Complexity (SLOa) - Homework 1

Šimon Stupinský - xstupi00@stud.fit.vutbr.cz

Obtained points:

1. TASK

Assignment: Let L be the following formal language $L = \{a^i b^i c^i \mid i \geq 0\}$.

(a) Design a Turing machine M *deciding*[1] the language L.

(b) Estimate the upper bounds for its time and space complexity.

---

a) Design a Turing machine M deciding the language L:

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ be a such TS that $L(M) = L$, where:

$$Q = \{q_A, q_R, q_0, q_1, q_2, q_3, q_4, q_5\},\ \Sigma = \{a, b, c\},\ \Gamma = \{a, b, c, X, Y, Z, \Delta\},\ F = \{q_A, q_R\},$$

and transition function δ is defined by the table as follows:

δ	Δ	a	b	c	X	Y	Z
q_0	q_1, R						
q_1	q_A	q_2, X	q_R	q_R	q_5, R		
q_2	q_R	q_2, R	q_3, X	q_R	q_2, R	q_2, R	q_R
q_3	q_R	q_R	q_3, R	q_4, X		q_3, R	q_3, R
q_4		q_4, L	q_4, L		q_1, R	q_4, L	q_4, L
q_5	q_A	q_R	q_R	q_R		q_5, R	q_5, R

The machine gradually scans across the tape and crossing off a single character from the all possible symbols - a, b, c. In the state q_1 replaces the symbol a by the symbol X, then in the state q_2 finds the symbol b, which replaces by the symbol Y, and then in the state q_3 finds and replaces the symbol c by Z. Subsequently, in the state q_4 returns to the beginning of the input word, respectively to first occurrences of the symbol X, where it starts crossing out another triplet. Accepts, only when the input word is empty or when he managed to cross the whole input word by triplets of the symbols. Rejects, when the order of symbols in the word was violated or when the count of the individual symbols are not equal. Figure 1 view the diagram of this Turing machine M.

b) Estimate the upper bounds for its time and space complexity:

As was said above, the processing of the input word by TMM, that decides the language L, we can divide into the following stages:

1. Repeat as long as all three symbols (a, b, c) remain on the tape:
    2. Scan across the tape, the crossing of a single symbol **a**, the single symbol **b** and the single symbol **c**.
3. If some symbols (**a**, **b**, **c**) still remains unmarked on the tape, then M rejects. Otherwise, accepts.

We will analyse the **time** complexity of each these three stages separately. Because each scan crosses of three symbols (**a**, **b**, **c**), at most $\frac{\mathbf{n}}{\mathbf{3}}$ scans occur. Therefore the maximum number of the repeating of the **second** stage is $\frac{\mathbf{n}}{\mathbf{3}}$. The second stage itself includes the overwriting of three symbols, move to the right across the tape from the last occurrence of the symbol **X**, respectively in the first scanning from the beginning of the tape (Δ), and finally move back to the first occurrence of the symbol **X**.

[1] By deciding we understand that the Turing machine provides an answer YES or NO for each input.

