Camilla Bocciolone
Alba Ballarà

Programming assignment 1

Problem 1

Implementation in the file ex1.cpp

Problem 2

Implementation in the file ex2.cpp

Problem 3

Class implementation in the file ex3.cpp

Problem 4

Experiment setup: the files used to calculate the time and to draw the graphs are graphs.ipynb, ex4.cpp, ex4numberunion.cpp, ex4root.cpp, ex4rootnumberunion2.cpp and ex4_uf.cpp

In this experiment our purpose is to see how the unionfind and quickunionfind behave when changing the number unions and the sizes of the input (N). The class unionfind and quickunionfind both contains two methods connected and unions but we limited our analysis to union since

- connected in the unionfind consists in a single operation of comparison that is constant and
- connected in the quickunionfind calls the root method two times and also union in the quickunionfind calls the root method two times so their behavior will be similar.

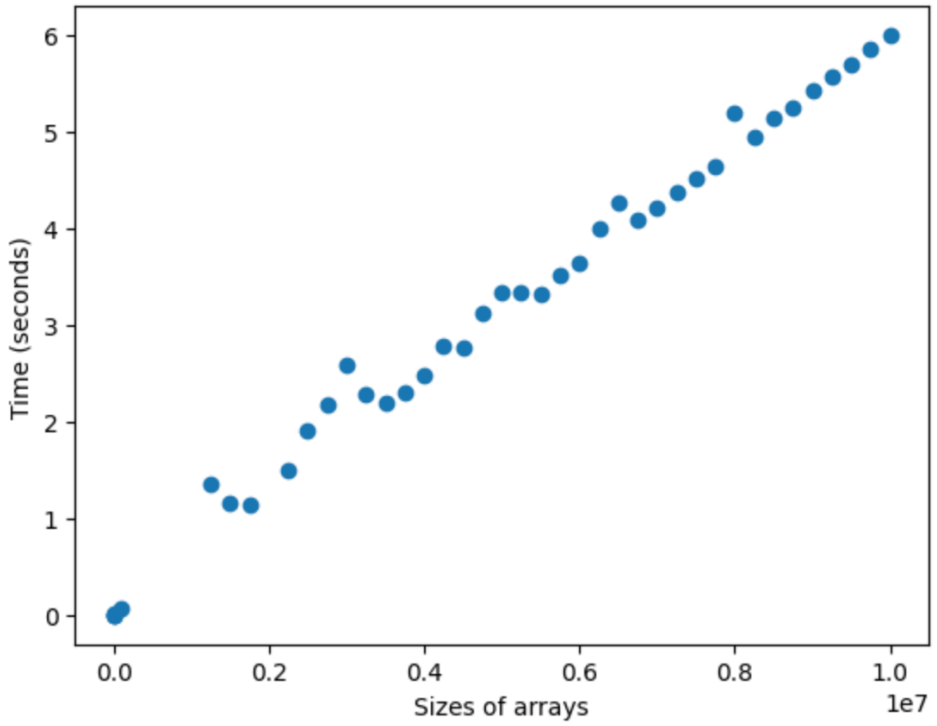
So first of all we started by changing the vector sizes in union find and we got the following results:

Changing the vector sizes with Union Find:

vector sizes	Time in seconds
10	3.96e-05
100	0.000184326

10000	0.00940735
100000	0.0710513
2250000	1.49526
2500000	1.90325
2750000	2.18313
3000000	2.59357
4000000	2.47502
5000000	3.33573
6000000	3.63908
7000000	4.22034
8000000	5.18884
9000000	5.42813
10000000	6.00078

We expect the time grows linearly when we increase the size of the vector because the union method iterates on the entire vector.



