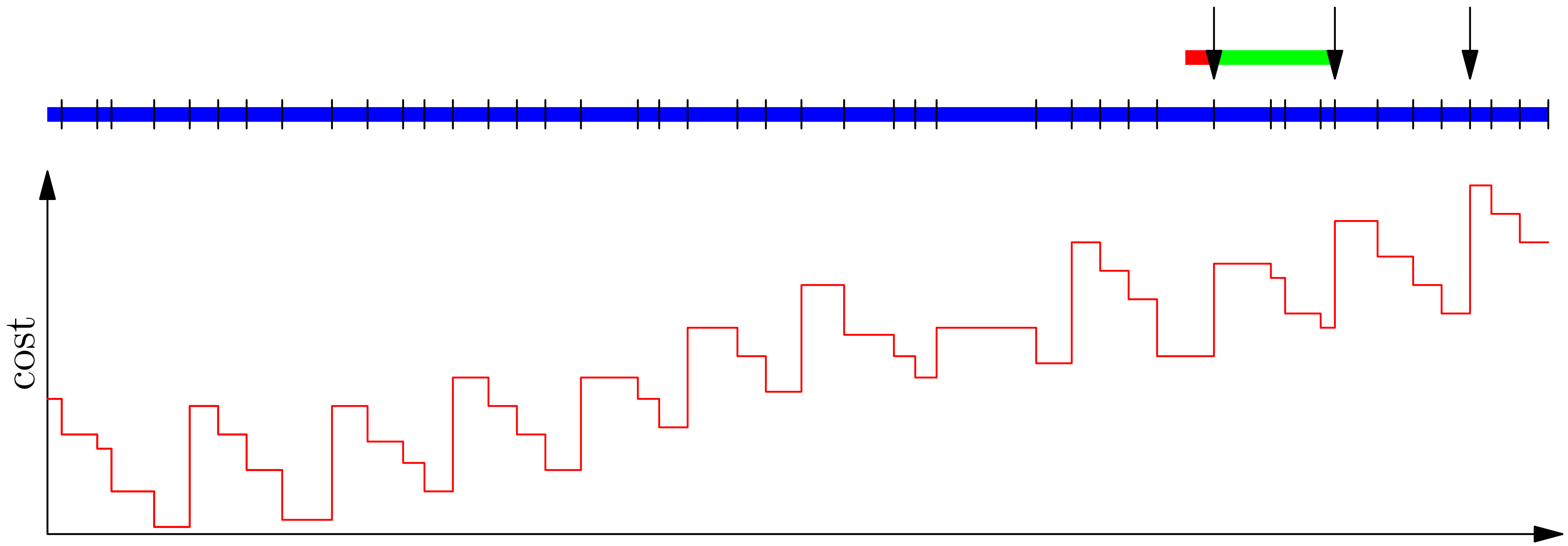
Using Dynamic Programming

- I have a dream that one day this nation will rise up and live out the. . .



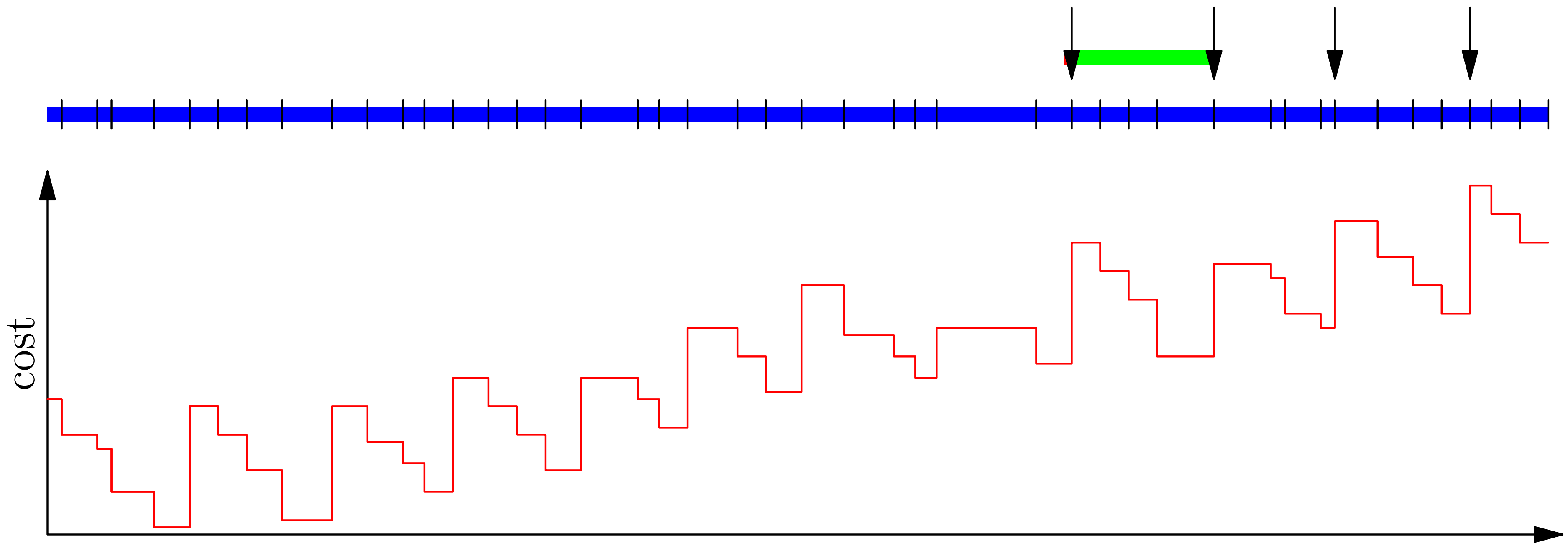
Using Dynamic Programming

- I have a dream that one day this nation will rise up and live out the. . .



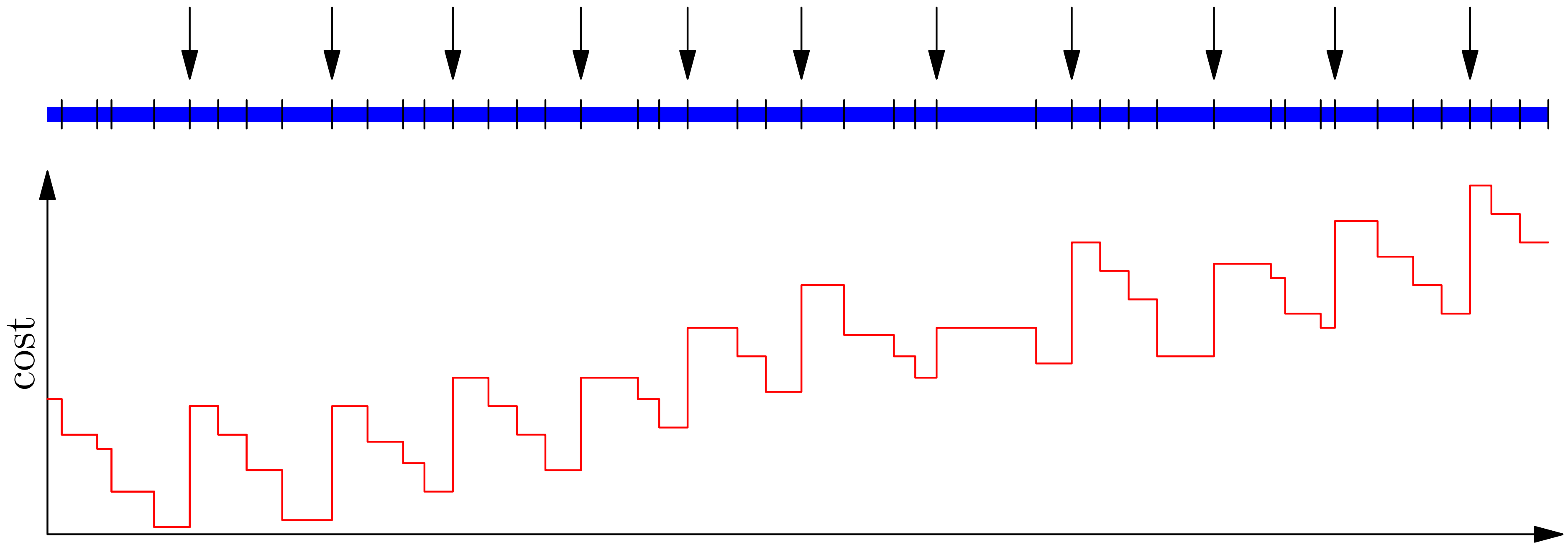
Using Dynamic Programming

- I have a dream that one day this nation will rise up and live out the. . .



Using Dynamic Programming

- I have a dream that one day this nation will rise up and live out the. . .



Real Word Breaking

- In advanced word processing you care about hyphenation, large gaps at the end of lines, etc.
- These all affect the way you would assign costs
- Dynamic programming is used in LaTeX to produce nice line breaks
- A similar algorithm is used to produce nice page breaks

Real Word Breaking

- In advanced word processing you care about hyphenation, large gaps at the end of lines, etc.
- These all affect the way you would assign costs
- Dynamic programming is used in LaTeX to produce nice line breaks
- A similar algorithm is used to produce nice page breaks

Real Word Breaking

- In advanced word processing you care about hyphenation, large gaps at the end of lines, etc.
- These all affect the way you would assign costs
- Dynamic programming is used in LaTeX to produce nice line breaks
- A similar algorithm is used to produce nice page breaks

Real Word Breaking

- In advanced word processing you care about hyphenation, large gaps at the end of lines, etc.
- These all affect the way you would assign costs
- Dynamic programming is used in LaTeX to produce nice line breaks
- A similar algorithm is used to produce nice page breaks

Inexact Matching

- A second example of dynamic programming is to find inexact matches
- The edit distance between two strings is the number of changes needed to move from one string to another
- The exact metric depends on the application, but might include number of substitutions, insertions and deletions
- This has many applications, e.g. in genomics to see what DNA strings (or proteins) are related

Inexact Matching

- A second example of dynamic programming is to find inexact matches
- The edit distance between two strings is the number of changes needed to move from one string to another
- The exact metric depends on the application, but might include number of substitutions, insertions and deletions
- This has many applications, e.g. in genomics to see what DNA strings (or proteins) are related

Inexact Matching

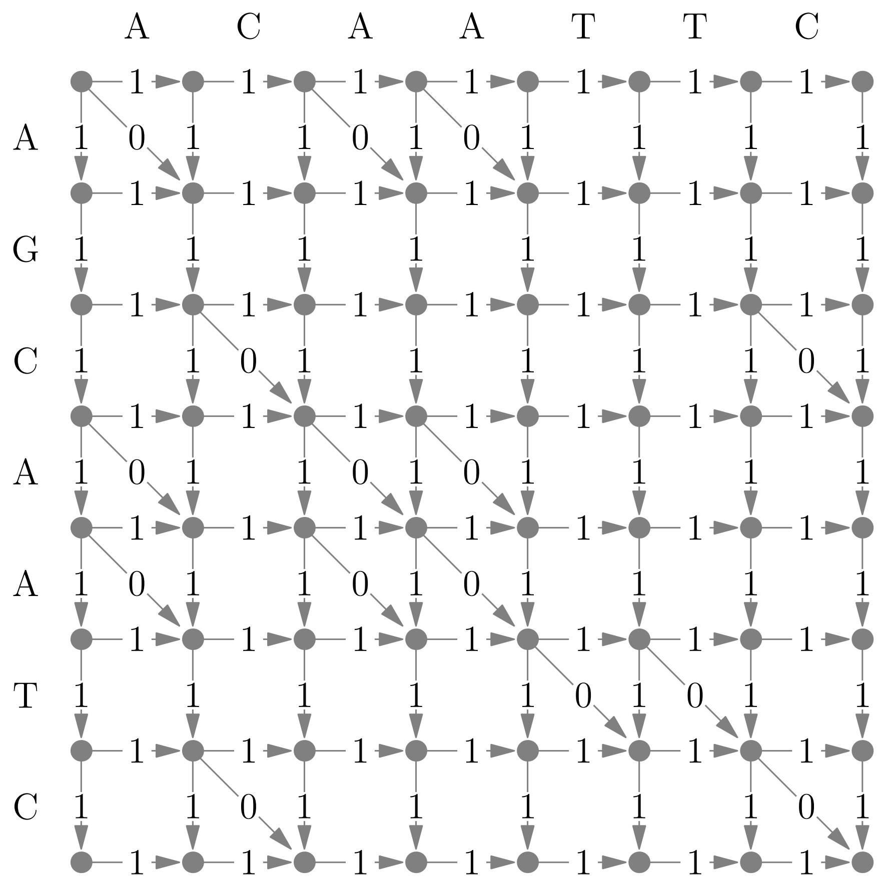
- A second example of dynamic programming is to find inexact matches
- The edit distance between two strings is the number of changes needed to move from one string to another
- The exact metric depends on the application, but might include number of substitutions, insertions and deletions
- This has many applications, e.g. in genomics to see what DNA strings (or proteins) are related

Inexact Matching

- A second example of dynamic programming is to find inexact matches
- The edit distance between two strings is the number of changes needed to move from one string to another
- The exact metric depends on the application, but might include number of substitutions, insertions and deletions
- This has many applications, e.g. in genomics to see what DNA strings (or proteins) are related

