Part 1 - Analysis of search algorithms

Q1.

As search_strategy_1() is linear search, iterating in order until we find the Entry, it is:

Within O(n) // Our Entry is the last in the list

Within Ω(1) // We find at the first iteration, constant

As our Big O and Ω are different, we do not have a Big θ, but our average time complexity will follow a growth of about (n / 2), θ(n), which is linear.

As search_strategy_2() is a variation of binary search, halving its problem size every iteration;

Not accounting for the time complexity of first sorting, it is:

Within $O(\log_2 n)$ which is within O(logn)

// We remove half our input logn times, until we reach our Entry

Within Ω(1) // Best case—our first Entry is (0 + size) / 2

Our average cases will have a growth of logn, but we do not have a Big θ

As search_strategy_3() looks through a hash bucket, it is:

Within O(n) // All our hashes are mapped to the same bucket, and we have to iterate through all of them

Within Ω(1) // Best case—we go to our hash bucket and it is the only or first result there

We do not have a Big θ, but our average line will follow a growth of (load factor + 1), where our load factor is (#elements / #buckets)

Q2.

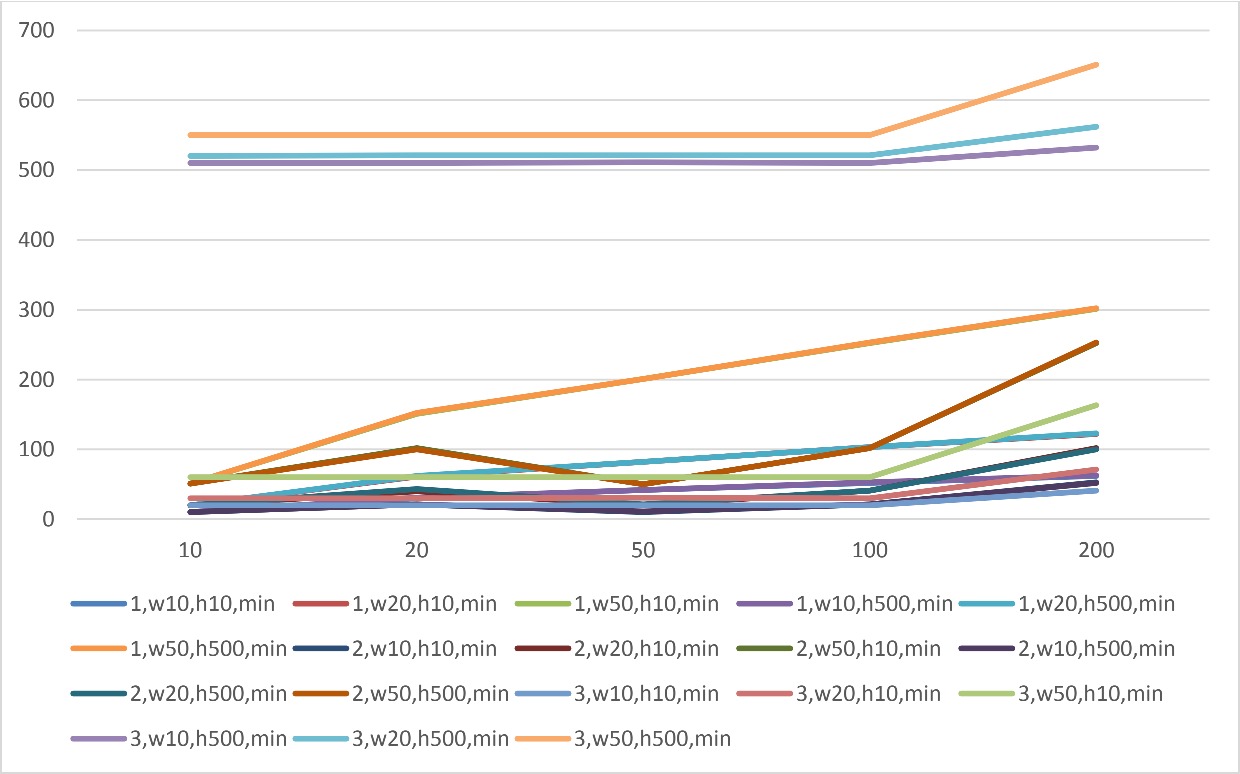
In the following graphs, the lines of graph are formatted as follows, delimited by commas:

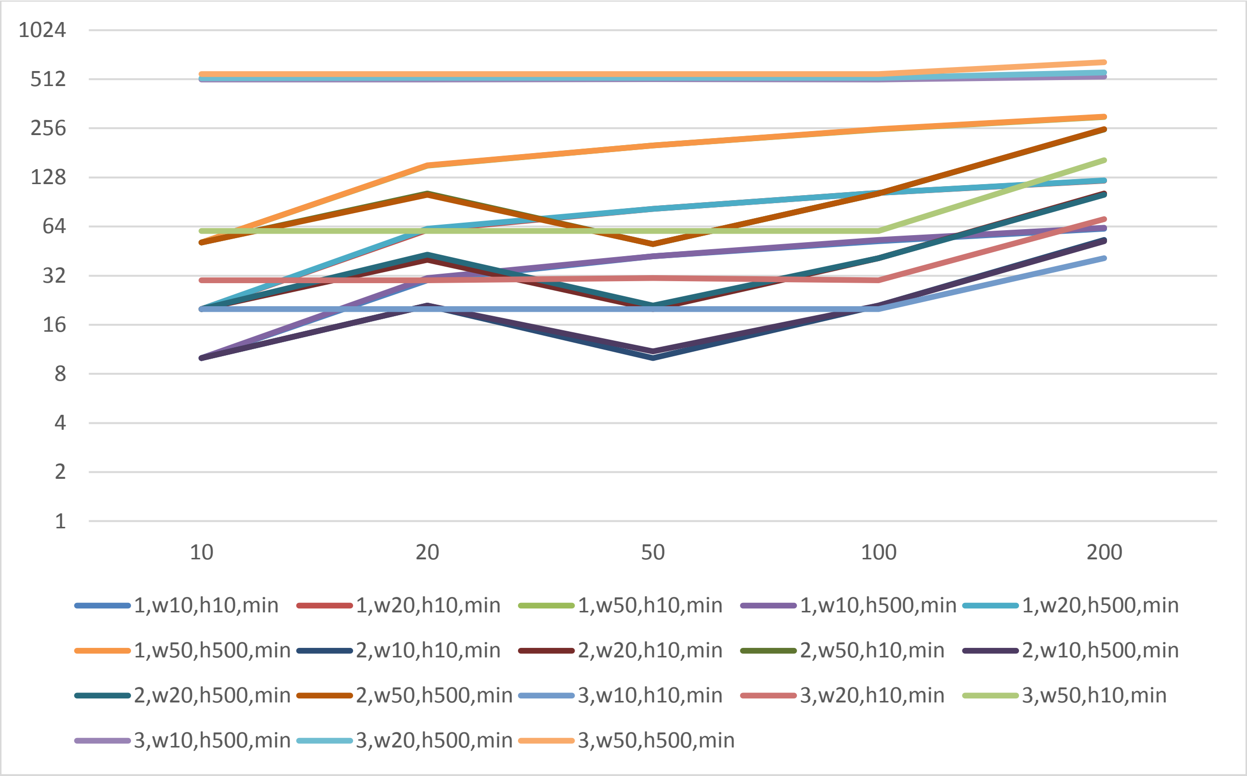
Type of search, work time, hash time, and whether it is the upper, lower, or average bound

For example, “2,w20,h500,max” denotes using:

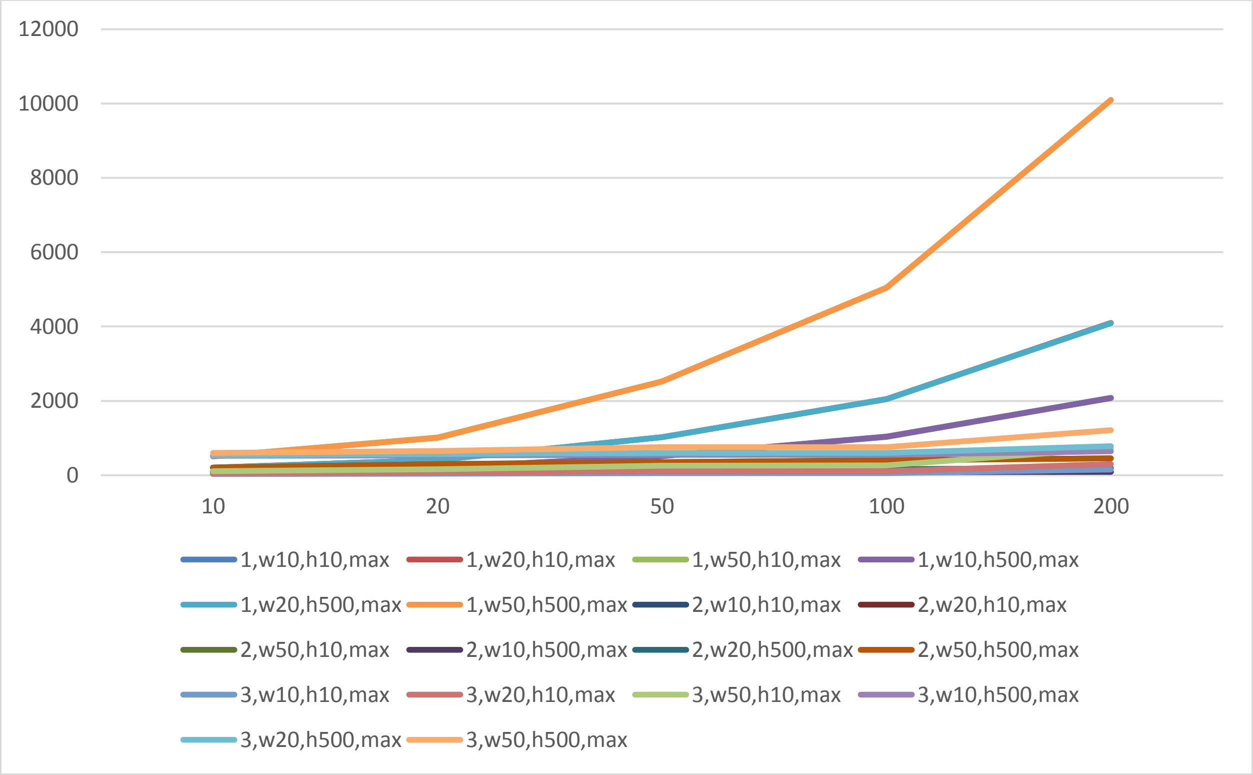
search_strategy_2(), has a work time of 20, a hash time of 500, and is the upper bound.