If we plot using python our times we can see that, as we expected, the time grows linearly when we increase the size of the vector. Also by doing a analysis we can see that if calculate the slope we get:

$\frac{5.42813-4.22034}{9000000-7000000}$ = 6.0390e-07~6e-07
$\frac{5.42813-3.63908}{9000000-6000000}$= 5.9635e-07~6e-07

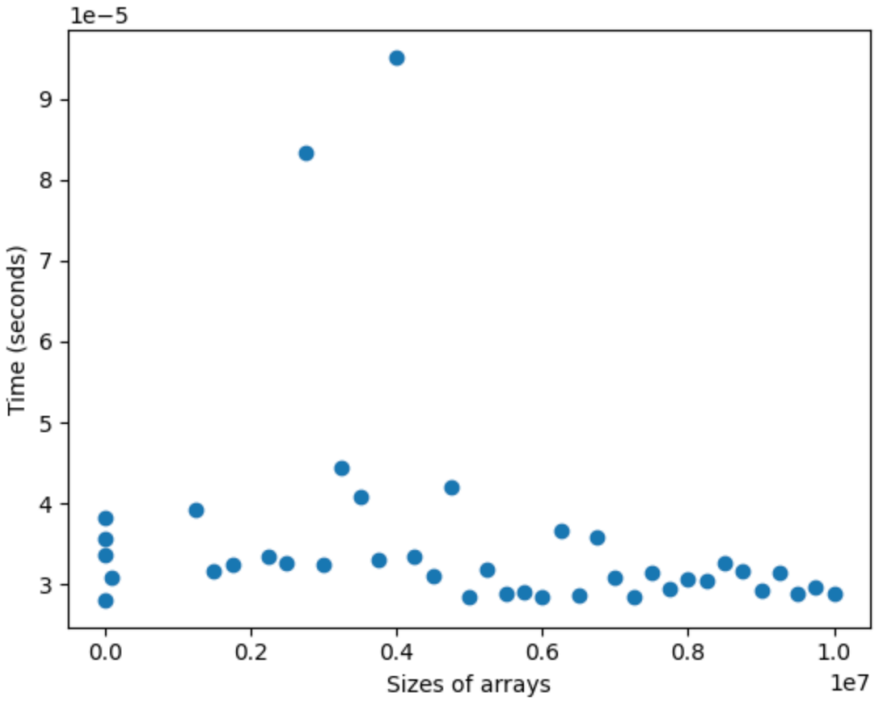
so the growth is linear $y = mx + b$ where
$m = \frac{5.42813-4.22034}{9000000-7000000} = 6e\text{-}07$
and the intersection is
$b = -(6e-07 * 225000) + 1.49526 \sim 0$

Changing the array sizes with Quick Union Find:
if we change the array sizes in the quick union find instead we get

Array sizes	Time in seconds
10	2.6037e-05
100	2.749e-05
10000	3.8013e-05

100000	3.9266e-05
2250000	3.3655e-05
2500000	4.0003e-05
2750000	3.1577e-05
3000000	3.6489e-05
4000000	3.3018e-05
5000000	3.2622e-05
6000000	2.878e-05
7000000	2.9409e-05
8000000	2.943e-05
9000000	3.8792e-05
10000000	3.8264e-05

Using the version Quick Union Find, we don’t expect it to grow when increasing the size of the vector because we don’t go through the vector every time we call unify, which means that changing the vector size is not going to affect the time it takes to unify two elements and we expect to see no growth in the graph.



As we expected, each iteration takes roughly the same time and we don't see any growth. By doing the maths analysis we get

$\frac{3.3655e{-}05 - 3.9266e{-}05}{2250000-100000}$ = -2.6098e-12

$\frac{2.9409e{-}05 - 2.878e{-}05}{7000000-6000000}$ = 6.2900e-13

so since the two numbers are really little (order of magnitude e-12) compared to the duration of the 100 unions in the code (order of magnitude e-5) we can say that unify is constant when changing the size of the vector. So $y = b =$ 3e-5

Changing the number of unions we do with Union Find:

Now we want to see how the unionfind behave when changing the number of unions. We did in total two experiments.