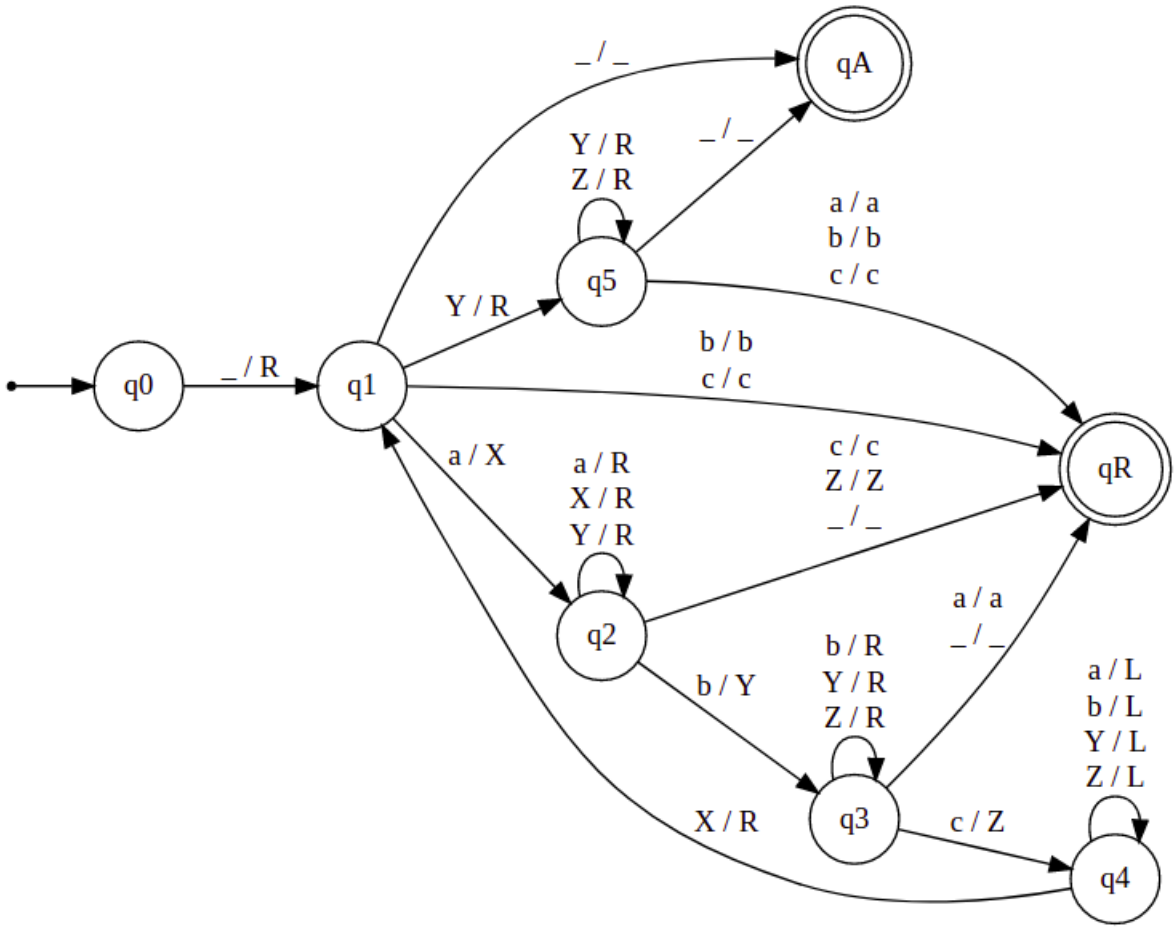

Figure 1. The diagram of the Turing machine M. The symbol _ in the diagram represents the symbol Δ from the transition function δ. The marking of transitions v/w represents the following meaning: $v \in \Gamma$ represents the current symbol under the machine head and $y \in \Gamma \cup \{L, R\}$ represents the shift of the machine head (to the right - R or left - L) or overwriting the symbol under the machine head.

It is clear, that the overwriting of three symbols takes totally **3** operations. The move from the last (from the beginning of the tape) occurrence of the symbol **X** to the first occurrence of the symbol **c** takes maximally $\mathbf{n} - (\frac{\mathbf{n}}{\mathbf{3}} - \mathbf{1})$ transitions. A subsequent move back to the first occurrence of the symbol **X** (from the end of the tape) takes totally $\frac{\mathbf{2}}{\mathbf{3}}\mathbf{n}$. In summary, the time taken by one repeating of the whole second stage is equal to:

$$3 + n - (\frac{n}{3} - 1) + \frac{2}{3}n = 3 + n - \frac{n}{3} + 1 + \frac{2}{3}n = 4 + \frac{2}{3}n + \frac{2}{3}n = \frac{\mathbf{4}}{\mathbf{3}}\mathbf{n} + \mathbf{4}.$$

Therefore, each scan of the tape in stages 1 and 2 is performed in $\mathcal{O}(\mathbf{n})$. The total time taken by these stages is $\frac{\mathbf{n}}{\mathbf{3}}\mathcal{O}(\mathbf{n}) = \mathcal{O}(\mathbf{n^2})$. In stage 3 the machine makes a single scan from the last occurrence of the symbol **X** (from the beginning of the tape) to decide whether to accept or reject, taking by $\frac{\mathbf{2}}{\mathbf{3}}\mathbf{n} + \mathbf{2}$ transitions maximally, so in $\mathcal{O}(\mathbf{n})$ time. The total **time** of Turing machine M on an input of length **n** is: $\mathcal{O}(n^2) + \mathcal{O}(n) = \mathcal{O}(\mathbf{n^2})$.