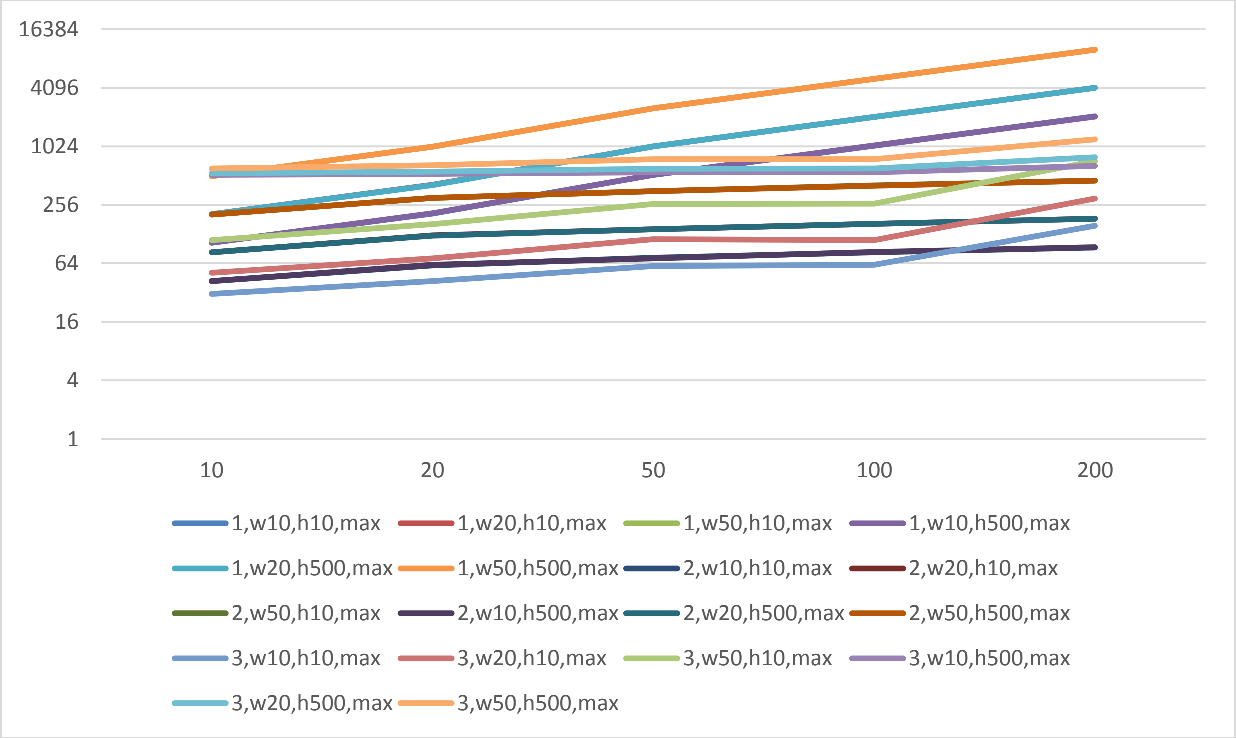
All, Min


700
600
500
400
300
200
100
0
10
20
50
100
200
1,w10,h10,min
1,w20,h10,min
1,w50,h10,min
1,w10,h500,min
1,w20,h500,min
1,w50,h500,min
2,w10,h10,min
2,w20,h10,min
2,w50,h10,min
2,w10,h500,min
2,w20,h500,min
2,w50,h500,min
3,w10,h10,min
3,w20,h10,min
3,w50,h10,min
3,w10,h500,min
3,w20,h500,min
3,w50,h500,min

