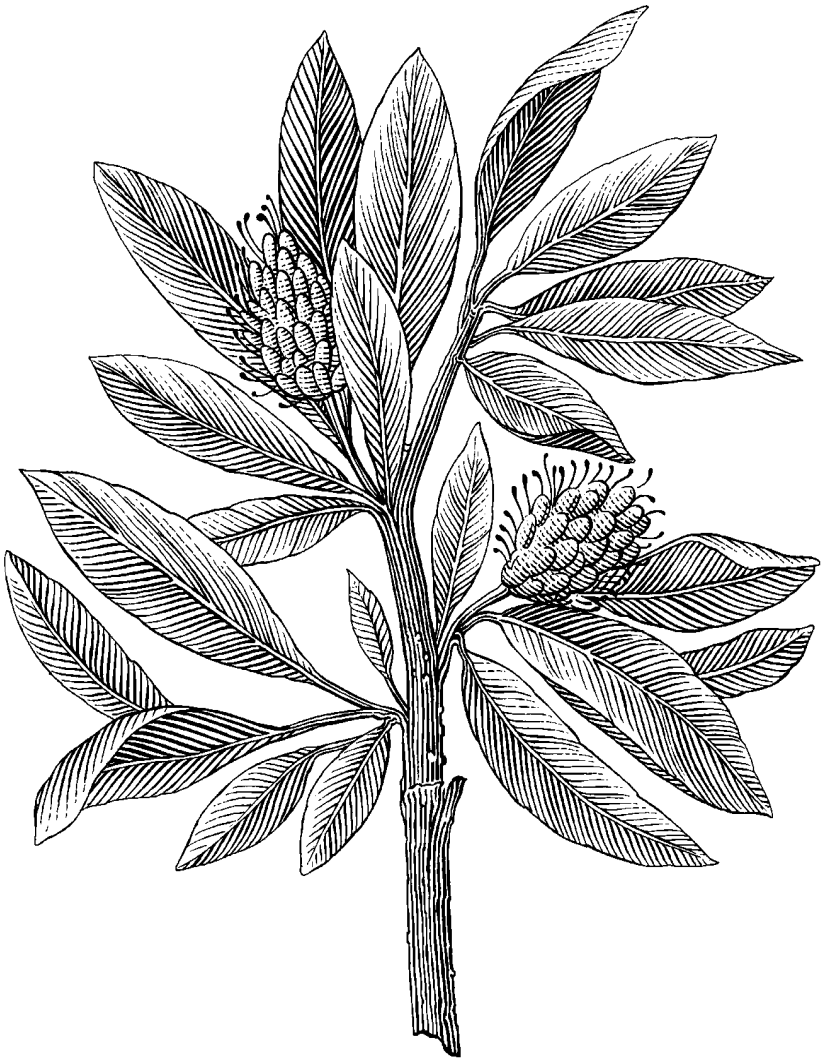
Linnéuniversitetet
Kalmar Växjö

Report

Assignment 1

1DV516



Supervisor: Morgan Ericsson
Semester: Autumn 2023

Author1: Katarina Simakina
Email es225hi@student.lnu.se

Author2: Zejian Wang
Email zw222bb@student.lnu.se

Table of Contents

Task 4 _____3

Task 7 _____7

Task 8 _____10

Task 4

Table_1 presents the timing results for two variants of the Union-Find data structure: QuickFind (Task 1) and QuickUnion (Task 2). We measured the time taken by various union operations with different array sizes. All decimals results are rounded to four decimal places.

Table_1

Number of Array size	**Number of Union Operations**	**Task 1 (QuickFind) Results in seconds**	**Task 2 (QuickUnion) Results in seconds**
50000	500	0.1201	12.0000E-4
	1000	0.2345	8.4260E-4
	1500	0.4271	15.0000E-4
	2000	0.4477	8.5819E-4
100000	500	0.2035	3.2180E-4
	1000	0.3465	3.7320E-4
	1500	0.5243	5.6610E-4
	2000	0.7174	8.0669E-4
150000	500	0.2186	2.6790E-4
	1000	0.4156	5.0340E-4
	1500	0.5888	7.1459E-4
	2000	0.7935	8.9149E-4
200000	500	0.3291	4.7740E-4
	1000	0.5847	5.9380E-4
	1500	0.7392	7.4340E-4
	2000	1.0296	9.9649E-4

From the table_1, we can see that QuickUnion consistently outperforms QuickFind for given array sizes and number of union operations. As the array size and the number of union operations increase, the execution time for QuickFind and QuickUnion also increases.