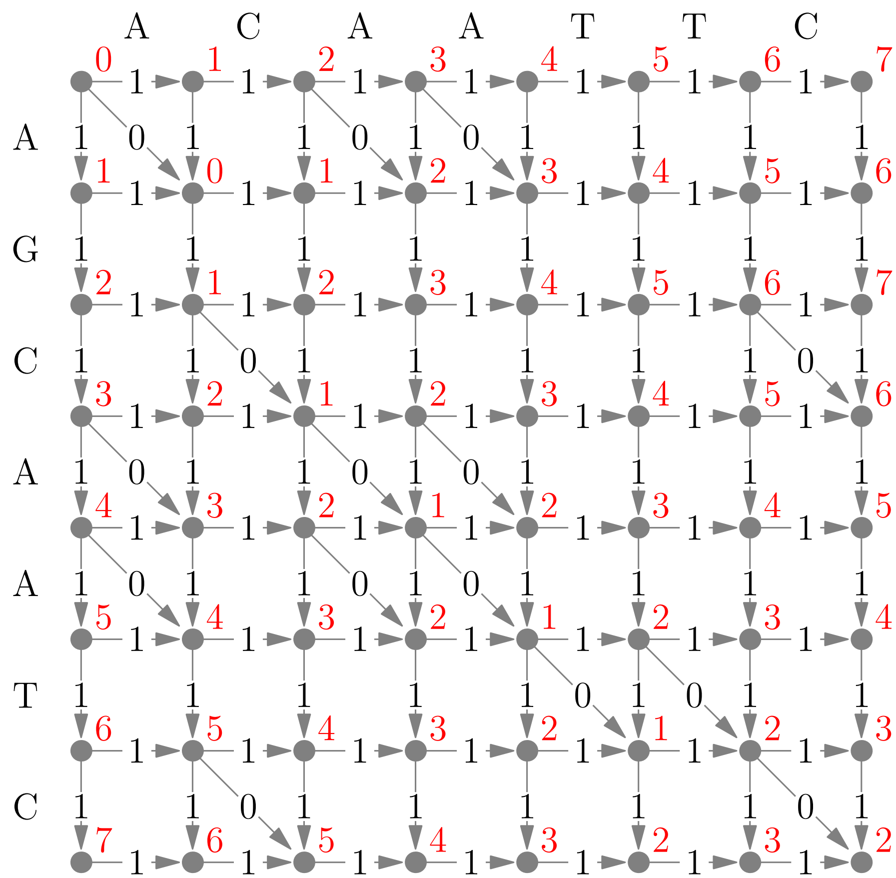
Edit Distance

- What is the minimum edit distance between ACAATTC and AGCAATC



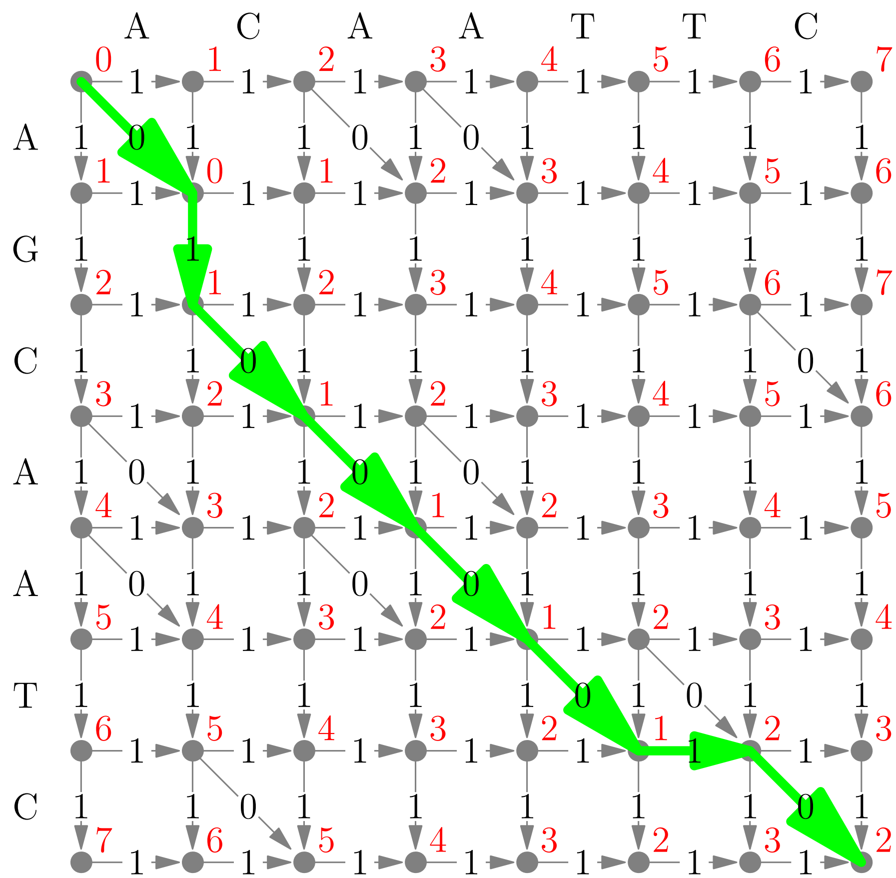
Edit Distance

- What is the minimum edit distance between ACAATTC and AGCAATC



Edit Distance

- What is the minimum edit distance between ACAATTC and AGCAATC



Dijkstra's Algorithm

- We saw early Dijkstra's algorithm for find the minimum distance between a source and destination node
- We grouped this with the greedy algorithms as we choose the next node to add to the minimum-distance spanning tree to be the closest node to the source we could access
- However, we should perhaps more rightly identify it as using a dynamic programming strategy as we are building up the cost of getting of the partial solution to reach a node
- We use the greedy strategy to ensure that we always find the shorter paths first

Dijkstra's Algorithm

- We saw early Dijkstra's algorithm for find the minimum distance between a source and destination node
- We grouped this with the greedy algorithms as we choose the next node to add to the minimum-distance spanning tree to be the closest node to the source we could access
- However, we should perhaps more rightly identify it as using a dynamic programming strategy as we are building up the cost of getting of the partial solution to reach a node
- We use the greedy strategy to ensure that we always find the shorter paths first

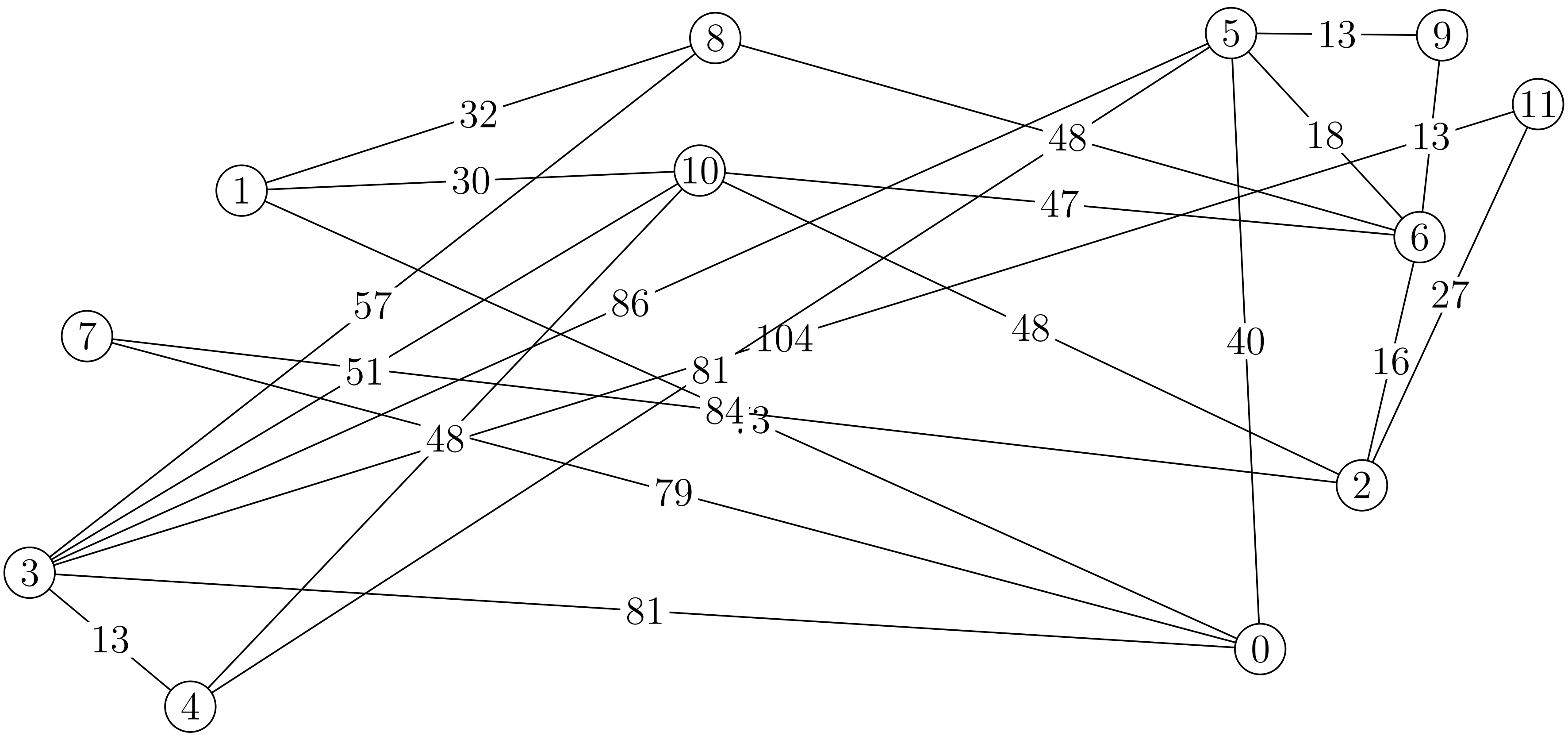
Dijkstra's Algorithm

- We saw early Dijkstra's algorithm for find the minimum distance between a source and destination node
- We grouped this with the greedy algorithms as we choose the next node to add to the minimum-distance spanning tree to be the closest node to the source we could access
- However, we should perhaps more rightly identify it as using a dynamic programming strategy as we are building up the cost of getting of the partial solution to reach a node
- We use the greedy strategy to ensure that we always find the shorter paths first

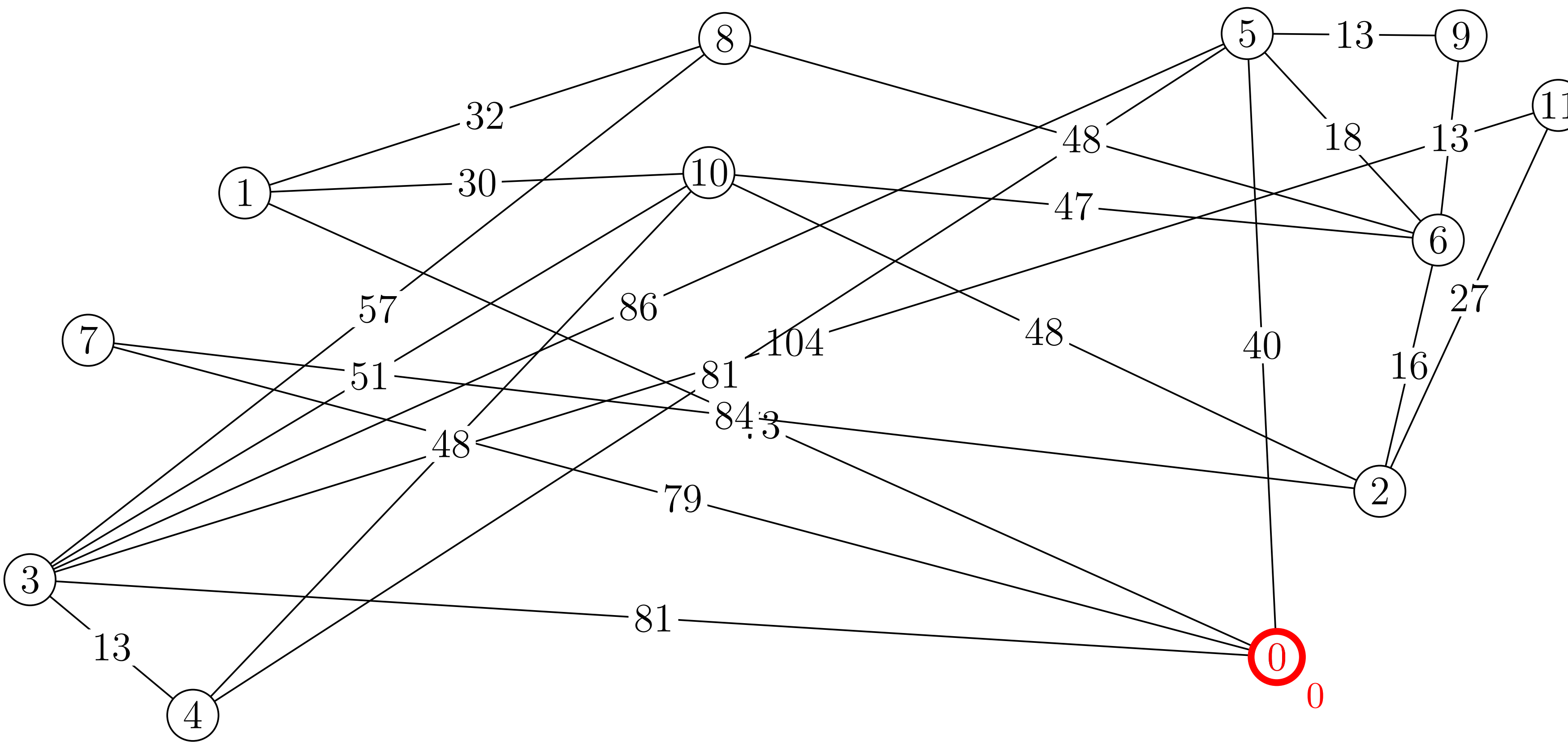
Dijkstra's Algorithm

- We saw early Dijkstra's algorithm for find the minimum distance between a source and destination node
- We grouped this with the greedy algorithms as we choose the next node to add to the minimum-distance spanning tree to be the closest node to the source we could access
- However, we should perhaps more rightly identify it as using a dynamic programming strategy as we are building up the cost of getting of the partial solution to reach a node
- We use the greedy strategy to ensure that we always find the shorter paths first