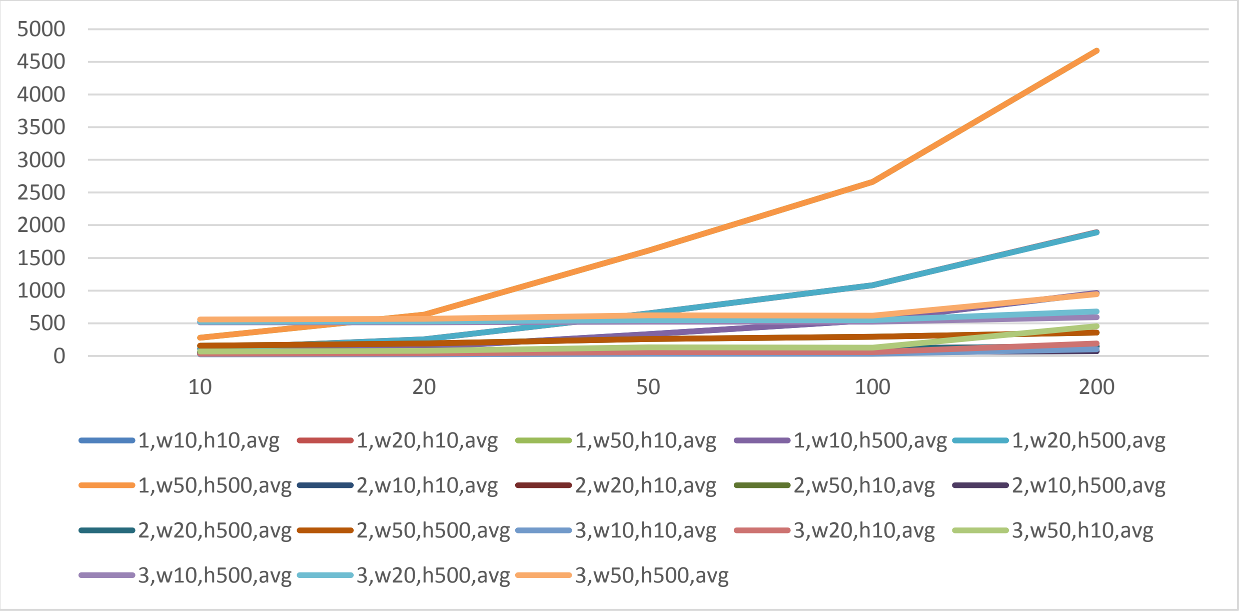

1024
512
256
128
64
32
16
8
4
2
1
10
20
50
100
200
1,w10,h10,min
1,w20,h10,min
1,w50,h10,min
1,w10,h500,min
1,w20,h500,min
1,w50,h500,min
2,w10,h10,min
2,w20,h10,min
2,w50,h10,min
2,w10,h500,min
2,w20,h500,min
2,w50,h500,min
3,w10,h10,min
3,w20,h10,min
3,w50,h10,min
3,w10,h500,min
3,w20,h500,min
3,w50,h500,min