Space (memory) complexity of Turing machine is given by the number of cells required for a given computation. Note, that the content of the tape at the beginning of the computation is in the form $\underline{\Delta} w \Delta^{\omega}$, where $w \in \Sigma^*$ represents the input word. When the Turing machine M accepts the input w, then the number of used cells will be equal $\mathbf{n} + \mathbf{2}$, where $|w| = n$. Since, the machine M does not extend the content of the tape, only rewrites, it never uses more cells than the length of the input word plus two blank symbols at the beginning and end of the tape. Therefore, the **space** complexity of the *TM* M on an input of length **n** is $\mathcal{O}(\mathbf{n})$.

2. TASK

Assignment: Design a RAM program, which for the input vector $I = (n)$ computes a 3^{rd} power of the number n (let us suppose that $n \geq 0$). After the `HALT` instruction, the register r_0 will contain a number a such that $a = n^3$. (Note: It is not necessary to implement optimal algorithm.)

(a) Analyse uniform time and space complexity and estimate upper bounds.

(b) Analyse logarithmic time and space complexity and estimate upper bounds.

Do not forget that time and space complexity of RAM program is estimated w.r.t the size of its input (i.e. the number of bits of the input number n).

Design a RAM program, which for the input vector $I = (n)$ computes a 3^{rd} power of the number n:
To generalise the assignment we designed a RAM program, which for the input vector $I = (x,n)$ computes n^{th} power of the number x.

1.	`READ 2`	$(R_0 \leftarrow I_2)$	19.	`STORE 5`	$(R_5 \leftarrow R_0)$
2.	`SUB ↑1`	$(R_0 \leftarrow R_0 - 1)$	10.	`LOAD 3`	$(R_0 \leftarrow R_3)$
3.	`STORE 6`	$(R_6 \leftarrow R_0)$	21.	`JZERO 31`	(if $R_0 = 0$ then $\kappa \leftarrow 31$)
4.	`READ 1`	$(R_0 \leftarrow I_1)$	22.	`JUMP 8`	$(\kappa \leftarrow 8)$
5.	`STORE 1`	$(R_1 \leftarrow R_0)$	23.	`LOAD 4`	$(R_0 \leftarrow R_4)$
6.	`STORE 5`	$(R_5 \leftarrow R_0)$	24.	`STORE 1`	$(R_1 \leftarrow R_0)$
7.	`STORE 7`	$(R_7 \leftarrow R_0)$	25.	`STORE 5`	$(R_5 \leftarrow R_0)$
8.	`STORE 2`	$(R_2 \leftarrow R_0)$	26.	`LOAD ↑0`	$(R_0 \leftarrow 0)$
9.	`HALF`	$(R_0 \leftarrow R_0/2)$	27.	`STORE 4`	$(R_4 \leftarrow R_0)$
10.	`STORE 3`	$(R_3 \leftarrow R_0)$	28.	`STORE 3`	$(R_3 \leftarrow R_0)$
11.	`ADD 3`	$(R_0 \leftarrow R_0 + R_3)$	29.	`LOAD 7`	$(R_0 \leftarrow R_7)$
12.	`SUB 2`	$(R_0 \leftarrow R_0 - R_2)$	30.	`JUMP 8`	$(\kappa \leftarrow 8)$
13.	`JZERO 17`	(if $R_0 = 0$ then $\kappa \leftarrow 17$)	31.	`LOAD 6`	$(R_0 \leftarrow R_6)$
14.	`LOAD 4`	$(R_0 \leftarrow R_4)$	32.	`SUB ↑1`	$(R_0 \leftarrow R_0 - 1)$
15.	`ADD 5`	$(R_0 \leftarrow R_0 + R_5)$	33.	`STORE 6`	$(R_6 \leftarrow R_0)$
16.	`STORE 4`	$(R_4 \leftarrow R_0)$	34.	`JPOS 23`	(if $R_0 > 0$ then $\kappa \leftarrow 23$)
17.	`LOAD 5`	$(R_0 \leftarrow R_5)$	35.	`LOAD 4`	$(R_0 \leftarrow R_4)$
18.	`ADD 5`	$(R_0 \leftarrow R_0 + R_5)$	36.	`HALT`	$(\kappa \leftarrow 0)$

This RAM program computes the function $\phi : N \times N \longmapsto N$, where $\phi(i_1, i_2) = i_1^{i_2}$. It uses the ordinary binary multiplication of two integers $(i_1 * i_1)$, which is repeating according to the chosen power number i_2. The essence lies in the use of the instruction `HALF`, which recovers the binary representation of the current number. The lines `1. – 7.` executes the loading and storing of both operands i_1 and i_2 to all required registers.