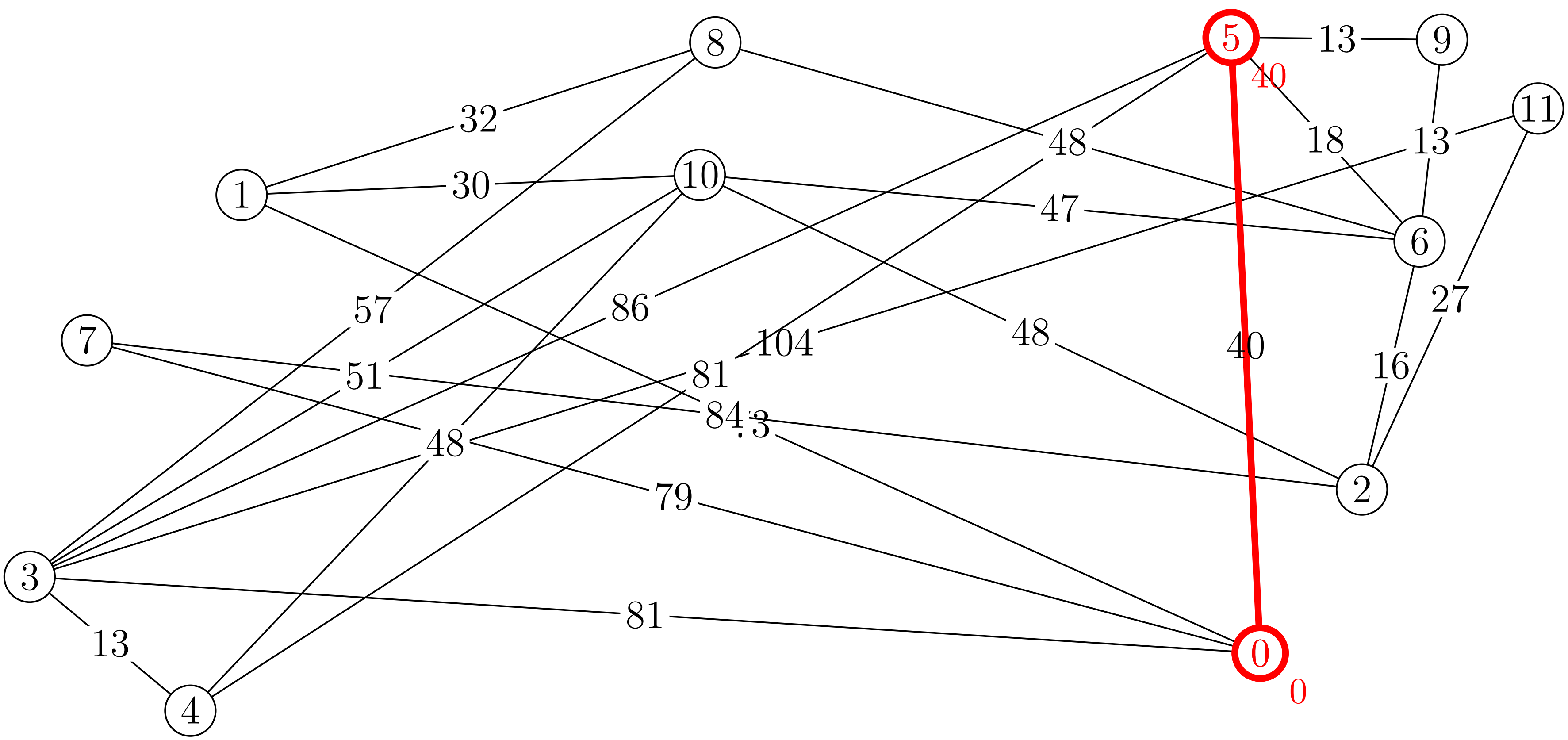
Going from Node 0 to Node 11



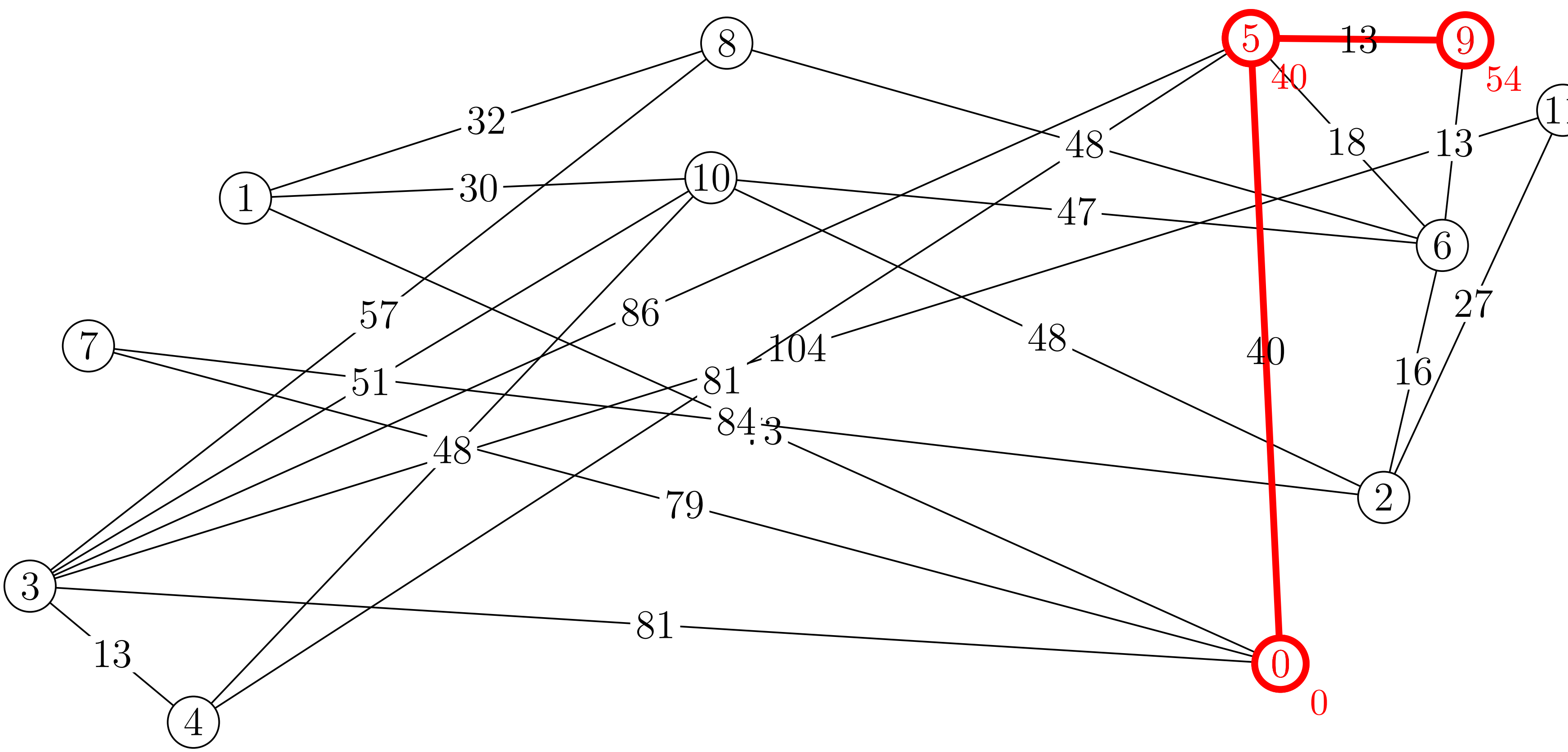
Going from Node 0 to Node 11



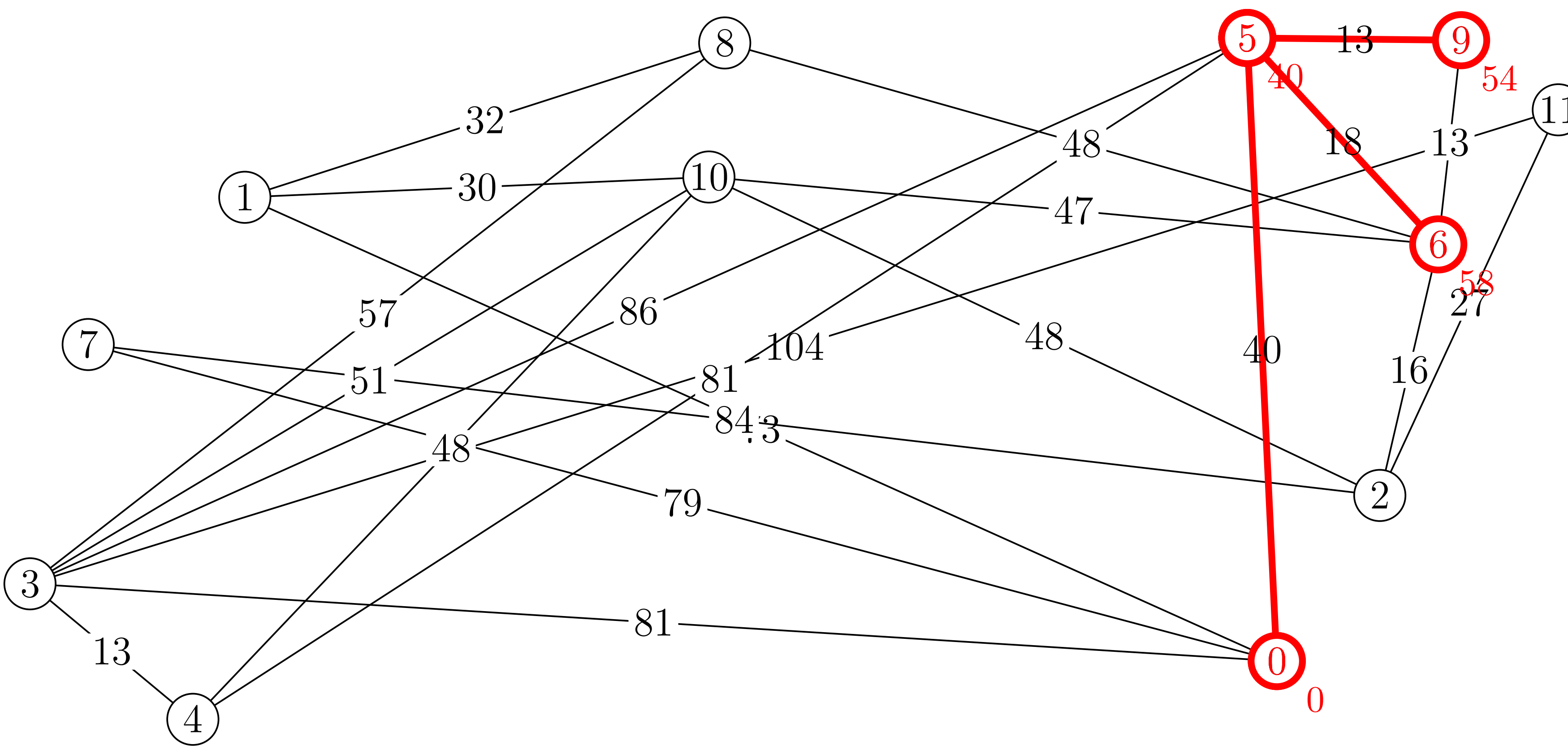
Going from Node 0 to Node 11



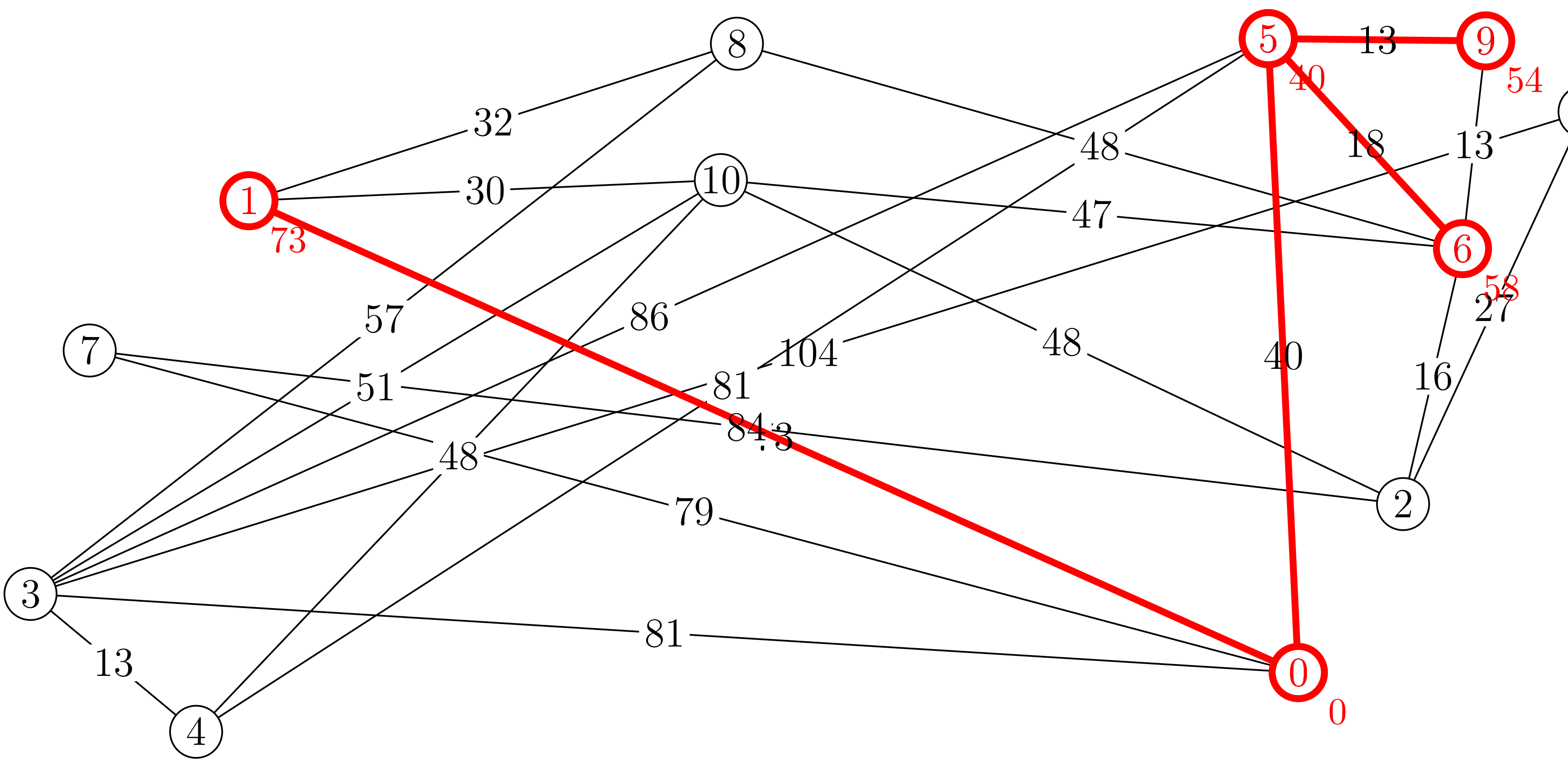
Going from Node 0 to Node 11



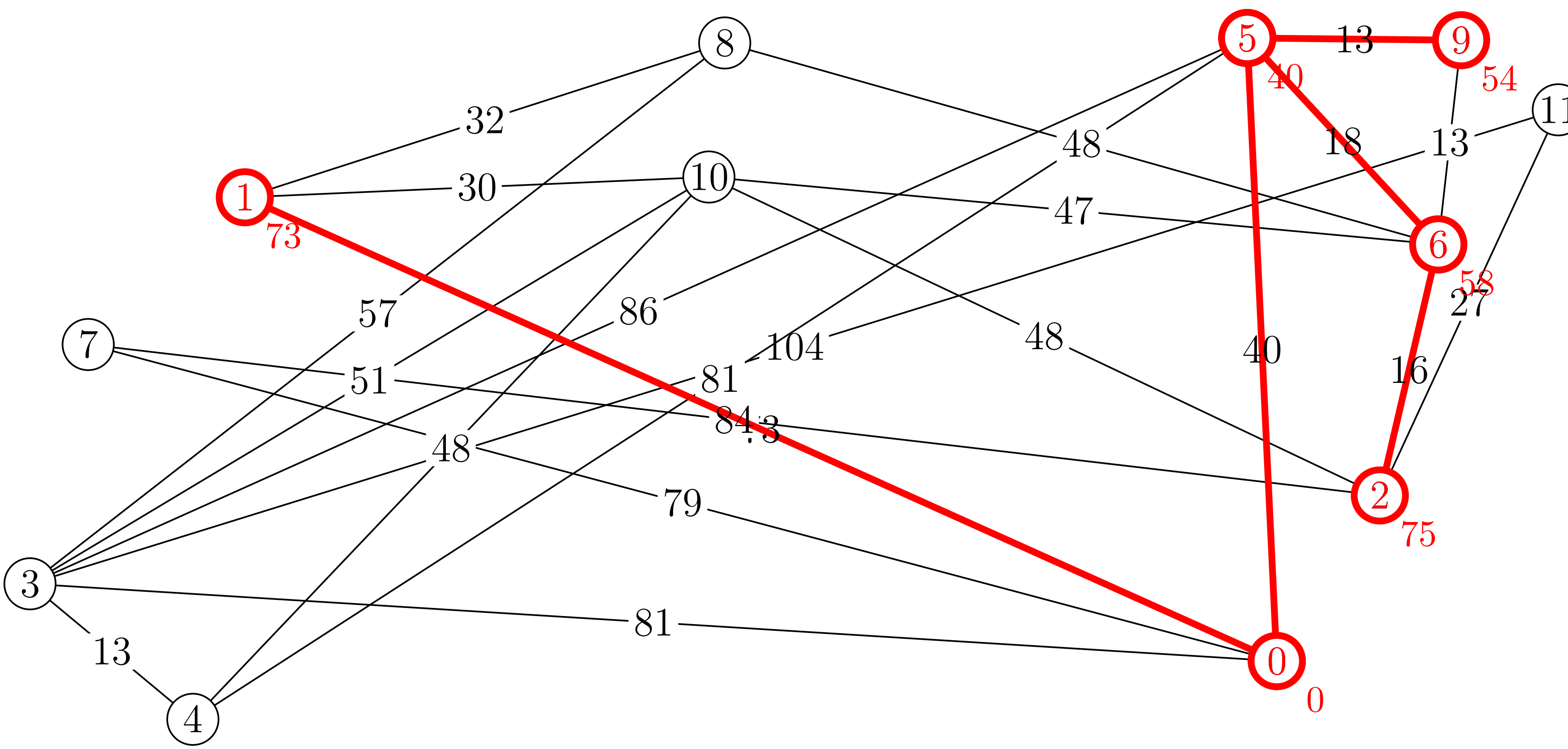
Going from Node 0 to Node 11



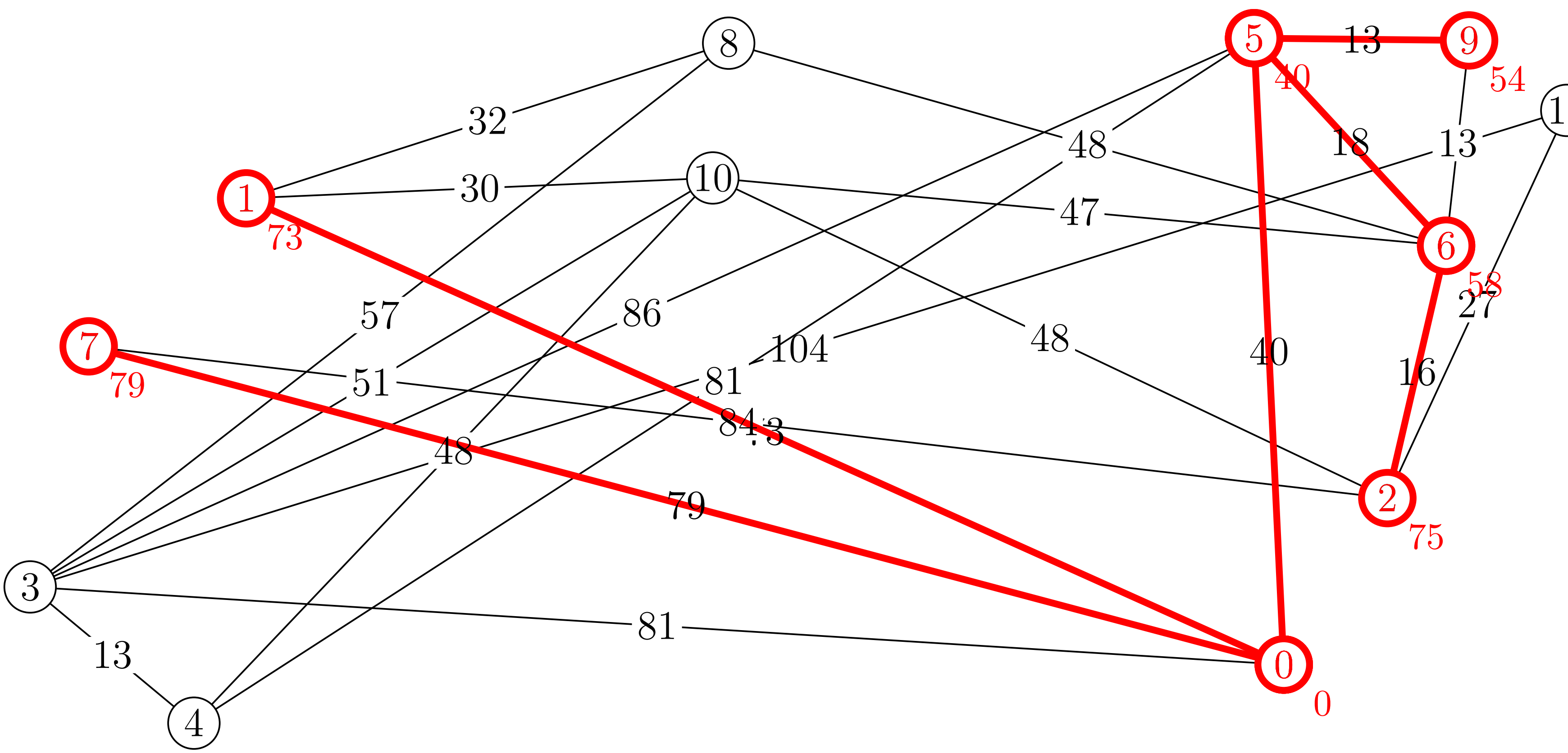
Going from Node 0 to Node 11



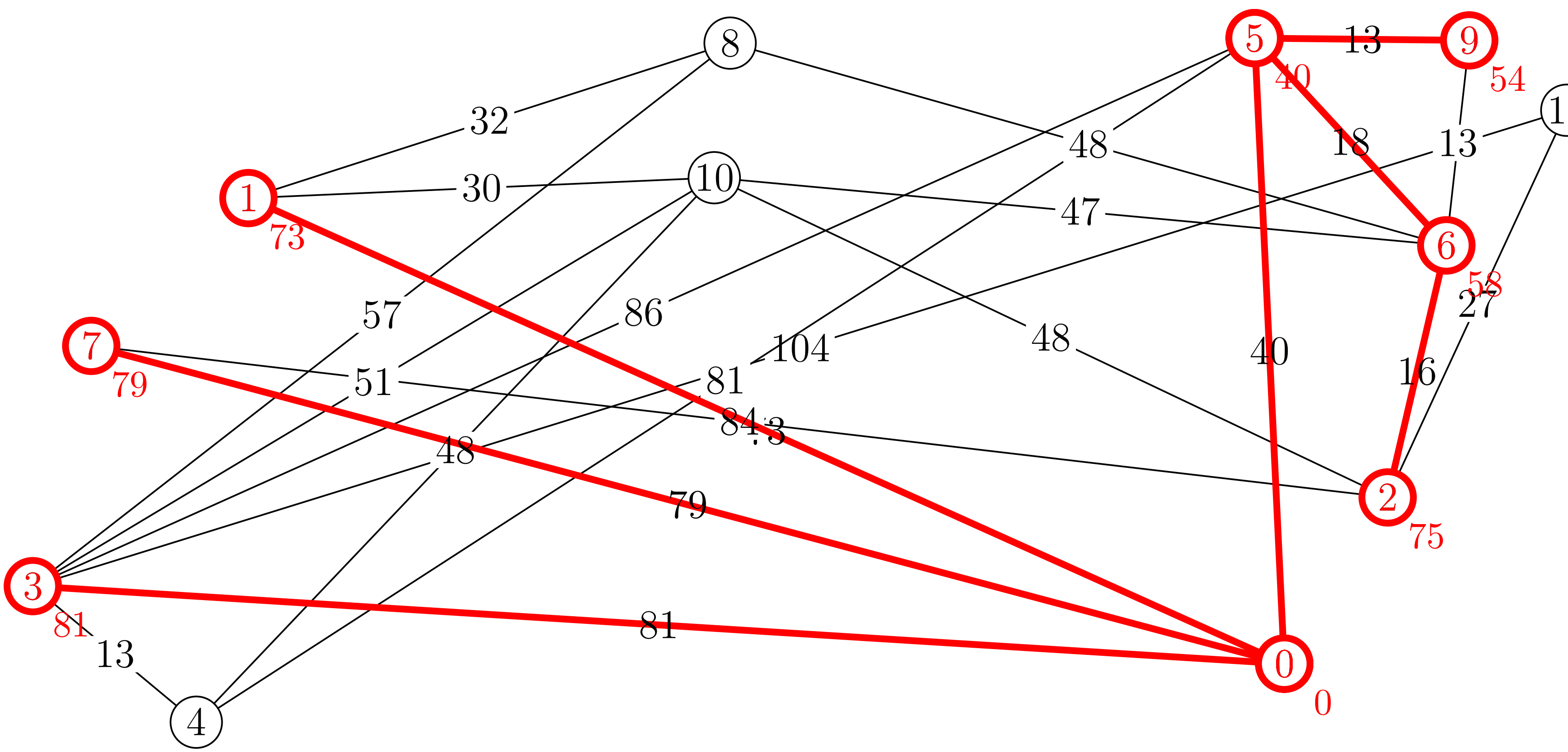
Going from Node 0 to Node 11



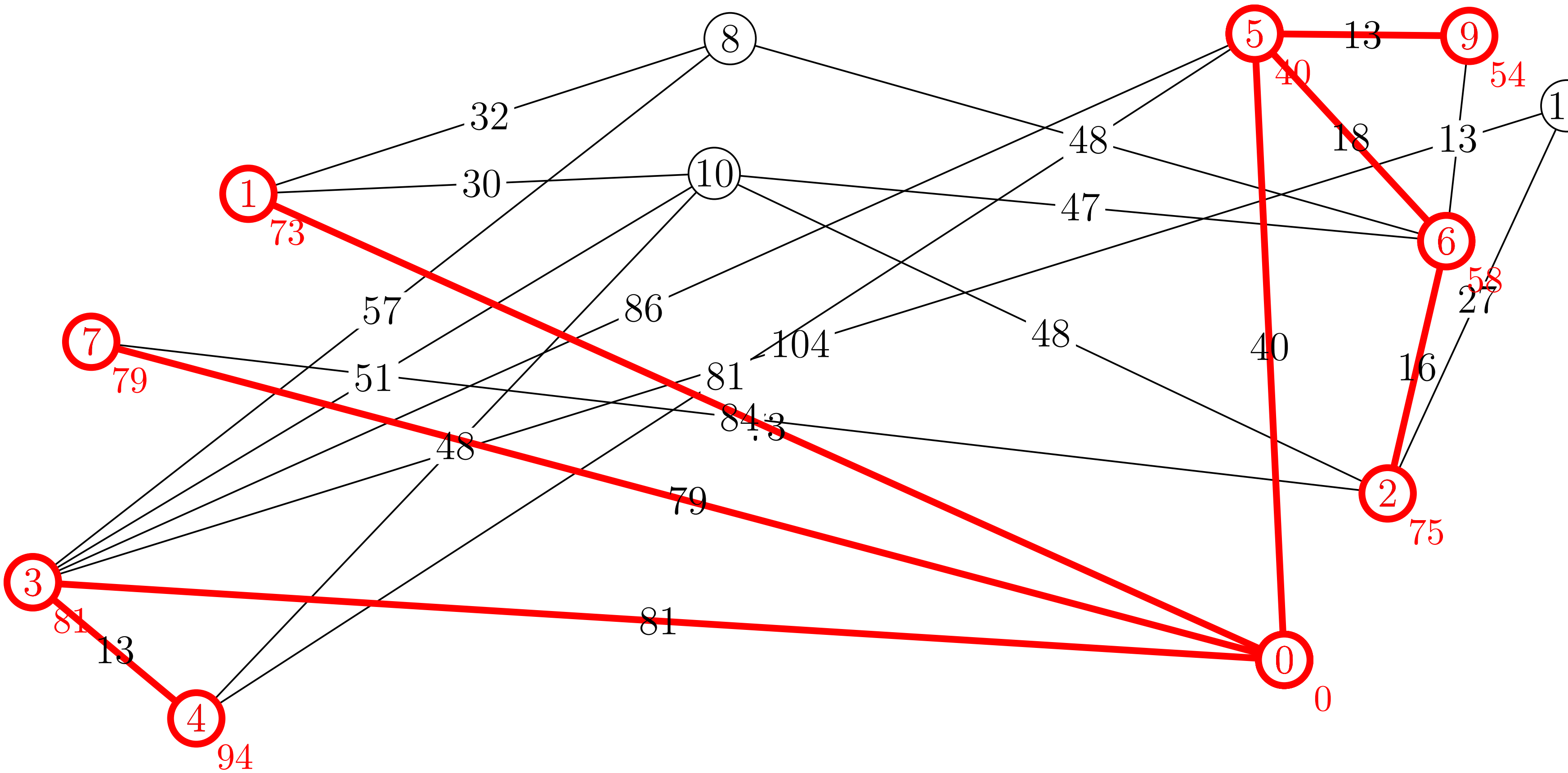
Going from Node 0 to Node 11



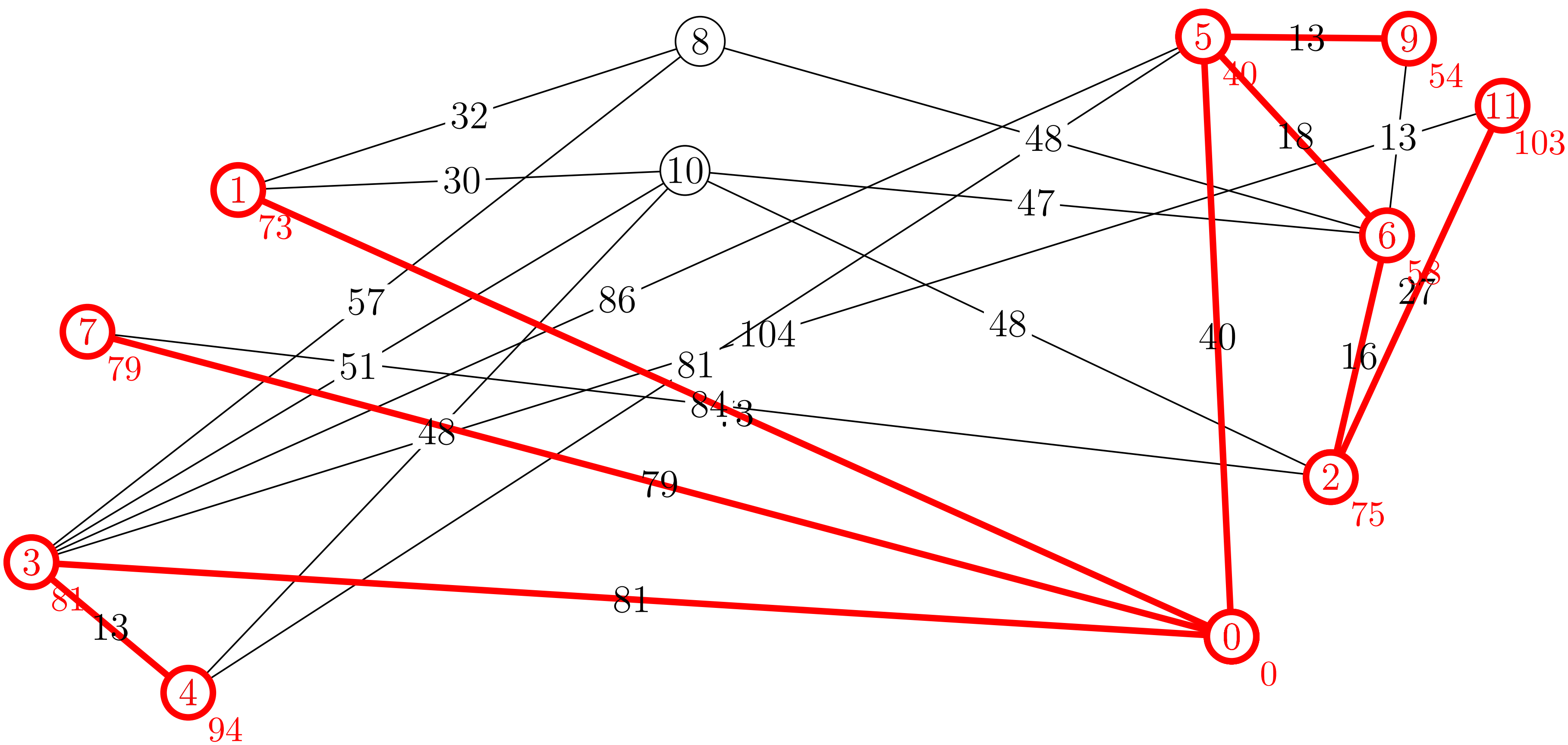
Going from Node 0 to Node 11



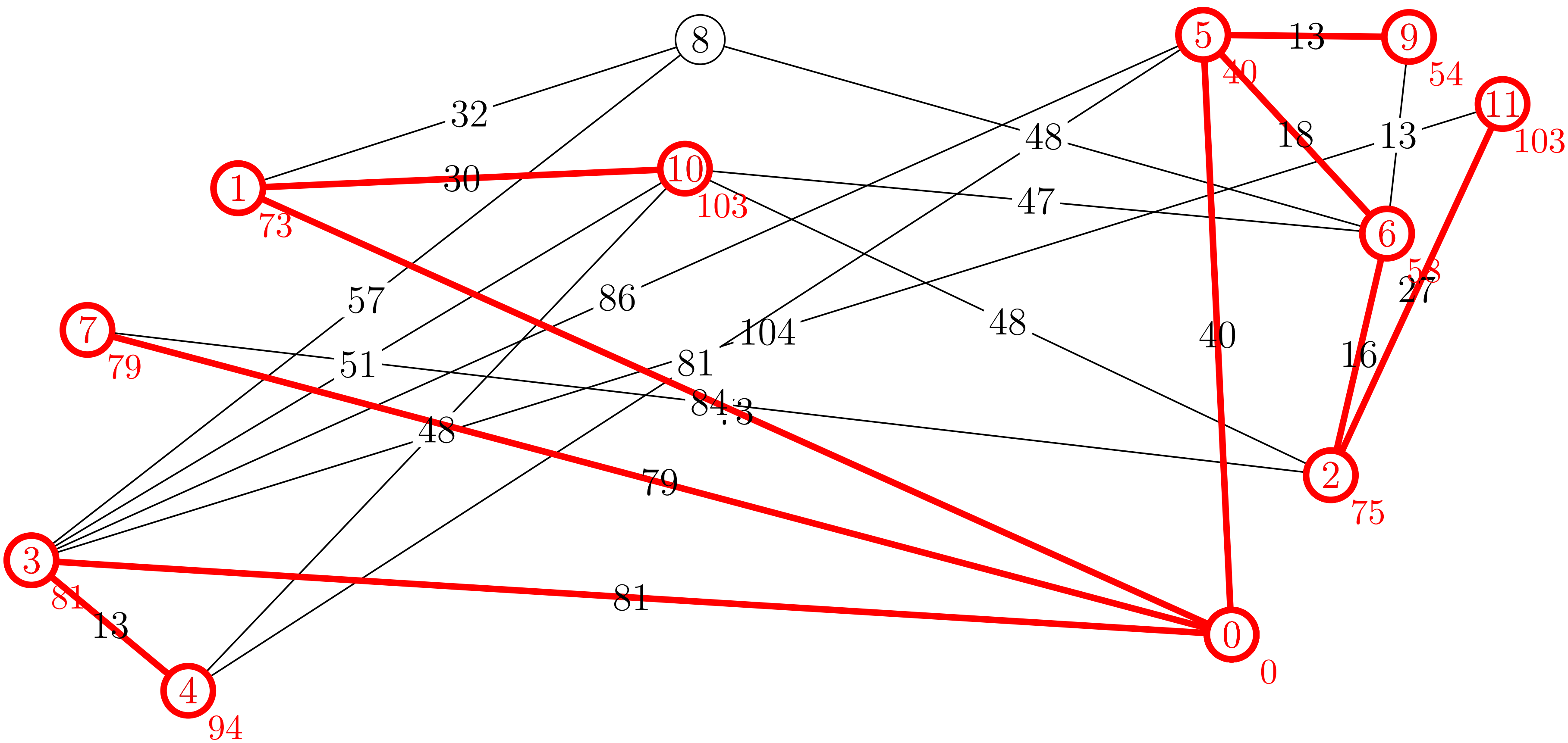
Going from Node 0 to Node 11



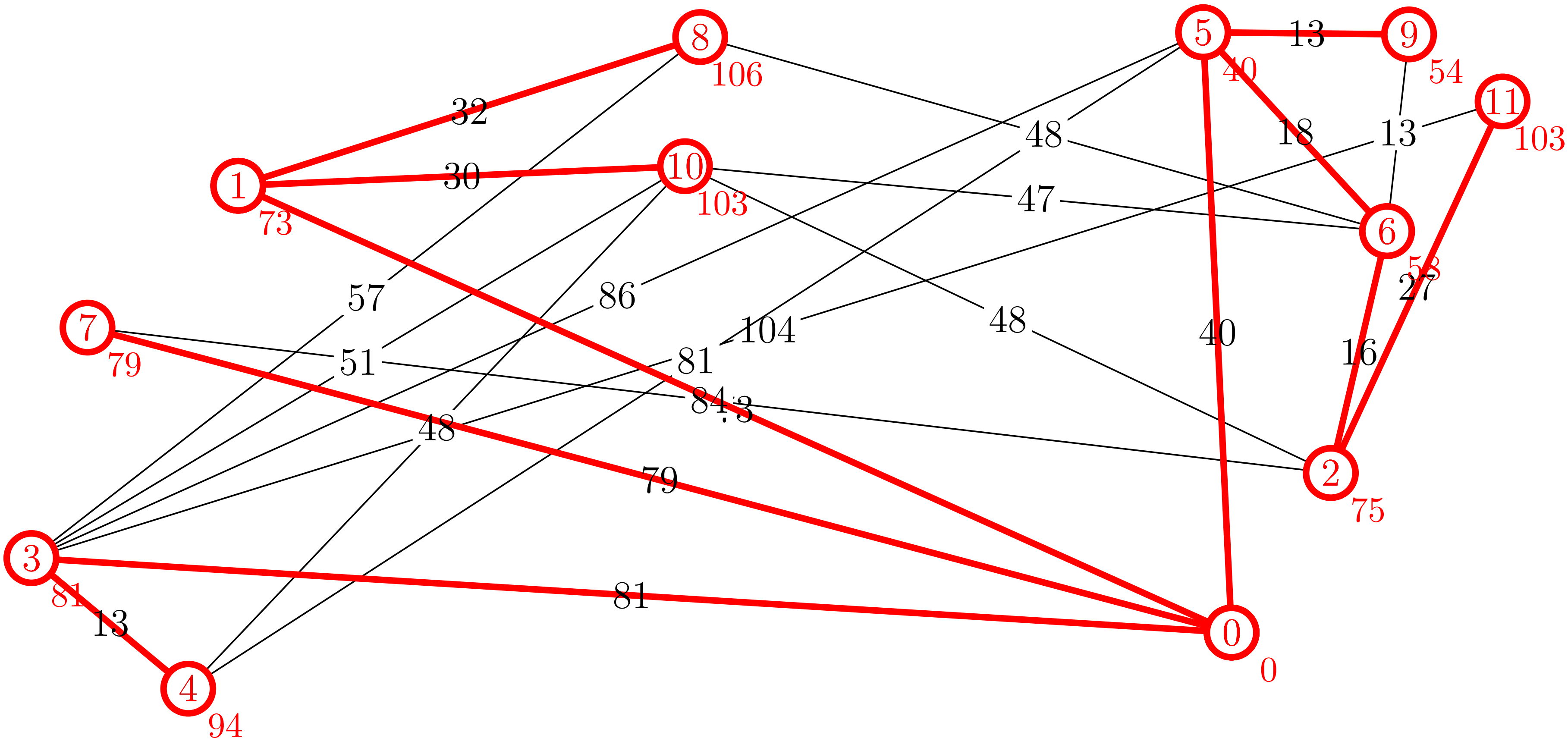
Going from Node 0 to Node 11



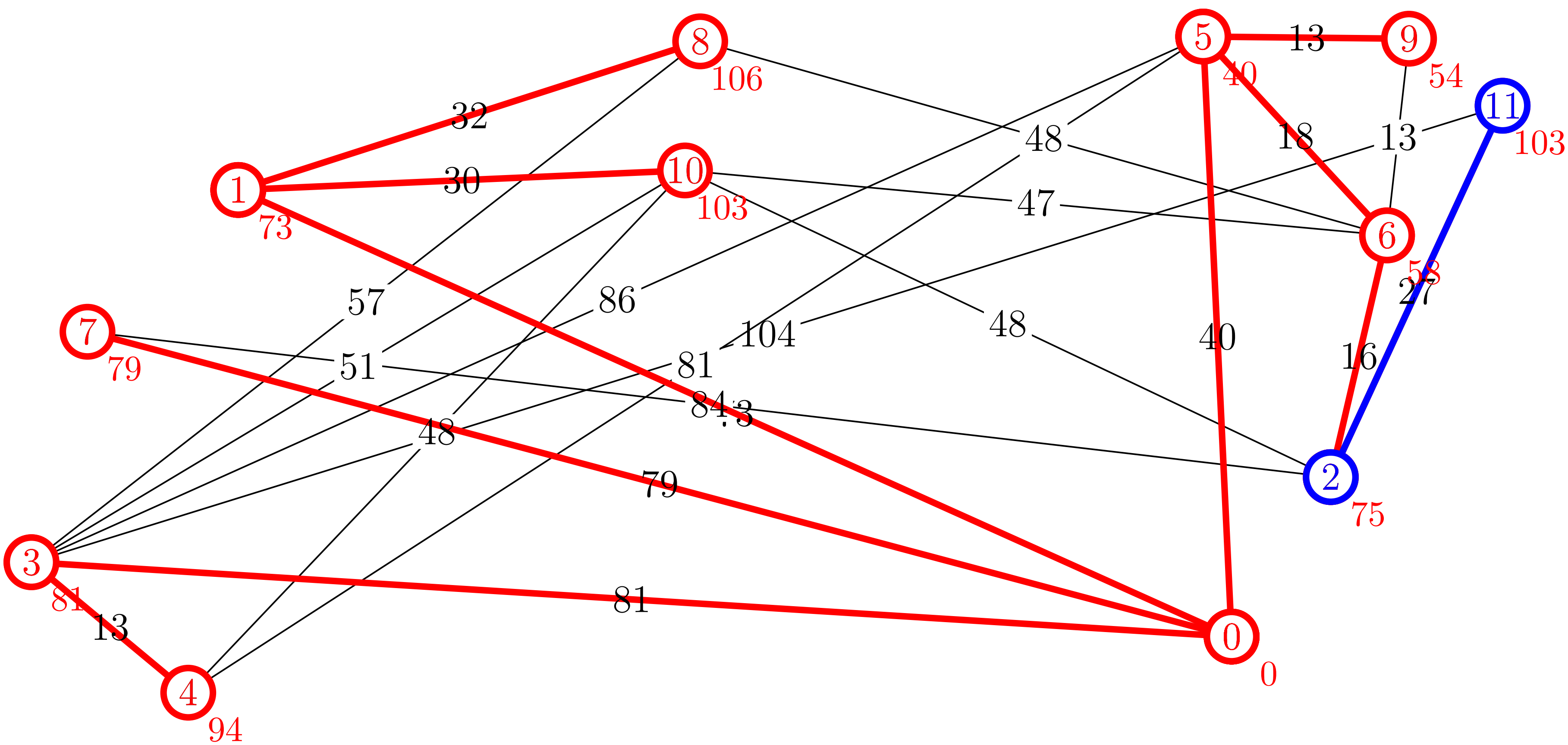
Going from Node 0 to Node 11



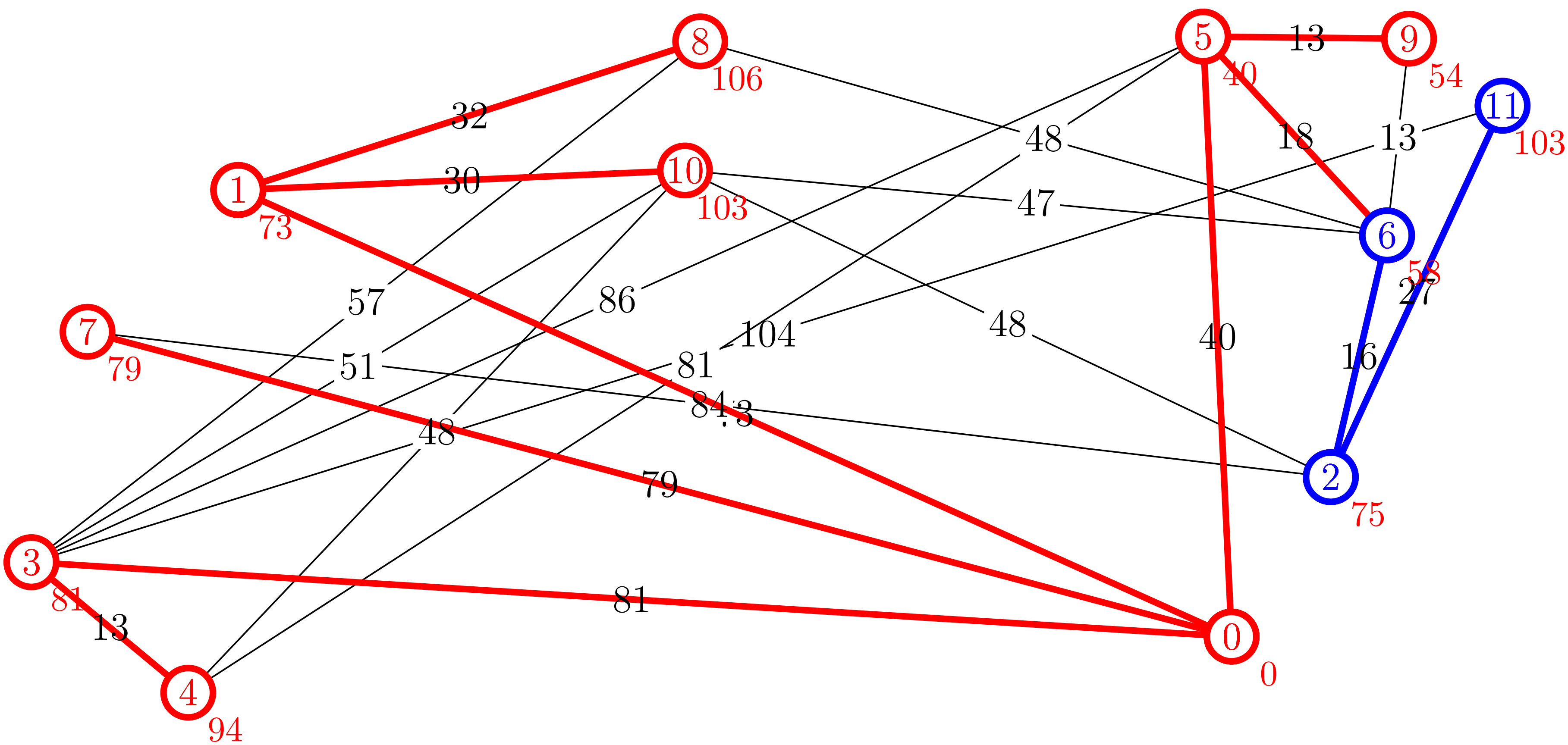
Going from Node 0 to Node 11



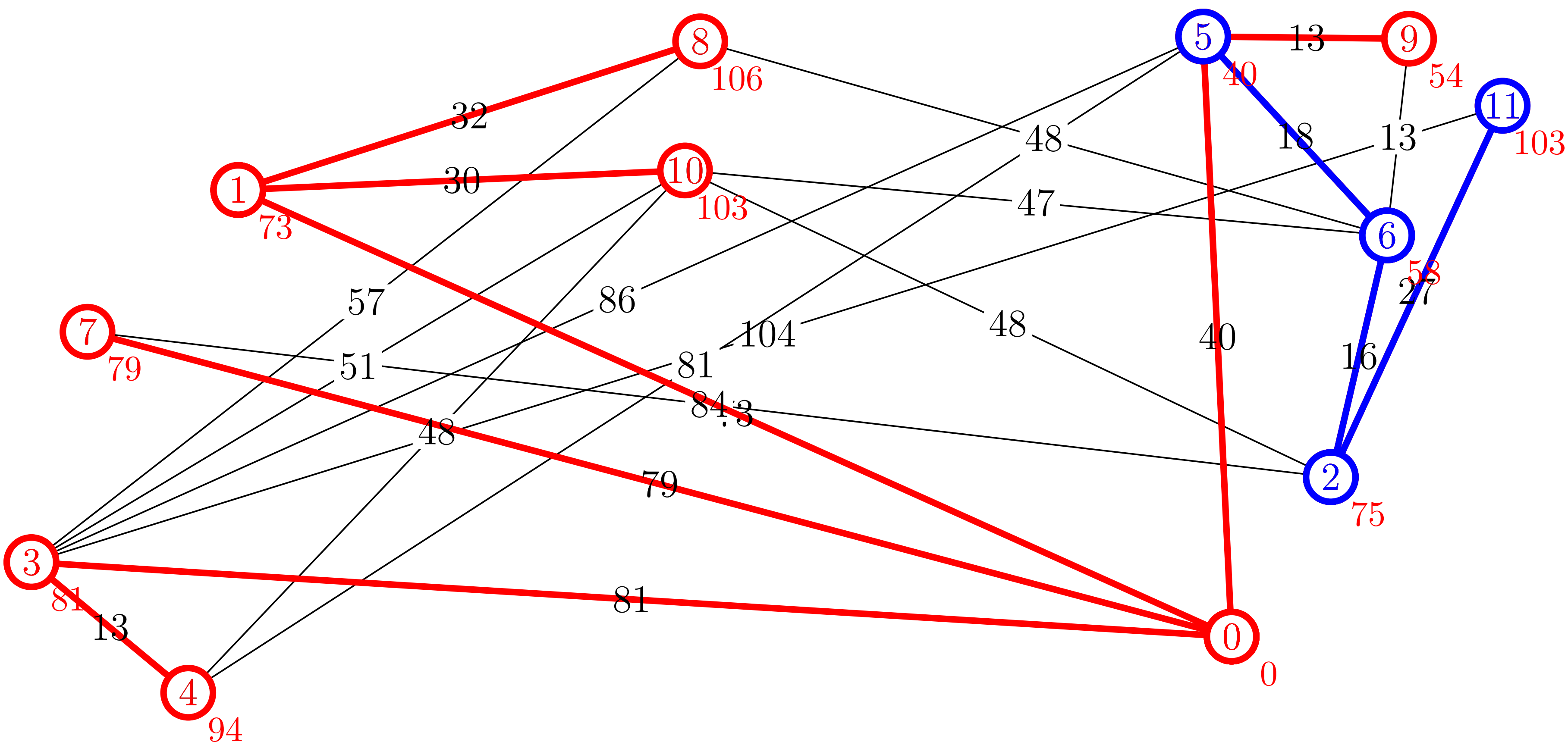
Going from Node 0 to Node 11



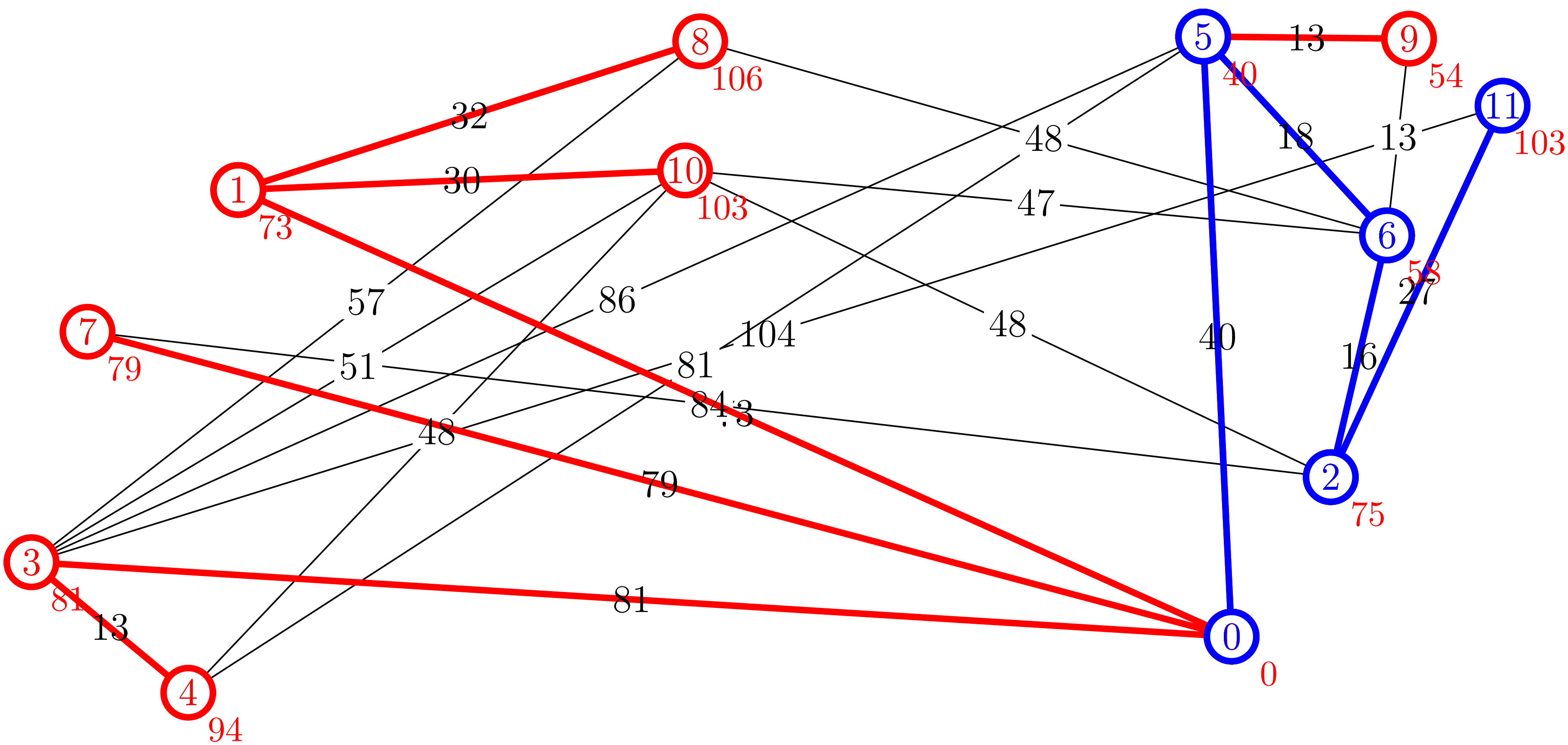
Going from Node 0 to Node 11



Going from Node 0 to Node 11



Going from Node 0 to Node 11



Uses of Dynamic Programming

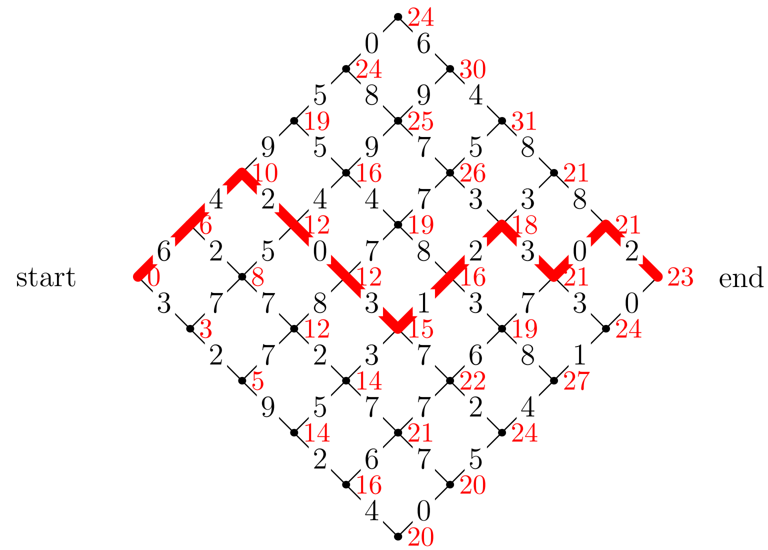
- Recurrent solutions to lattice models for protein-DNA binding
- Backward induction as a solution method for finite-horizon discrete-time dynamic optimization problems
- Method of undetermined coefficients can be used to solve the Bellman equation in infinite-horizon, discrete-time, discounted, time-invariant dynamic optimization problems
- Many string algorithms including longest common subsequence, longest increasing subsequence, longest common substring, Levenshtein distance (edit distance)
- Many algorithmic problems on graphs can be solved efficiently for graphs of bounded treewidth or bounded clique-width by using dynamic programming on a tree decomposition of the graph.