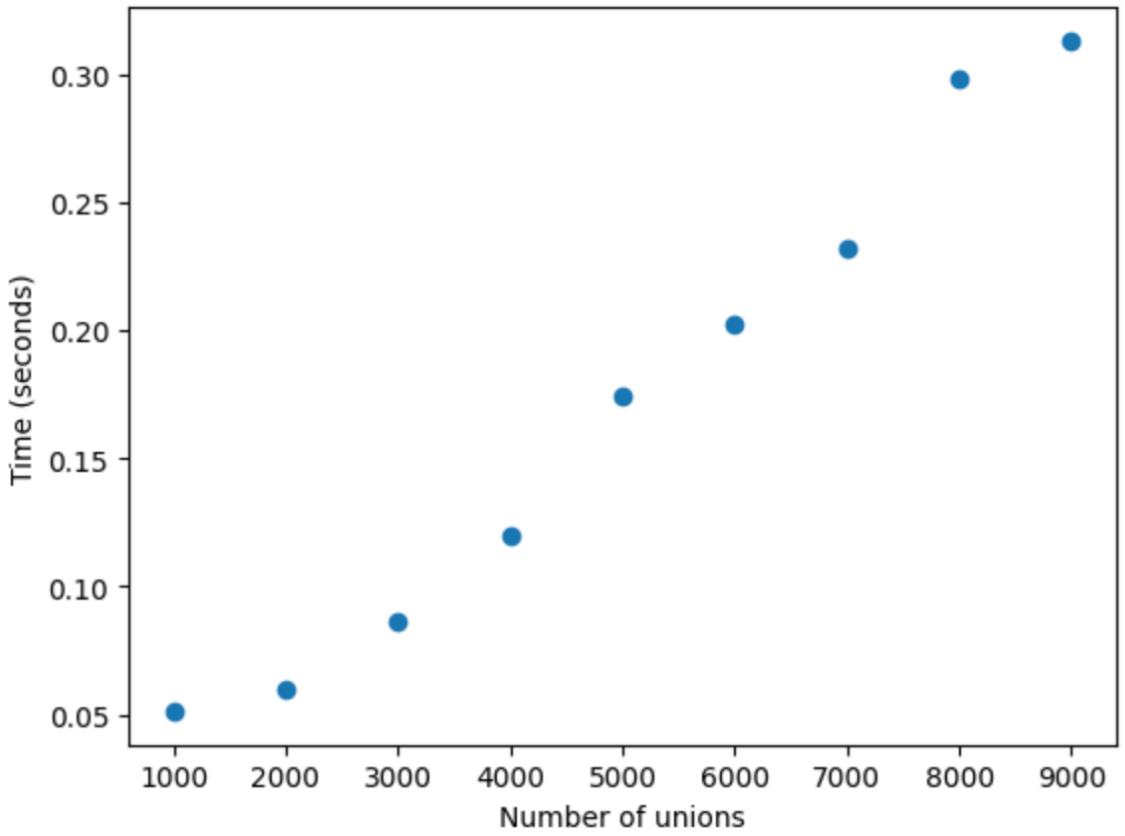
In the first one we decided to unify 0 and i + 1 where i grow for each iteration so that each iteration we keep growing the same union component. We did two loop one to increase the number of unions we do each iteration and the other that indicates the unions we do.From our experiment we get the following results:

Number of unions	Time in seconds
1000	0.0512601
2000	0.0600504

3000	0.0863881
4000	0.120035
5000	0.174656
6000	0.202102
7000	0.231813
8000	0.297899
9000	0.313029

File of the experiment: `ex4numberunion.cpp`

With the class Union Find, we expect that the time will increase linearly because the method union takes always the same time (when N size of the vector is fixed) and we repeat it more times, so that increases the time in a constant way.



from the graph we can see that the growth is approximately linear. There are some fluctuations probably due to system conditions. From calculating the slope we can see as well that it is not precisely linear but in general we have that