Firstly, we will nearly describe the processing within lines `8. – 22.`, which represents the multiplication of two given numbers loaded in registers R_1 and R_2. At the beginning of the k^{th} iteration register R_3 contains $\left\lfloor i_2 \,/\, 2^k \right\rfloor$, register R_5 contains the value $2^k * r_1$, and register R_4 contains the value $r_1 * (i_2 \mod 2^k)$. At the end of each iteration is compared the register R_3 with 0. When it contains the value 0, then the final or partial result is available in the register R_4. Note, that we start with the value $k = 0$ and it is incrementing at the line number `9`. We use the notation r_1 for the content of register R_1, because this value is changing during the computation, and it is not equal to i_1 just.

The next section of the computation, the lines `23. – 30.`, represents the steps between the different iteration of multiplication itself. This phase will be first reached after the first multiplication and so the result is available in the register R_4. This result will represent the first operand (r_1) in the next iteration of multiplication and therefore it is loading to the registers R_1 and R_5 for next run. The registers R_3 and R_4 contain the control variables of each

multiplication iteration and therefore they must be reset before each new iteration. The first step of each multiplication iteration is the loading of the operand i_2 from the register R_O to register R_2. Therefore, the last step in this phase is the loading of stored operand i_2 from the register R_7 to R_0.

Complexity Analysis

To analyse the complexity of the designed RAM program we list the following abstract algorithm:

1. **instructions** : 1. − 7. $\Big\}\,\mathbf{1}\times$
2. **for** $i \in <1, i_2-1>$: $\Bigg\}\,(\mathbf{i_2}-\mathbf{1})\times$
3. **for** $k \in <1, \lceil log\, i_1 \rceil>$: $\Big\}\,\lceil \mathbf{log\, i_1} \rceil \times$
4. **instructions** : 8. − 22. $\Big\}$
5. **if** $i+1 \leq i_2-1$:
6. **instructions** : 23. − 30.
7. **instructions** : 31. − 34. $\Bigg\}$
8. **instructions** : 35. − 36. $\Big\}\,\mathbf{1}\times$

Notice that due to condition on the line **5** will instructions **23 − 30** executed exactly $(\mathbf{i_2}-\mathbf{2})$ times because in the last $(i_2-1)^{th}$ iteration does not need to perform the re-initialisation of the registers for next iteration. Firstly, we will analyse the inner loop, so the lines **3** and **4**. The number of iterations of this loop will be at most equal to $\lceil \mathbf{log\, i_1} \rceil$ and each such iteration entails the execution of a constant number of instructions. Denote **n** as the length of the binary representation of the integer $\mathbf{i_1}$ and then we are valid, that the maximal number of iterations will not greater than this length: $\lceil \mathbf{log\, i_1} \rceil \leq n$. Therefore, we can conclude, that this loop computes the product of two given numbers in $\mathcal{O}(\mathbf{n})$ time. Just to remind, by $\mathcal{O}(\mathbf{n})$ time we mean a total number of instructions that is proportional to the logarithm of the integer i_1 in the input vector.

In contrary, the outer loop is performed dependent on the value of the second integer $\mathbf{i_2}$ in the input vector. Individual repetitions represent successively multiplication so that after the first iteration we get the square of the number $\mathbf{i_1}$, in the next iteration we get the third of the given number $\mathbf{i_1}$, etc. As we can see in the above algorithm, this loop is repeated $(\mathbf{i_2}-\mathbf{1})$ times. The number of iterations of this program is at most $(\mathbf{i_2}-\mathbf{1})\cdot\lceil \mathbf{log\, i_1} \rceil$ times. In the case when we stoked to the original assignment and the value of integer $\mathbf{i_2}$ was fixed to value 3, then the number of iterations would be at most $\mathbf{2}\cdot\lceil \mathbf{log\, i_1} \rceil$ times. In this case, the entire program would be executed at most in $\mathcal{O}(\mathbf{n^2})$ time, where **n** represents the length of the binary representation of the integer $\mathbf{i_1}$. In designed RAM program, where we generalised the original assignment and allow to enter any value for integer $\mathbf{i_2}$, then the program would be executed at most in $\mathcal{O}(\mathbf{2^n})$ time. Now, the **n** represents the sum of the lengths of binary representations of both integers from input vector: $\mathbf{n} = \mathbf{bin\_len}(\mathbf{i_1}) + \mathbf{bin\_len}(\mathbf{i_2})$.

a) Analyse uniform time and space complexity and estimate upper bounds

The **uniform space complexity** of the computation of a specific RAM program on the input vector $I = (i_1, \ldots, i_n)$ is the length of this input plus the number of used registers. As was mentioned above, due to the generalisation of designed RAM program, the input vector contains two integers $I = (i_1, i_2)$. As we can see in the designed RAM program, during the computation are used the registers $R_0, \ldots, R_7$. Based on this, we can conclude the **uniform space** complexity of this RAM program, which is equal to: $|I| + |\{R_0, \ldots, R_7\}| = \mathbf{10} \Rightarrow \mathcal{O}(\mathbf{1})$.