All, Max

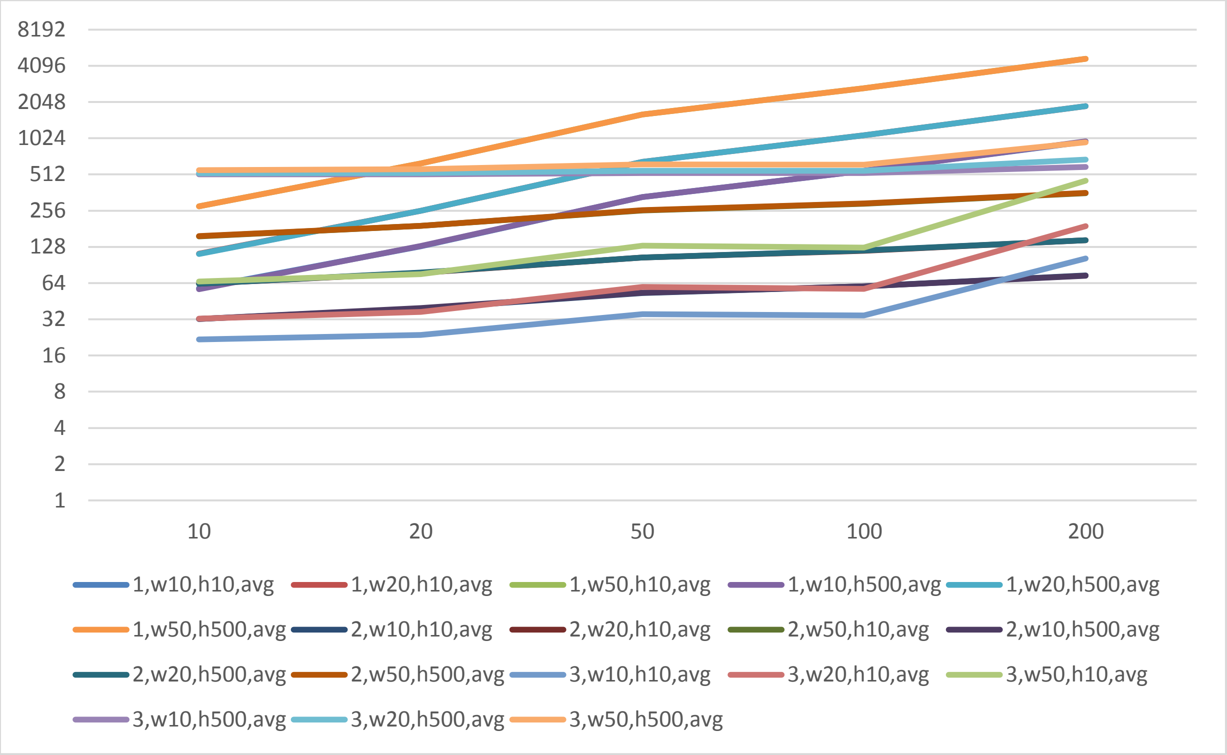

12000
10000
8000
6000
4000
2000
0
10
20
50
100
200
1,w10,h10,max
1,w20,h10,max
1,w50,h10,max
1,w10,h500,max
1,w20,h500,max
1,w50,h500,max
2,w10,h10,max
2,w20,h10,max
2,w50,h10,max
2,w10,h500,max
2,w20,h500,max
2,w50,h500,max
3,w10,h10,max
3,w20,h10,max
3,w50,h10,max
3,w10,h500,max
3,w20,h500,max
3,w50,h500,max


16384
4096
1024
256
64
16
4
1
10
20
50
100
200
1,w10,h10,max
1,w20,h10,max
1,w50,h10,max
1,w10,h500,max
1,w20,h500,max
1,w50,h500,max
2,w10,h10,max
2,w20,h10,max
2,w50,h10,max
2,w10,h500,max
2,w20,h500,max
2,w50,h500,max
3,w10,h10,max
3,w20,h10,max
3,w50,h10,max
3,w10,h500,max
3,w20,h500,max
3,w50,h500,max

All, Mean

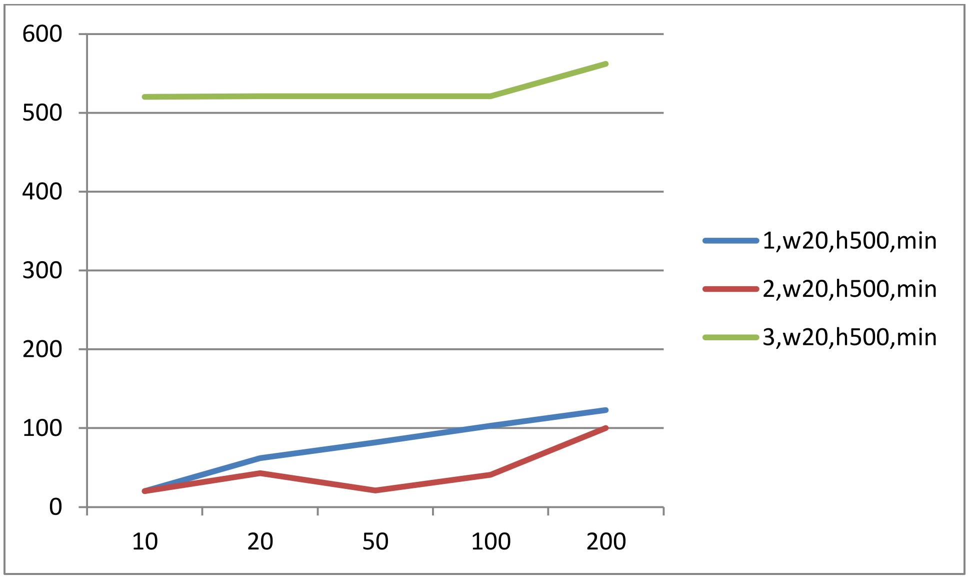


5000
4500
4000
3500
3000
2500
2000
1500
1000
500
0
10
20
50
100
200
1,w10,h10,avg
1,w20,h10,avg
1,w50,h10,avg
1,w10,h500,avg
1,w20,h500,avg
1,w50,h500,avg
2,w10,h10,avg
2,w20,h10,avg
2,w50,h10,avg
2,w10,h500,avg
2,w20,h500,avg
2,w50,h500,avg
3,w10,h10,avg
3,w20,h10,avg
3,w50,h10,avg
3,w10,h500,avg
3,w20,h500,avg
3,w50,h500,avg