However, there are some statistics that do not match our expectations when the array length is 50000 as table shows, for example, the execution time for 1000 operations is smaller than 500 operations. This may be because for QuickUnion, it's a tree-based data structure, the time complexity is O(h), h is the height of the tree, the searching path of the root of the element when doing the union operation may be longer even if the amount of elements are smaller.

Mathematical calculations process are below:

Array size = 100 000 ;

Task1(QuickFind):

Slope = ($\log_2$(0,3465)- $\log_2$ (0,2035)) / ($\log_2$ 1000 - $\log_2$500)= (-1.529 - (´-2.297))/ (9.966-8.966) =0,768 /1= 0.768 ≈ 0.76

Intercept = $\log_2$(0,3465) - 0.76($\log_2$1000) = -1.529 - 0.76 x 9.966= - 9.10316 ≈ - 9.1

Log_2 time (x) = 0.76 x $\log_2$x-9.1

Moving to a power law:

a · x^b

a = 2^c

2^-9.1 x X^0.76

Task2 (QuickUnion):

Slope = ($\log_2$(3,7320)- $\log_2$ (3,2180)) / ($\log_2$ 1000 - $\log_2$500)= (1.9 -1.686) / (9.966-8.966) =0,214 /1= 0.214 ≈ 0.21

Intercept = $\log_2$(3,7320) - 0.21($\log_2$1000) = 1.9 - 0.21 x 9.966= - 0.19286 ≈ - 0.19

Log_2 time (x) = 0.21 x $\log_2$x - 0.19

Moving to a power law:

a · x^b

a = 2^c

2^-0.19 x X^0.21

Array size = 150000;

Task1(QuickFind).

Slope = ($\log_2$(0,4156)- $\log_2$ (0,2186)) / ($\log_2$ 1000 - $\log_2$500)= (-1,2667 - (-2.1936))/ (9.966-8.966) = 0.9269≈ 0.93

Intercept = $\log_2$(0,4156) - 0.93($\log_2$1000) = -1.2667- 0.93 x 9.966= - 10.53508 ≈ - 10.53

Log_2 time (x) = 0.93 x $\log_2$x-10.53

Moving to a power law:

a · x^b

a = 2^c

2^-10.53 x X^0.93

Task2 (QuickUnion):

Slope = ($\log_2$(5.0340)- $\log_2$ (2.6790)) / ($\log_2$ 1000 - $\log_2$500)= (2.3317 - 1.4217)/ (9.966-8.966) = 0.91

Intercept = $\log_2$(5.0340) - 0.91($\log_2$1000) = 2.3317- 0.91 x 9.966= - 6,73736 ≈ - 6,73

Log_2 time (x) = 0.91 x $\log_2$x-6.73

Moving to a power law:

a · x^b

a = 2^c

2^-6.73 x X^0.91

Array size = 200000;

Task1(QuickFind):

Slope = ($\log_2$(1.0296)- $\log_2$ (0,7392)) / ($\log_2$ 2000 - $\log_2$1500)= (0.04208 - (-0.436)) / (10.966-10.55) = 0.47808/ 0.416 = 1.0901 ≈ 1,09

Intercept = $\log_2$(1.0296) - 1.09($\log_2$2000) = 0.04208- 1.09 x 10.966= - 11.91086 ≈ - 11,91

Log_2 time (x) = 1.09 x $\log_2$x - 11,91

Moving to a power law:

a · x^b

a = 2^c

2^-11,91 x X^1,09

Task2 (QuickUnion):

Slope = ($\log_2$(9.9649)- $\log_2$ (7,4340)) / ($\log_2$ 2000 - $\log_2$1500)= (3.317- 2.894) / (10.966-10.55) =0.423/ 0.416 = 1.0168269 ≈ 1,01

Intercept = $\log_2$(9.9649) -1.01 ($\log_2$2000) = 3.317- 1.01 x 10.966= -7.75866 ≈ - 7.76

$\log_2$ time (x) = 1,01 x $\log_2$x -7.76

Moving to a power law:

a · x^b

a = 2^c

2^-7.76 x X^1,01

Summarizing calculations above in the table_2 below.