The **uniform time complexity** of the computation of a specific RAM program on the input $I \in D$ is the number of steps, after which this RAM program halts on given input vector I. This analysis will be based on the above-listed analysis, which described the individual parts of the RAM program and indicates their number of repetitions. Recall, that input vector for designed RAM program contains two integers: $I = (i_1, i_2)$. The overall number of instructions, which are executed during the run of designed RAM program is the following:

$$\begin{aligned} 1\cdot 7 \;+\; (i_2-1)\cdot 4 \;+\; (i_2-2)\cdot 8 \;+\; (i_2-1)\cdot(\lceil log\, i_1 \rceil)\cdot 15 + 1\cdot 2 = \\ = 15\cdot i_2\cdot\lceil log\, i_1 \rceil - 15\cdot\lceil log\, i_1 \rceil + 12\cdot i_2 - 11 \;\leq\; 2^n \end{aligned}$$

As we showed, designed RAM program that computes $\mathbf{3^{rd}}$ power of the given number **x** has uniform time complexity at most in $\mathcal{O}(\mathbf{n^2})$ time ($\mathbf{DTIME}(\mathbf{n^2})$) and the general program that computes $\mathbf{y^{th}}$ power of number **x** at most in $\mathcal{O}(\mathbf{2^n})$ time ($\mathbf{DTIME}(\mathbf{2^n})$).

b) Analyse logarithmic time and space complexity and estimate upper bounds

In this logarithmic way of analyses, the algorithm is taken into account the size of the operands. The cost of operation rises logarithmically with the size of the operand because the analysis is close to the behaviour of real-world programs. The important role of this analysis has the operand, which would be during the computation storing in the register and its size will be the largest in compare to the remaining operands. In our designed RAM program the register R_5 contains the largest value during the computation. During the individual $\mathbf{i^{th}}$ iteration it contains $\mathbf{i_1 \cdot 2^i}$. How can be seen on the designed pseudo algorithm, the inner loop iterate from $\mathbf{1}$ to $\lceil \mathbf{\log i_1} \rceil$ and therefore the maximal value of exponent can be $\lceil \mathbf{\log i_1} \rceil$. The outer loop gradually changes the value of the first operand $\mathbf{i_1}$ and in its last iteration contains the value equal to $\mathbf{i_1^{i_2-1}}$.

Now, we can conclude the maximal value, which can be stored in the register R_5 during the computation: $\mathbf{2^{\lceil \log i_1 \rceil} \cdot i_1^{i_2-1}} = i_1^{i_2}$. The maximal length of product from two number a and b is the sum of their lengths of binary representations: $\mathbf{bin\_len(a) + bin\_len(b)}$. Therefore, the maximal length of the result from computing $\mathbf{i_1^{i_2}}$ is $\mathbf{i_2 \cdot bin\_len(i_1)}$. However, since the loop runs to the nearest larger squared of two from the number $\mathbf{i_1}$, we must add one more possible bit to this length. However, this does not change the fact that the maximum possible length $\mathbf{i_2 \cdot bin\_len(i_1) + 1}$ belongs to $\mathcal{O}(\mathbf{n})$.

Denote $\mathbf{n}$ as the length of the binary representation of the number $\mathbf{i_1}$. Based on analysis listed in the section about uniform space complexity, we can conclude **logarithmic space** complexity of this RAM program, which is equal to: $\mathbf{10 \cdot n} = \mathcal{O}(\mathbf{n}) \in \mathbf{DSPACE(n)}$. In the same way we can deduce the **logarithmic time** complexity of the designed RAM program, that computes $\mathbf{3^{rd}}$ power of the given number $\mathbf{i_1}$: $\mathcal{O}(\mathbf{n^2}) * \mathcal{O}(\mathbf{n}) = \mathcal{O}(\mathbf{n^3}) \in \mathbf{DTIME(n^3)}$. In the case of the generalised version of this RAM program, that computes $\mathbf{i_1^{i_2}}$, stay the **logarithmic time** complexity without a change in comparison to **uniform time** complexity because $\mathcal{O}(\mathbf{n})$ does not affect the $\mathcal{O}(\mathbf{2^n})$ and still will belong to the $\mathbf{DTIME(2^n)}$.