$\frac{0.313029-0.0863881}{9000-3000}$ = 3.7773e-05

$\frac{0.231813-0.120035}{7000-4000}$ = 3.7259e-05

so we can say that the growth is linear $y = mx + b$ where

$m = \frac{0.313029-0.0863881}{9000-3000} = 3.7773e-05$

and the intersection is

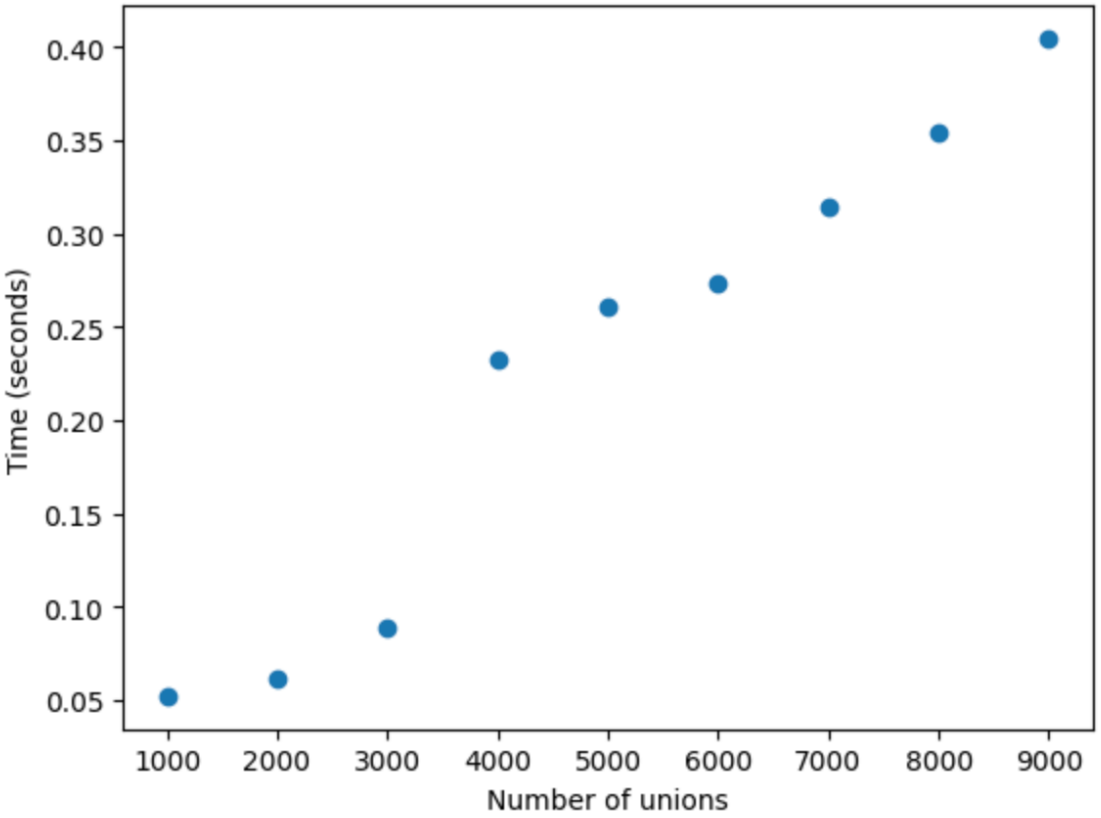
$b = -(3.7773e-05 * 9000) + 0.313029 = -0.0269$

In the second experiment we unified random numbers. We got the following results :

Number of unions	Time in seconds
1000	0.0522336,
2000	0.0617212
3000	0.0885951
4000	0.23252
5000	0.260492
6000	0.273465
7000	0.314363
8000	0.353936
9000	0.404491

File of the experiment: `ex4_uf.cpp`

In this case as well we expect the growth to be linear because it doesn't matter which numbers we unify.