- The Cocke–Younger–Kasami (CYK) algorithm which determines whether and how a given string can be generated by a given context-free grammar
- Knuth's word wrapping algorithm that minimizes raggedness when word wrapping text

- The use of transposition tables and refutation tables in computer chess
- The Viterbi algorithm (used for hidden Markov models)
- The Earley algorithm (a type of chart parser)
- The Needleman–Wunsch and other algorithms used in bioinformatics, including sequence alignment, structural alignment, RNA structure prediction
- Floyd's all-pairs shortest path algorithm
- Optimizing the order for chain matrix multiplication
- Pseudo-polynomial time algorithms for the subset sum and knapsack and partition problems
- The dynamic time warping algorithm for computing the global distance between two time series
- The Selinger (a.k.a. System R) algorithm for relational database query optimization
- De Boor algorithm for evaluating B-spline curves
- Duckworth–Lewis method for resolving the problem when games of cricket are interrupted

- The value iteration method for solving Markov decision processes
- Some graphic image edge following selection methods such as the "magnet" selection tool in Photoshop
- Some methods for solving interval scheduling problems
- Some methods for solving word wrap problems
- Some methods for solving the travelling salesman problem, either exactly (in exponential time) or approximately (e.g. via the bitonic tour)
- Recursive least squares method
- Beat tracking in music information retrieval
- Adaptive-critic training strategy for artificial neural networks
- Stereo algorithms for solving the correspondence problem used in stereo vision
- Seam carving (content aware image resizing)
- The Bellman–Ford algorithm for finding the shortest distance in a graph
- Some approximate solution methods for the linear search problem
- Kadane's algorithm for the maximum subarray problem