Table_2

Array size	Parameters	Task 1 (QuickFind) Results	Task 2 (QuickUnion) Results
100 000	Slope	0.76	0.91
	Intercept	-9.1	-6.73
	Power law	2^-9.1 x X^0.76	2^-6.73 x X^0.91
150 000	Slope	0.93	0.91
	Intercept	-10.53	-6.73
	Power law	2^-10.53 x X^0.93	2^-6.73 x X^0.91
200 000	Slope	1,09	1,01
	Intercept	-11,91	-7.76
	Power law	2^-11,91 x X^1,09	2^-7.76 x X^1,01

From the table above, we can see that the power law of union operation for QuickFind is approximately 1. The slope tends to 1 as linear, the algorithm for this task considers time complexity O(N), this is because when doing the union operation, in the worst case, it should iterate the whole list to set the id of one element as the id of the other.

The same tendency is for QuickUnion (task 2), which is a tree-based structure, as the array size grows, larger and more complex trees can form, leading to longer paths during find and union operations, thus causing increased execution times. As mentioned earlier, it is a tree-based data structure, when doing the union operation for 2 elements, it should find the root for them, of which the time complexity is O(h), h is the height of the tree, in the worst case, the height of the tree is the number of all elements, so the time complexity is O(n), as the power-law values show in the table, the slope is 0.91 when the array size is 100000 and 150000, and 1.01 when the array size is 200000. The slope has a tendency to be around 1 which also proves the time complexity O(N).

Task 7

For implementation of task 7 we have done the $O(N\text{^}3)$ brute-force variant of 3sum (task5) and caching variant of 3 sum(task6). Table 3 displays the running time data by the two different 3sum algorithms with various array sizes.

Table_3

Array size	**Task 5 ($O(N\text{^}3)$ brute-force variant of 3sum) Results in seconds**	**Task 6 (version of 3sum with an upper bound that is lower than $O(N\ 3)$) Results in seconds**
500	0.047 seconds	0.02 seconds
1000	0.208 seconds	0.027 seconds
1500	0.474 seconds	0.063 seconds
2000	0.968 seconds	0.021 seconds

2500	1.952 seconds	0.033 seconds
3000	3.27 seconds	0.049 seconds

We can see from the table above that the performance of caching 3Sum is always better than brute-force 3Sum for a given input array size, mathematical calculations process are below:

Task5(Brute-force 3sum)):

Slope = ($\log_2(1.952) - \log_2(0.968)$) / ($\log_2 2500 - \log_2 2000$)= (0.965 - (-0.04692) / (11.288-10.966) =1.01192 /0.322= 3,1426 ≈ 3,14

Intercept = $\log_2(1.952)$ - 3.14($\log_2 2500$) = 0.965 - 3.14 x 11.288=-34,47932 ≈ -34,48

Log_2 time (x) = 3,14 x $\log_2 x$-34,48

Moving to a power law:

$a \cdot x^b$

$a = 2^c$

$2^{-34,48} \times X^{3,14}$

Task5(Brute-force 3sum)):

Slope = ($\log_2(3.27) - \log_2(1.952)$) / ($\log_2 3000 - \log_2 2500$)= (1.7093 - 0.965) / (11.55-11.288) =2.84083 ≈ 2.8

Intercept = $\log_2(3.27)$ - 2.8($\log_2 3000$) = 1.7093 - 2.8 x 11.55 = - 30.6307 ≈ - 30.63

Log_2 time (x) = 2.8 x $\log_2 x$ - 30.63

Moving to a power law:

$a \cdot x^b$

$a = 2^c$

$2^{} - 30.63 \times X^{2.8}$

Task 6 (Caching 3Sum):

Slope = ($\log_2(0.033) - \log_2(0.021)$) / ($\log_2 2500 - \log_2 2000$)= (-4.921- (-5.573) / (11.288-10.966) =0.649/0.322= 2.015527 ≈ 2.01