As you can see in the graph in this case as well the growth seems linear. In fact if I calculate the slope we get

m=(0.353936 - 0.0885951)/(8000-3000) = 5.3068e-05
m=(0.314363 - 0.0617212)/(7000-2000)= 5.0528e-05

that means it is linear $y = mx + b$ where
$m = \frac{0.353936 - 0.0885951}{8000-3000} = 5.3068e - 05$
and the intersection is
$b = -(5.3068e - 05 * 8000) + 0.353936 = -0.0706$

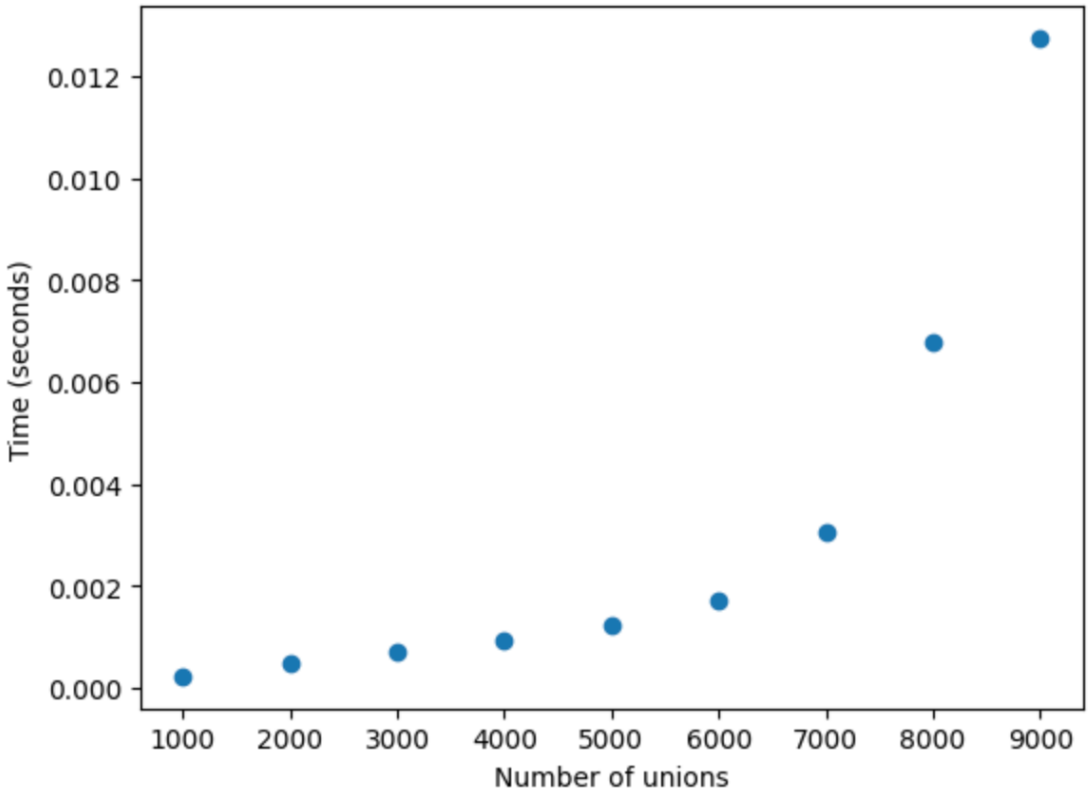
Changing the number of unions we do with Quick Union Find:

Now we repeated the experiment with quick union find. We studied what happened when we increased the number of union using random unions. That means we used random a and b and unified them. We got the following results

Number of unions	Time in seconds
1000	0.000227106

2000	0.000466794
3000	0.000697218
4000	0.000945997
5000	0.00122984
6000	0.00172086
7000	0.00305899
8000	0.00677331
9000	0.0127499

Name of the experiment: `ex4rootnumberunion2.cpp`



We can see rapid and accelerating growth and this is due to the fact that the distance to the root is bigger every time because we keep adding more nodes randomly. Finding the root in later iterations takes significantly more time compared to earlier iterations, leading to the observed rapid growth in the execution time.

Also by doing the mathematical analysis, we can see that the growth is not linear:

$m_1 = (0.0127499 - 0.000227106)/(9000-1000) = 0.00000156534925 = 1.56534925 * 10^{-6}$
$m_2 = (0.000945997 - 0.000227106)/(4000-1000) = 0.000000239630333 = 2.39630333 * 10^{-7}$

$m_1 \neq m_2 \rightarrow$ not linear

In general we can say that for the quickunion the time taken to do the union depends on the shape of the tree, we can for example have a balanced tree or a highly unbalanced tree and this will affect the union time complexity.