Outline

1. Dynamic Programming
2. Applications
    - Line Breaks
    - Edit Distance
    - Dijkstra's Algorithm
3. **Limitation**



When You Can't Use It

- Not all problems can be split neatly to make dynamic programming possible
- Dynamic programming works on problems with some natural ordering
- We need this to build up a list of optimum cost of partial solutions—these have to depend on the cost of previous partial solutions
- Sometime no natural ordering exists

When You Can't Use It

- Not all problems can be split neatly to make dynamic programming possible
- Dynamic programming works on problems with some natural ordering
- We need this to build up a list of optimum cost of partial solutions—these have to depend on the cost of previous partial solutions
- Sometime no natural ordering exists

When You Can't Use It

- Not all problems can be split neatly to make dynamic programming possible
- Dynamic programming works on problems with some natural ordering
- We need this to build up a list of optimum cost of partial solutions—these have to depend on the cost of previous partial solutions
- Sometime no natural ordering exists

When You Can’t Use It

- Not all problems can be split neatly to make dynamic programming possible
- Dynamic programming works on problems with some natural ordering
- We need this to build up a list of optimum cost of partial solutions—these have to depend on the cost of previous partial solutions
- Sometime no natural ordering exists

Travelling Salesman Problem

- For TSP we can find a dynamic programming solution by considering optimal sub-tours (paths) involving sets of k cities

- If we know the optimal sub-tour through all sets of cities of size k (starting and finishing at each possible pair in the set) then we can quickly compute the optimal sub-tour between set of size $k+1$

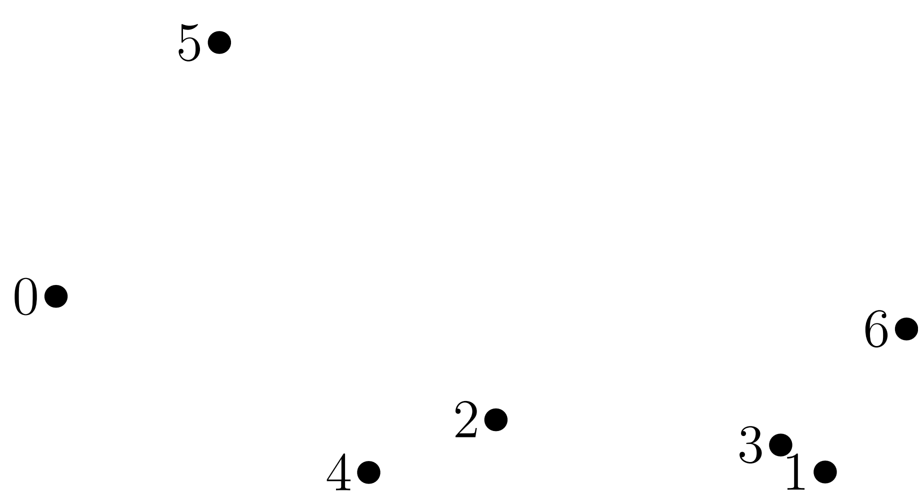
Travelling Salesman Problem