3. TASK

Assignment: Analyse time and space complexity of the following Dijkstra algorithm for computing shortest paths from a node s in a graph $G = (E, V)$. Provide the resulting upper bounds in terms of the number of nodes in the input graph. The size of the input is not counted into the space complexity. Do not forget on the complexity of the set implementation (lines 6 and 8).

```
1 function Dijkstra(E, V, s):
2   for each vertex v in V:
3     d[v] := infinity                 // Length of the path from s to v
4     p[v] := undefined                // Predecessor in the shortest path
5   d[s] := 0
6   N := V                            // A set of not-yet visited nodes
7   while N is not empty:
8     u := extract_min(N)              // Pick u from V with minimal v[u]
9     for each neighbor v of u:
10      alt = d[u] + 1
11      if alt < d[v]
12        d[v] := alt
13        p[v] := u
```

Analysis

The complexity bound of this algorithm is mainly dependent on the data structure used to represent the set *N* (see line 6), respectively on the implementation of `extract_min` function (see line 8). However, let us analyse the whole function `Dijkstra` from up to down. The first `for` loop on the line `1` is called once for each vertex in the graph. It executes the initialisation of the path length from `s` to current vertex `v` and set the predecessor of this vertex. Therefore, the complexity bound of this loop is $\mathbf{2} \cdot |\mathbf{V}|$, so using *big-O notation* it uses only $\mathcal{O}(|\mathbf{V}|)$ time.

Line 6 contains the building of relevant data structure of not-yet visited nodes, so it adds every vertex in the graph to this structure. The complexity of this operation depends on the specific data structure and we will discuss it below. Each vertex is processed exactly once in the next `for` loop (line 9) so function `empty` and `extract_min` are called also exactly once, e.g. $|\mathbf{V}|$ times in total. The inner loop `for each neighbour v of u` is called once for each edge in the graph and each call does $\mathcal{O}(\mathbf{1})$ work. When we want to deduce the general complexity of this algorithm, regardless of the selected data structure to implement the set **N**, we can it define as: $\mathcal{O}(|\mathbf{E}| + |\mathbf{V}| \cdot \mathbf{T_{em}})$, where T_{em} represents the complexity of function `extract_min`. Next, we will consider different data structures to implement the set **N** and look at their effect on the complexity of this algorithm.

Linear Array. The simplest version of this algorithm stores the vertex set **N** as an ordinary linear array. It is clear, that all of the linear array operations require $\mathcal{O}(|\mathbf{N}|) = \mathcal{O}(\mathbf{n})$ time, therefore, building the set **N** (line 6) takes $\mathcal{O}(|\mathbf{V}|)$ time since we add every vertex in the graph. As already we indicated, the function `extract_min` takes $\mathcal{O}(|\mathbf{V}|)$ time and there are $|\mathbf{V}|$ operations, and therefore its total time in `while` loop is $\mathcal{O}(|\mathbf{V}|^{\mathbf{2}})$. Recalls that inner `for` loop iterates $|\mathbf{E}|$ times and each iteration takes $\mathcal{O}(\mathbf{1})$ time. Hence, the running time of the algorithm with linear array implementation of the set is $\mathcal{O}(|\mathbf{V}|^{\mathbf{2}} + |\mathbf{E}|) = \mathcal{O}(|\mathbf{V}|^{\mathbf{2}})$.

Advanced Data Structures. This algorithm can be implemented with more efficient complexity when the graph will be stored in the form of *adjacency list* and using some advanced data structures such as *binary search tree* or *heap*, *priority queue* or more advanced data structure called a *Fibonacci heap*. For instance, if we consider the priority queue, which has to add or remove the entry in $\mathcal{O}(\mathbf{log\,n})$ time, then the implementation of the function `extract_min` will be more effective. We conclude that the total running time with the priority queue is equal to $\mathcal{O}(|\mathbf{E}| + |\mathbf{V}|\mathbf{log}|\mathbf{V}|) = \mathcal{O}(|\mathbf{V}|\mathbf{log}|\mathbf{V}|)$.