Problem 5

Implementation in the file ex5.cpp

Problem 6

Implementation in the file ex6.cpp

Problem 7

Experiment setup: the files used to calculate the time and to draw the graphs are ex7brute.m, ex7caching.m, ex7.cpp and ex7quicker.cpp

In this exercise we used the two versions of sum3 implemented in problem 5 and 6 with different sizes of vectors in order to compare them and analyze the growth of the algorithms. We set up the experiment by creating arrays with random numbers which size growth through an iteration, used 3sum on each array and timed the duration of the 3sum function.
Using the first 3sum we measured the time that is shown in the table and we plotted them using matlab as shown in fig 7.1.

vector size	times (seconds)
1	2.2740e-06
2	1.8080e-06
3	1.8630e-06
10	2.1092e-05
20	2.4509e-04
30	6.8580e-04
40	0.0015
50	0.0023
60	0.0046
70	0.0070
80	0.0101
90	0.0121
100	0.0168
200	0.2179
300	0.6391
400	1.4553

By looking to the code we expect brute 3 force to be $O(N^3)$ in fact 3 loops are present in the code and they iterate the vector v.size( ) times, this means I have three levels of nested loops. If we look at the graph we expect it to be a power of 3. We must then find b and c in the formula
log2 time(x) = b · log2 x+c
As expected if we calculate the slope we get

$$b = \frac{log_2(0.6391) - log_2(0.0023)}{log_2(300) - log_2(50)} = 3.1525$$

and the intercept is

$c = log_2(0.00225125) - 3.1525 * log_2(50) = -26.5873$

so

$$log_2\, time(x) = 3.1525 * log_2(x) - 26.5873$$

moving to the powerlaw

$$2^{log_2\, time(x)} = 2^{3.1525 * log_2(x) - 26.5873}$$

$$time(x) = x^{3.1525} \times 2^{-26.5873}$$

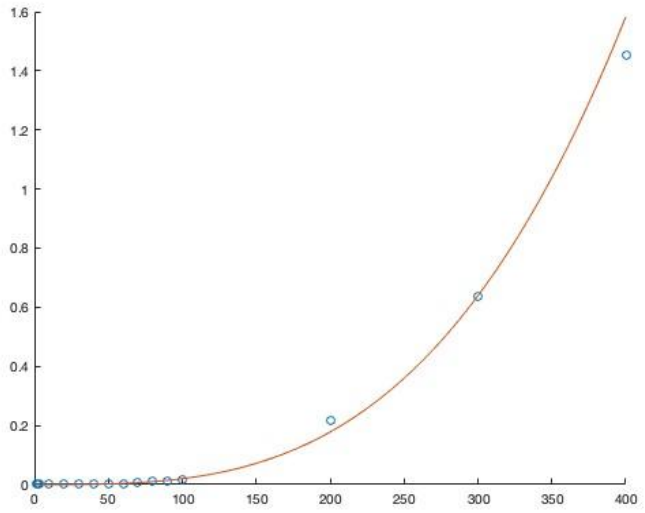


fig 7.1

Using instead the second implementation of 3sum we get the times that are shown in the table and we plot them using python as shown in fig 7.2. This time by looking to the code we expect 3 sum to be $O(N^2)$ in fact you have an O(N) loop nested within another O(N) loop. If we look at the graph we expect it to be a power of 2. We must then find b and c as we did before

vector size	times (seconds)
1	4.1049e-05
2	3.7480e-05
3	8.0970e-06
10	4.1520e-05
20	1.4716e-04
30	3.4075e-04
40	3.4856e-04
50	7.5117e-04
60	0.0013
70	0.0010
80	0.0017
90	0.0014
100	0.0018
200	0.0118
300	0.0224
400	0.0373