- For TSP we can find a dynamic programming solution by considering optimal sub-tours (paths) involving sets of k cities
- If we know the optimal sub-tour through all sets of cities of size k (starting and finishing at each possible pair in the set) then we can quickly compute the optimal sub-tour between set of size $k+1$

Travelling Salesman Problem

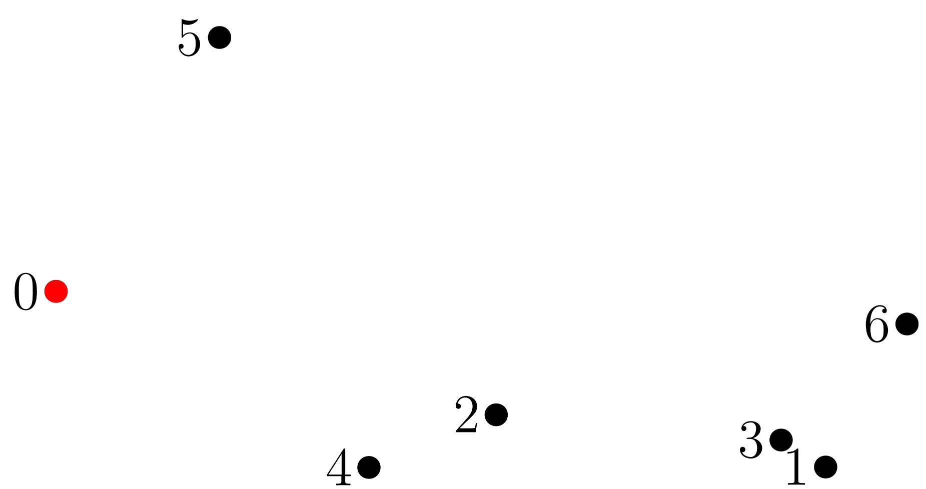
- For TSP we can find a dynamic programming solution by considering optimal sub-tours (paths) involving sets of k cities

- If we know the optimal sub-tour through all sets of cities of size k (starting and finishing at each possible pair in the set) then we can quickly compute the optimal sub-tour between set of size $k+1$



Travelling Salesman Problem

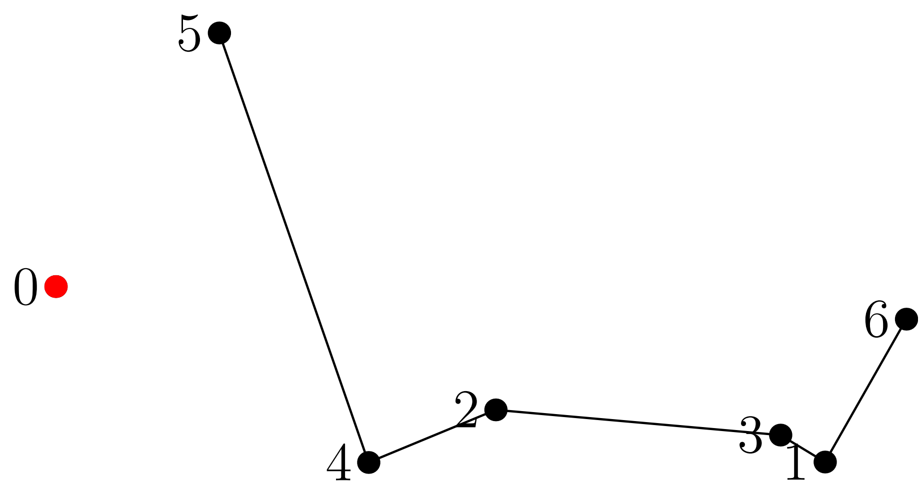
- For TSP we can find a dynamic programming solution by considering optimal sub-tours (paths) involving sets of k cities
- If we know the optimal sub-tour through all sets of cities of size k (starting and finishing at each possible pair in the set) then we can quickly compute the optimal sub-tour between set of size $k+1$



Travelling Salesman Problem

- For TSP we can find a dynamic programming solution by considering optimal sub-tours (paths) involving sets of k cities

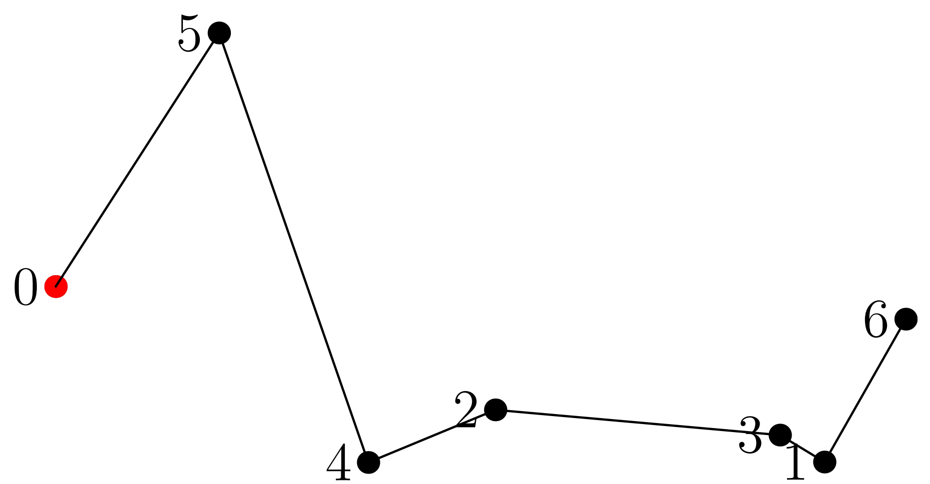
- If we know the optimal sub-tour through all sets of cities of size k (starting and finishing at each possible pair in the set) then we can quickly compute the optimal sub-tour between set of size $k+1$



Travelling Salesman Problem

- For TSP we can find a dynamic programming solution by considering optimal sub-tours (paths) involving sets of k cities

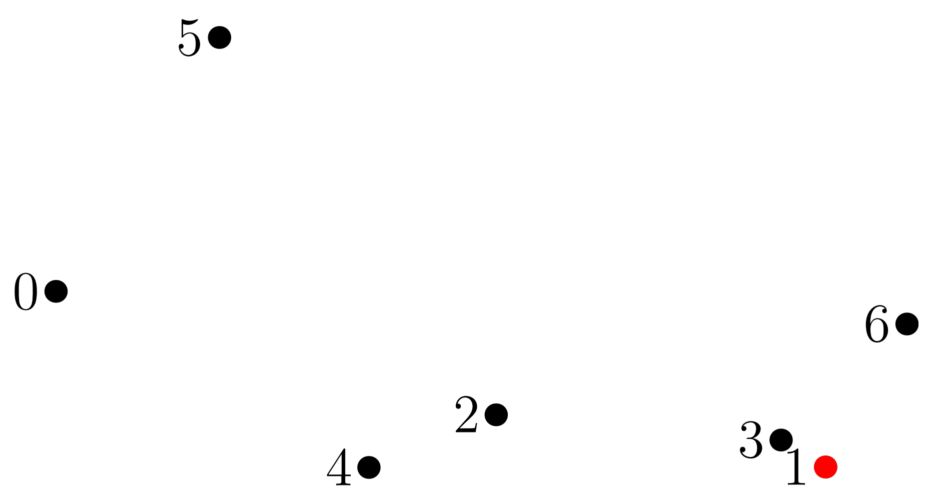
- If we know the optimal sub-tour through all sets of cities of size k (starting and finishing at each possible pair in the set) then we can quickly compute the optimal sub-tour between set of size $k+1$