Intercept = $\log_2(0.033)$ - 2.01($\log_2 2500$) = -4.921 - 2.01 x 11.288=-27,60988 ≈ -27,60

Log_2 time (x) = 2.01x $\log_2 x$ -27,60

Moving to a power law:

$a \cdot x^{\wedge}b$

$a = 2^{\wedge}c$

$2^{\wedge}- -27{,}60 \times X^{\wedge}2.01$

Task 6 (Caching 3sum):

Slope = $(\log_2(0{,}049) - \log_2(0.033)) / (\log_2 3000 - \log_2 2500) = (-4.351 - (-4.921)) / (11.55-11.288) = 0.47/0{,}262 = 2.175572 \approx 2.17$

Intercept = $\log_2(0{,}049) - 2.17(\log_2 3000) = -4.351 - 2.17 \times 11.55 = -29.4145 \approx -29.41$

Log_2 time (x) = $2.17 \times \log_2 x - 29.41$

Moving to a power law:

$a \cdot x^{\wedge}b$

$a = 2^{\wedge}c$

$2^{\wedge} - 29.41 \times X^{\wedge}2.17$

Calculations presented in table_4 below:

Table_4

Array sizes	Parameters	Task 5 (brute-force 3Sum) Results	Task 6 (caching 3Sum) Results
2000~2500	Slope	3,14	2.01
	Intercept	-34,48	-27,60
	Power-law	2^-34,48 x X^3,14	2^- 27,60 x X^2.01
2500~3000	Slope	2.8	2.17
	Intercept	- 30.63	- 29.41
	Power-law	2^ - 30.63 x X^2.8	2^ - 29.41 x X^2.17

The table represents the results of task5 and task6 related to the "3SUM" analyzing the time complexity of different algorithms. Here are some conclusions we can draw from the table:

For task5(brute-force three sum), from the array size range 2000-2500 and the array size 2500-3000, the time complexity follows a power-law relation with the slope of approximately 3(3.14, 2.8), which means that the complexity is O(n^3).

The intercept values for Task 6 are generally less negative than those for Task 5, suggesting that the caching-3SUM (Task 6) performs better than the brute-force 3Sum (Task 5) across both array size ranges.

For task6(three sum by caching), from the array size range 2000-2500 and the array size 2500-3000, the time complexity follows a power-law relation with the slope of approximately 2 (2.01, 2.17), which means that the complexity is O(n^2), showing a more efficient time complexity than task5. It matches our expectation because in task5, there are three nested loops compared to two nested loops in task6 when doing the searching and addition operations.

Task_8

In task 8, UnionFind(QuickUnion in task2) has been used to solve the percolation problem. In our model, there are n*n(grid) + 2(one is at virtual top and the other is at virtual bottom) sites. The virtual top site is connected to all the sites at the first row, the virtual bottom site is connected to all the sites at the bottom row. Initially all sites are closed, after randomly opening a site in the grid, the Connected method in QuickUnion is used to determine whether the virtual top site and virtual bottom site is connected, if it is connected, the system is percolated, if its not, the union method in QuickUnion has been used to connect a site to its neighbors(up, down, left and down) if they are open.,. When it is percolated, we calculate the percentage of the open sites in all sites. The formula of the probability is (number of open sites) / (all sites), after a certain amount of probability calculations, we can calculate the average probability P* which is the average threshold of percolation for the system, we also calculate the standard deviation of the probability.

We created a class Percolation with its attributes QuickUinon object and boolean grid[][] and its according operations to simulate the percolation process, grid[][] presentes all sites in the grid, and it's the boolean value presents if the site is open or closed, if a site is open, connects all its open neighbor sites(up, down, left, right) using open operation. The percolates() operation is used to check if the virtual top site is connected to the virtual bottom site by the connected() method of its

attribute QuickUnion. The Main class to test the percolation and do a certain time of runs to calculate the average probability.

In our model, the grid size has been set to 20 * 20, and the amount of probability calculation is set to 10000 times. In one run of the model, the average threshold P* = 0.5921830000000001 and standard deviation is 0.04828545494779934, which is sharp enough. As the following shows, each time running our model, the result is a bit different, but they are overall the same.

```
10000 percolates:
The average threshold P* is 0.5921830000000001
The standard deviation of threshold is 0.04828545494779934
```

.