$$b = \frac{log_2(0.0224) - log_2(3.4856e{-}04)}{log_2(15) - log_2(40)} = 2.0655$$

c=log2(0.000751172)-2.1575*log2(50) = - 22.5552

so

$$log_2\, time(x) = 2.0655 \times log_2(x) - 22.5552$$

moving to the powerlaw

$$2^{log_2\, time(x)} = 2^{2.0655 * log_2(x) - 22.5552}$$

$$time(x) = x^{2.0655} \times 2^{-22.5552}$$

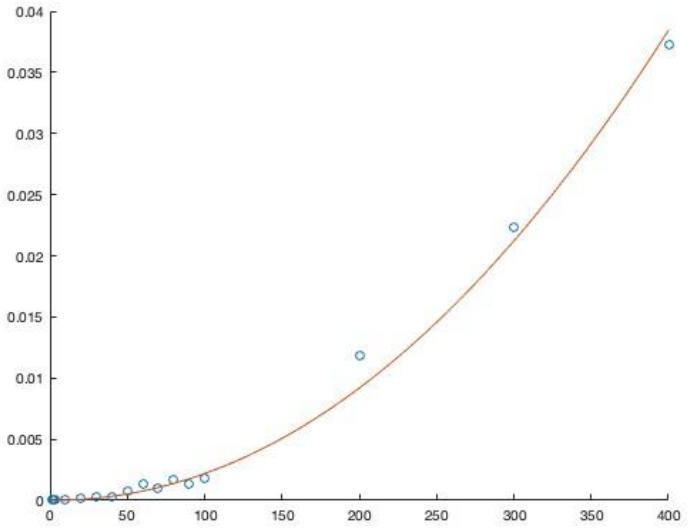


fig 7.2

In general both the results of the analysis match the expectation even if the power we find is not exactly 3 in the first 3sum and 2 in the second due to the fact that the exact time complexity depends on various factors, including the specific random input data and system conditions.

Problem 8

Code in the file ex8.cpp

We create a class called percolation that gets two inputs: an integer N which indicates the size of the grid and an instance of the class Union Find that has been created with the length N*N because it's the number of values that the grid is going to have. In the constructor we initialize the grid as a vector of vector of integers. The vector is gonna contain N vectors of size N. We assign values from 0 to N*N - 1 to each position. If N = 10, the grid will look like this:

[[0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
[20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
[30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
[40, 41, 42, 43, 44, 45, 46, 47, 48, 49],
[50, 51, 52, 53, 54, 55, 56, 57, 58, 59],
[60, 61, 62, 63, 64, 65, 66, 67, 68, 69],
[70, 71, 72, 73, 74, 75, 76, 77, 78, 79],
[80, 81, 82, 83, 84, 85, 86, 87, 88, 89],
[90, 91, 92, 93, 94, 95, 96, 97, 98, 99]]

We have two methods in the class: open_site() and percoletes(). The first one gets an integer as the input, and it is gonna have a value from 0 to N*N - 1. This method is used to open a site and check if the ones around it are opened as well and we need to connect them. We have to check the sites in the right, left, above and below. We know if a site is opened if it is in the set called *opened*. The one in the right is gonna have the value of the current site + 1 and we have to check that the current site is not in the right border. We do the same for the other 3 directions. The method percoletes() will return True if it percolates and False if it doesn't. We know that it percolates if an element from the first row and the last row are in the same component.

We create an instance of this class in the main. We generate a random number from 0 to N*N-1 to choose the site that we open, we check that the chosen site hasn't been opened yet, we call the method open_site(site) and we call the method percolates(). We keep doing this until it percolates and when it does we save the number of sites we have had to opened in order for it to percolate. We repeat all of this 50 times and we do the average between the opened sites it has taken each time.

We obtain an average of 58.98 with 100 total sites. That means that the estimated threshold p* using Monte Carlo is 0.58. The sites are set to open with probability p, if $p<0.58$ a random nxn grid almost never percolates and when $p>0.58$ it almost always percolates.