Travelling Salesman Problem

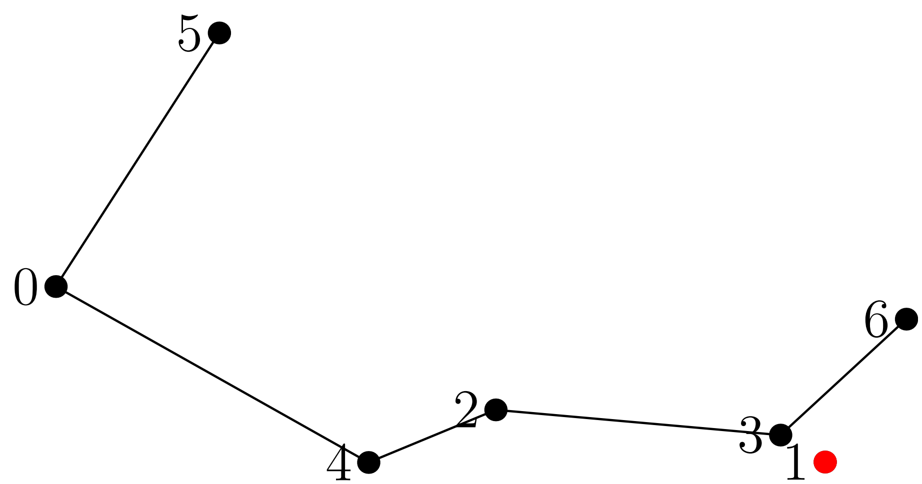
- For TSP we can find a dynamic programming solution by considering optimal sub-tours (paths) involving sets of k cities

- If we know the optimal sub-tour through all sets of cities of size k (starting and finishing at each possible pair in the set) then we can quickly compute the optimal sub-tour between set of size $k+1$



Travelling Salesman Problem

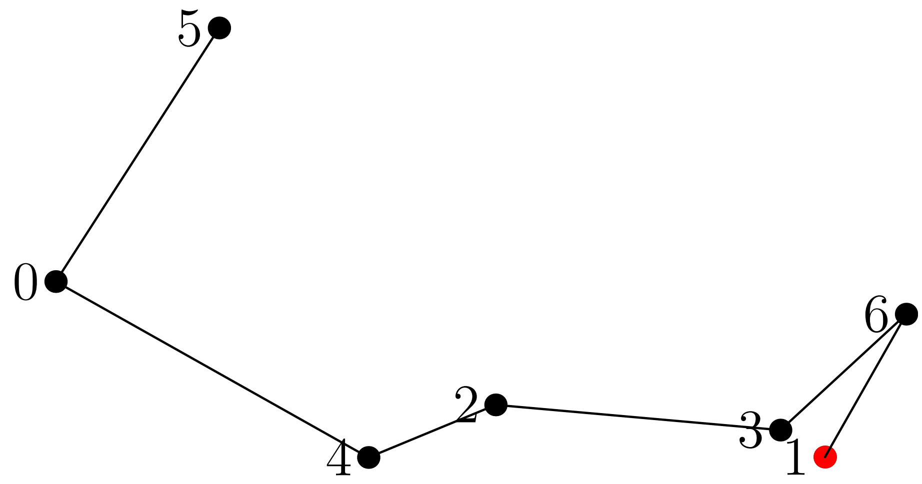
- For TSP we can find a dynamic programming solution by considering optimal sub-tours (paths) involving sets of k cities
- If we know the optimal sub-tour through all sets of cities of size k (starting and finishing at each possible pair in the set) then we can quickly compute the optimal sub-tour between set of size $k+1$



Travelling Salesman Problem

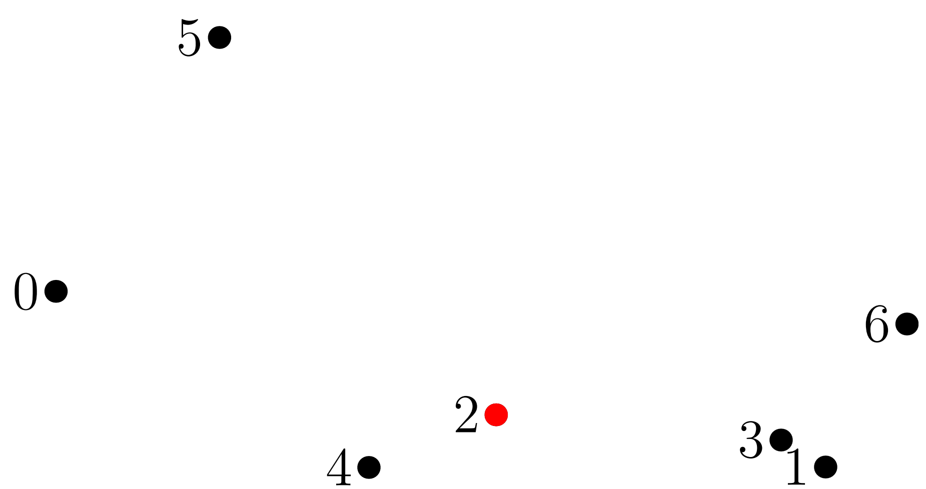
- For TSP we can find a dynamic programming solution by considering optimal sub-tours (paths) involving sets of k cities

- If we know the optimal sub-tour through all sets of cities of size k (starting and finishing at each possible pair in the set) then we can quickly compute the optimal sub-tour between set of size $k + 1$



Travelling Salesman Problem

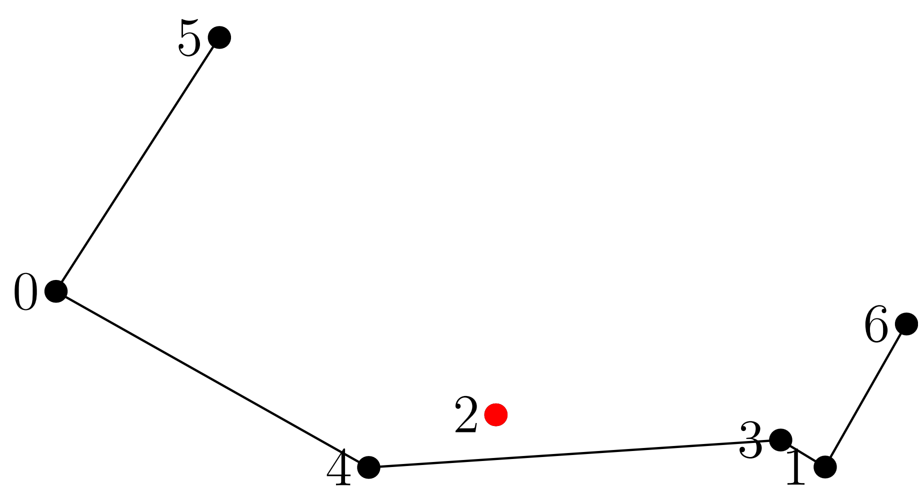
- For TSP we can find a dynamic programming solution by considering optimal sub-tours (paths) involving sets of k cities
- If we know the optimal sub-tour through all sets of cities of size k (starting and finishing at each possible pair in the set) then we can quickly compute the optimal sub-tour between set of size $k+1$



Travelling Salesman Problem

- For TSP we can find a dynamic programming solution by considering optimal sub-tours (paths) involving sets of k cities

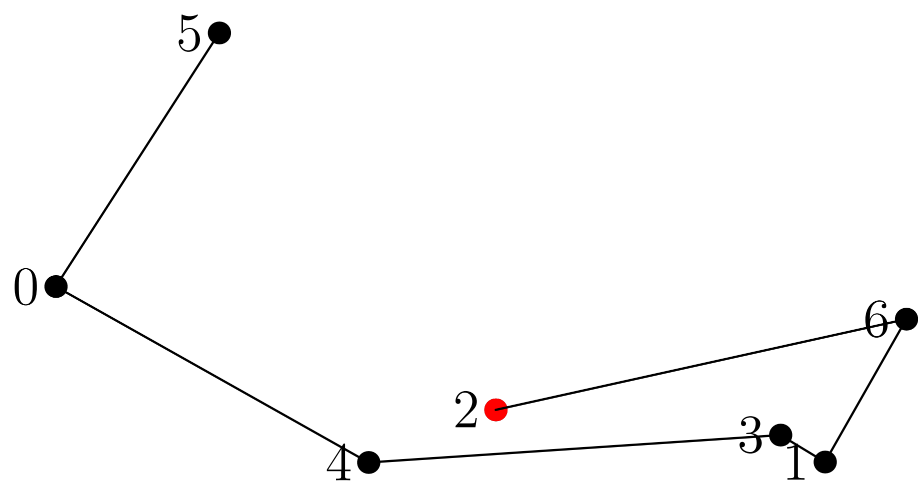
- If we know the optimal sub-tour through all sets of cities of size k (starting and finishing at each possible pair in the set) then we can quickly compute the optimal sub-tour between set of size $k+1$



Travelling Salesman Problem

- For TSP we can find a dynamic programming solution by considering optimal sub-tours (paths) involving sets of k cities

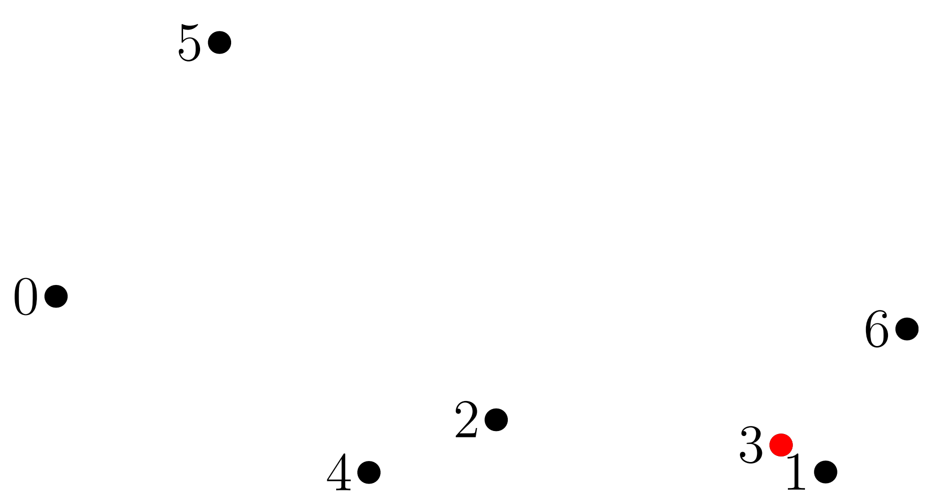
- If we know the optimal sub-tour through all sets of cities of size k (starting and finishing at each possible pair in the set) then we can quickly compute the optimal sub-tour between set of size $k+1$



Travelling Salesman Problem

- For TSP we can find a dynamic programming solution by considering optimal sub-tours (paths) involving sets of k cities

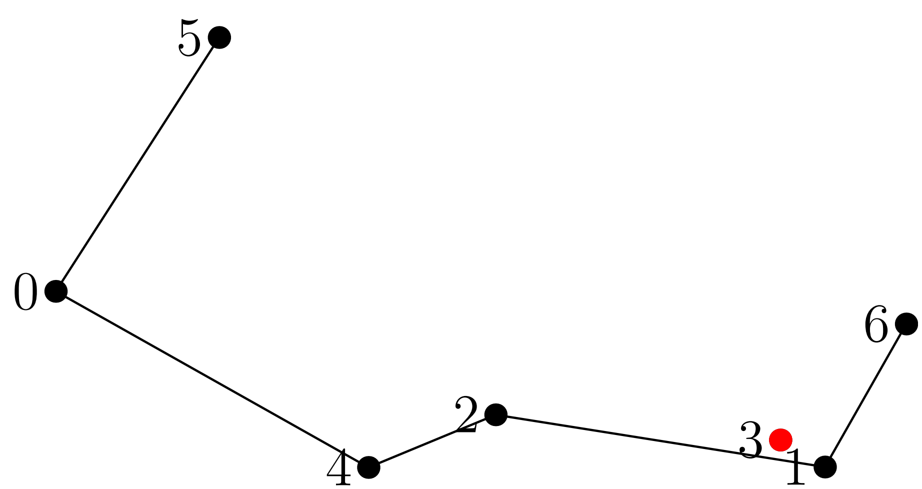
- If we know the optimal sub-tour through all sets of cities of size k (starting and finishing at each possible pair in the set) then we can quickly compute the optimal sub-tour between set of size $k+1$



Travelling Salesman Problem

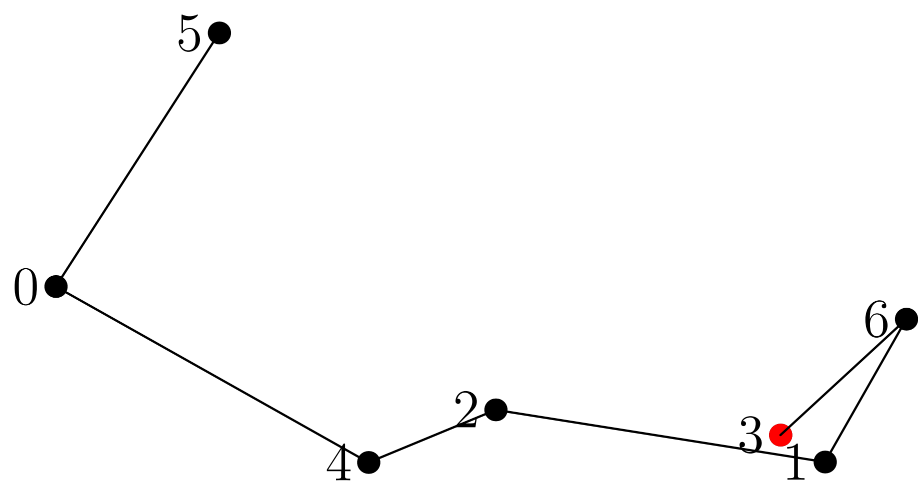
- For TSP we can find a dynamic programming solution by considering optimal sub-tours (paths) involving sets of k cities

- If we know the optimal sub-tour through all sets of cities of size k (starting and finishing at each possible pair in the set) then we can quickly compute the optimal sub-tour between set of size $k+1$



Travelling Salesman Problem

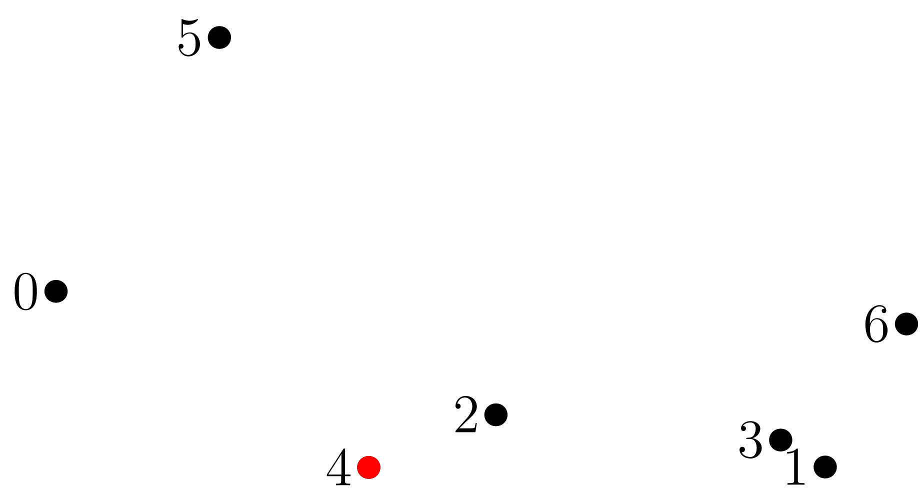
- For TSP we can find a dynamic programming solution by considering optimal sub-tours (paths) involving sets of k cities
- If we know the optimal sub-tour through all sets of cities of size k (starting and finishing at each possible pair in the set) then we can quickly compute the optimal sub-tour between set of size $k+1$



Travelling Salesman Problem

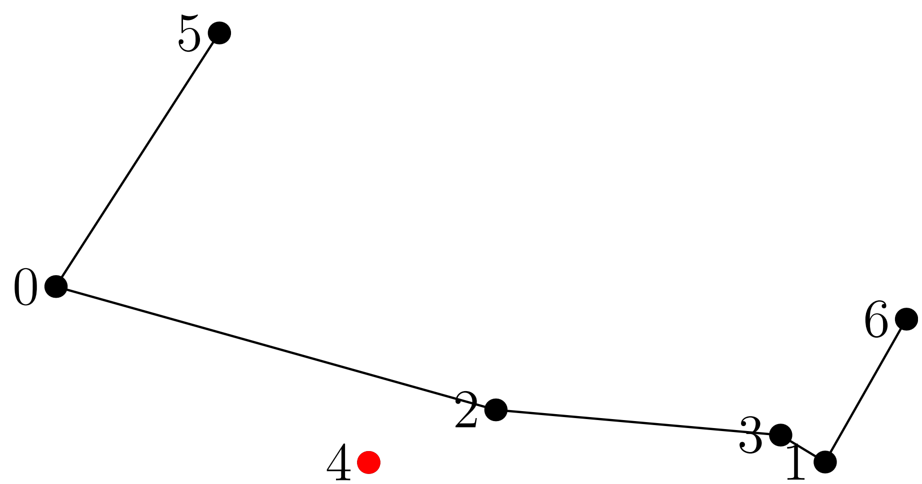
- For TSP we can find a dynamic programming solution by considering optimal sub-tours (paths) involving sets of k cities

- If we know the optimal sub-tour through all sets of cities of size k (starting and finishing at each possible pair in the set) then we can quickly compute the optimal sub-tour between set of size $k+1$



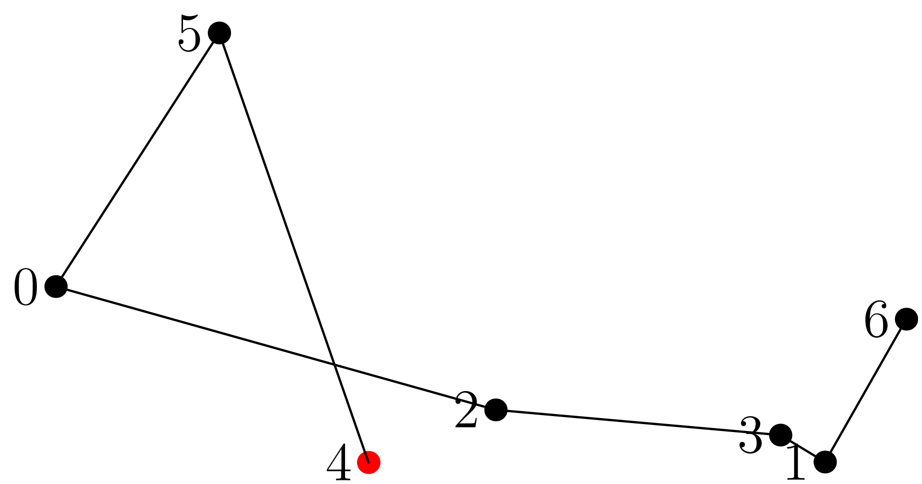
Travelling Salesman Problem

- For TSP we can find a dynamic programming solution by considering optimal sub-tours (paths) involving sets of k cities
- If we know the optimal sub-tour through all sets of cities of size k (starting and finishing at each possible pair in the set) then we can quickly compute the optimal sub-tour between set of size $k+1$



Travelling Salesman Problem

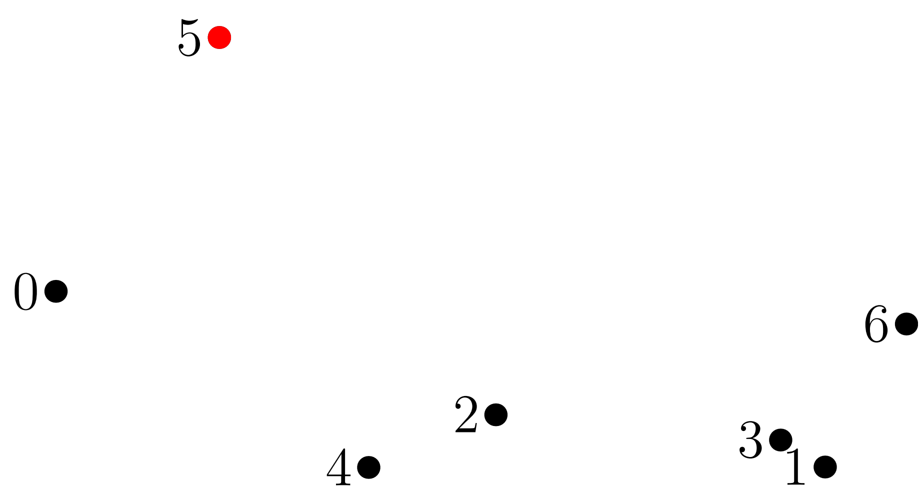
- For TSP we can find a dynamic programming solution by considering optimal sub-tours (paths) involving sets of k cities
- If we know the optimal sub-tour through all sets of cities of size k (starting and finishing at each possible pair in the set) then we can quickly compute the optimal sub-tour between set of size $k+1$



Travelling Salesman Problem

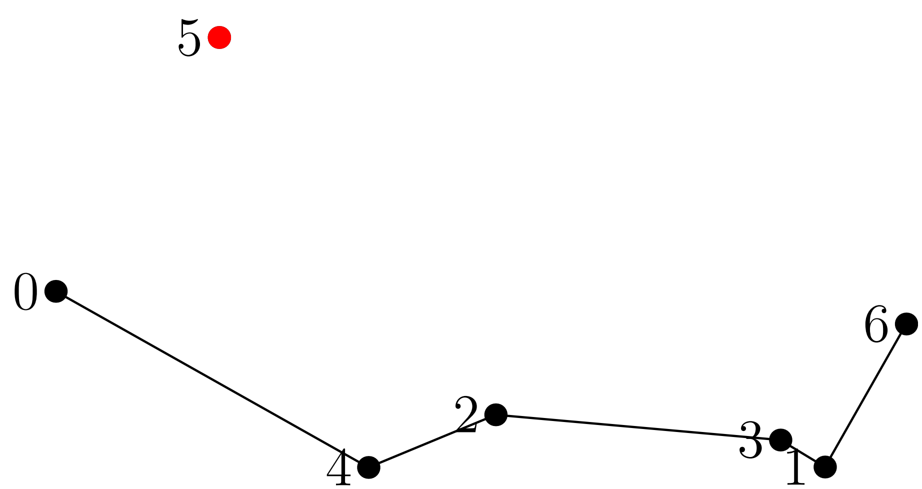
- For TSP we can find a dynamic programming solution by considering optimal sub-tours (paths) involving sets of k cities

- If we know the optimal sub-tour through all sets of cities of size k (starting and finishing at each possible pair in the set) then we can quickly compute the optimal sub-tour between set of size $k+1$



Travelling Salesman Problem

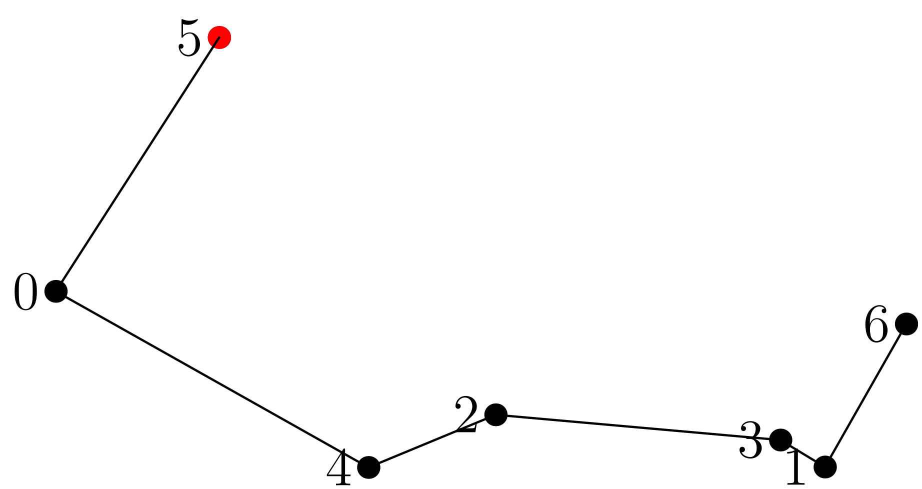
- For TSP we can find a dynamic programming solution by considering optimal sub-tours (paths) involving sets of k cities
- If we know the optimal sub-tour through all sets of cities of size k (starting and finishing at each possible pair in the set) then we can quickly compute the optimal sub-tour between set of size $k+1$



Travelling Salesman Problem

- For TSP we can find a dynamic programming solution by considering optimal sub-tours (paths) involving sets of k cities

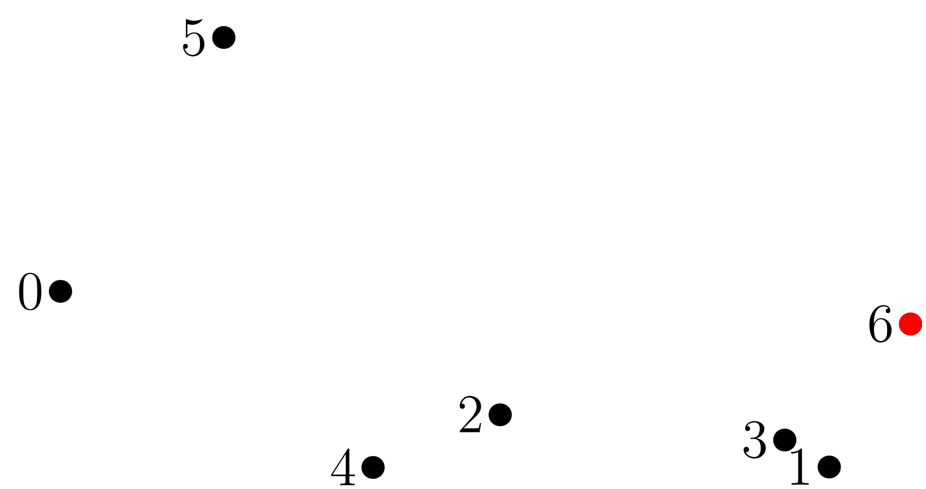
- If we know the optimal sub-tour through all sets of cities of size k (starting and finishing at each possible pair in the set) then we can quickly compute the optimal sub-tour between set of size $k+1$



Travelling Salesman Problem

- For TSP we can find a dynamic programming solution by considering optimal sub-tours (paths) involving sets of k cities

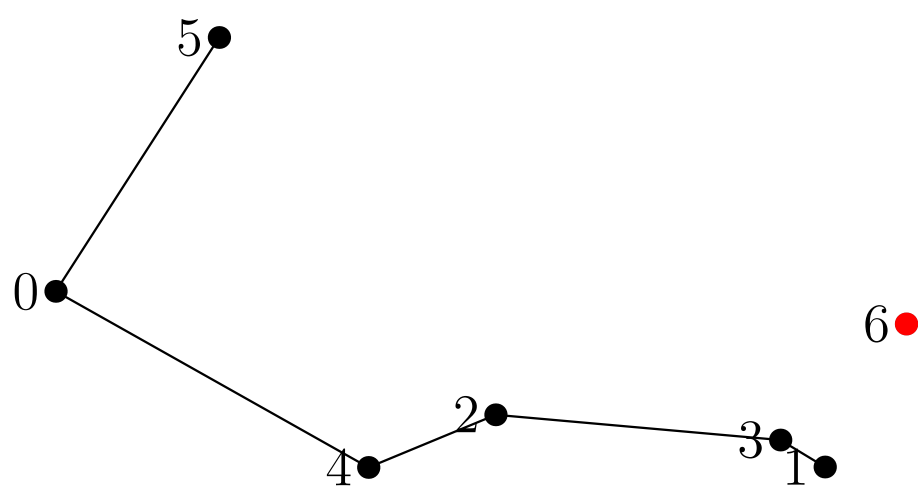
- If we know the optimal sub-tour through all sets of cities of size k (starting and finishing at each possible pair in the set) then we can quickly compute the optimal sub-tour between set of size $k+1$



Travelling Salesman Problem

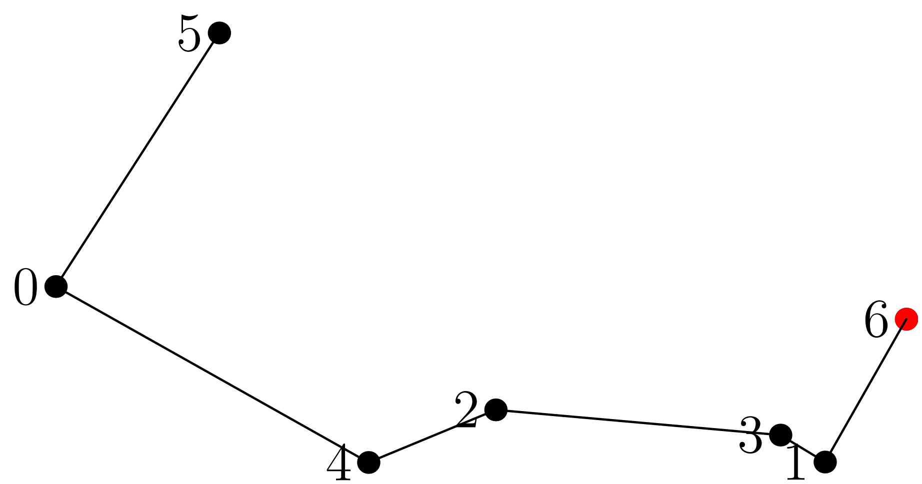
- For TSP we can find a dynamic programming solution by considering optimal sub-tours (paths) involving sets of k cities

- If we know the optimal sub-tour through all sets of cities of size k (starting and finishing at each possible pair in the set) then we can quickly compute the optimal sub-tour between set of size $k+1$



Travelling Salesman Problem

- For TSP we can find a dynamic programming solution by considering optimal sub-tours (paths) involving sets of k cities
- If we know the optimal sub-tour through all sets of cities of size k (starting and finishing at each possible pair in the set) then we can quickly compute the optimal sub-tour between set of size $k+1$



Travelling Salesman Problem

- The problem is there are $\binom{n}{k}$ subsets consisting of k cities out of a possible n
- The total number of subsets that need to be considered is

$$\sum_{k=0}^{n} \binom{n}{k} = 2^n$$

- The time complexity of the DP solution is $n^2\,2^n$ which is better than $n!$ and is currently the fastest known exact algorithm for TSP

Travelling Salesman Problem

- The problem is there are $\binom{n}{k}$ subsets consisting of k cities out of a possible n

- The total number of subsets that need to be considered is

$$\sum_{k=0}^{n} \binom{n}{k} = 2^n$$

- The time complexity of the DP solution is $n^2\,2^n$ which is better than $n!$ and is currently the fastest known exact algorithm for TSP

Travelling Salesman Problem

- The problem is there are $\binom{n}{k}$ subsets consisting of k cities out of a possible n
- The total number of subsets that need to be considered is

$$\sum_{k=0}^{n} \binom{n}{k} = 2^n$$

- The time complexity of the DP solution is $n^2 \, 2^n$ which is better than $n!$ and is currently the fastest known exact algorithm for TSP

Travelling Salesman Problem

- The problem is there are $\binom{n}{k}$ subsets consisting of k cities out of a possible n

- The total number of subsets that need to be considered is

$$\sum_{k=0}^{n} \binom{n}{k} = 2^n$$

- The time complexity of the DP solution is $n^2\,2^n$ which is better than $n!$ and is currently the fastest known exact algorithm for TSP, but it ain't very useful in practice!

Conclusions

- Dynamic programming is one of the most powerful strategies for solving hard optimisation problems
- It works by iteratively building up costs for partial solutions using the costs of smaller partial solutions
- When it works it is great and there are hosts of practical algorithms which use DP
- However, it doesn't always work

Conclusions

- Dynamic programming is one of the most powerful strategies for solving hard optimisation problems
- It works by iteratively building up costs for partial solutions using the costs of smaller partial solutions
- When it works it is great and there are hosts of practical algorithms which use DP
- However, it doesn't always work

Conclusions

- Dynamic programming is one of the most powerful strategies for solving hard optimisation problems
- It works by iteratively building up costs for partial solutions using the costs of smaller partial solutions
- When it works it is great and there are hosts of practical algorithms which use DP
- However, it doesn't always work

Conclusions

- Dynamic programming is one of the most powerful strategies for solving hard optimisation problems
- It works by iteratively building up costs for partial solutions using the costs of smaller partial solutions
- When it works it is great and there are hosts of practical algorithms which use DP
- However, it doesn't always work