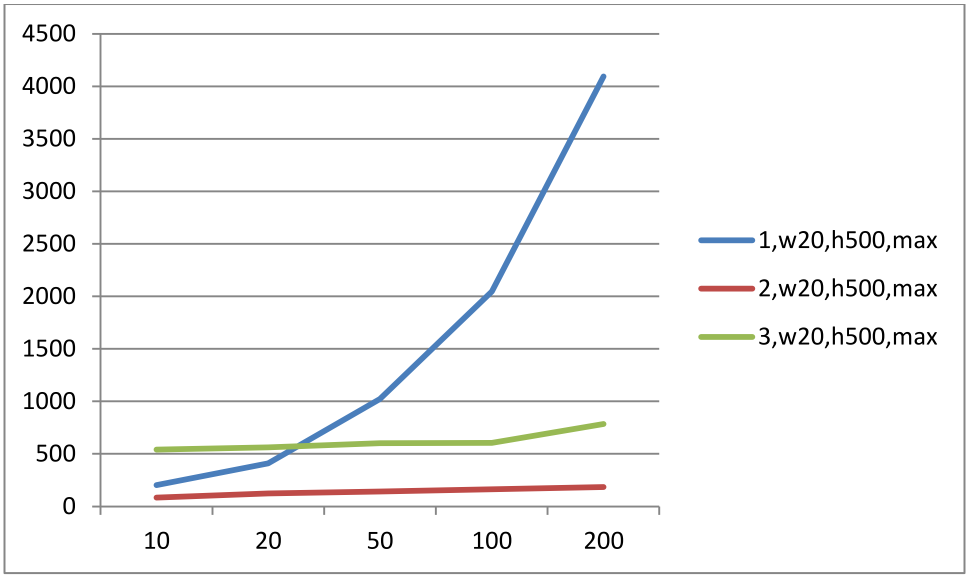


8192
4096
2048
1024
512
256
128
64
32
16
8
4
2
1
10
20
50
100
200
1,w10,h10,avg
1,w20,h10,avg
1,w50,h10,avg
1,w10,h500,avg
1,w20,h500,avg
1,w50,h500,avg
2,w10,h10,avg
2,w20,h10,avg
2,w50,h10,avg
2,w10,h500,avg
2,w20,h500,avg
2,w50,h500,avg
3,w10,h10,avg
3,w20,h10,avg
3,w50,h10,avg
3,w10,h500,avg
3,w20,h500,avg
3,w50,h500,avg

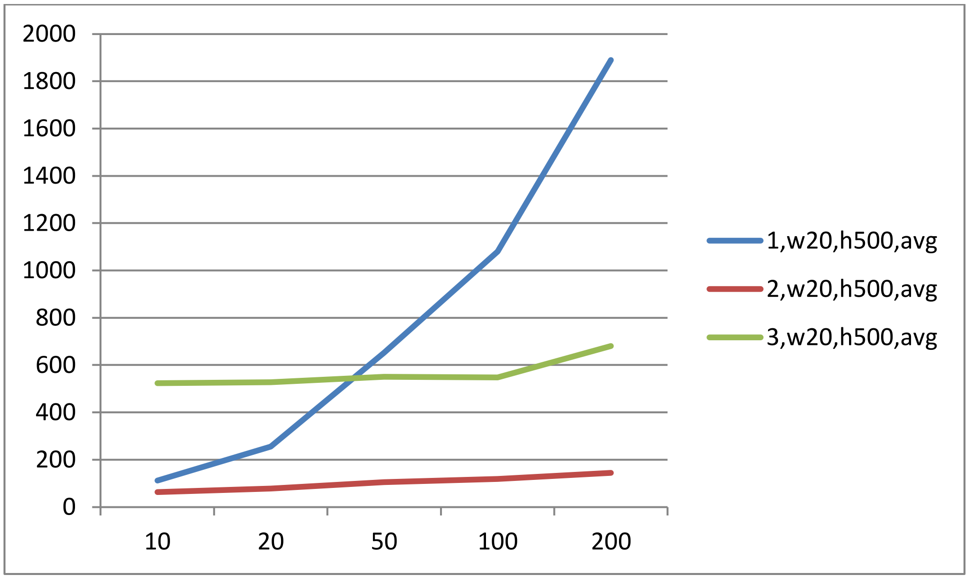
All, Work 20, Hash 500, Min



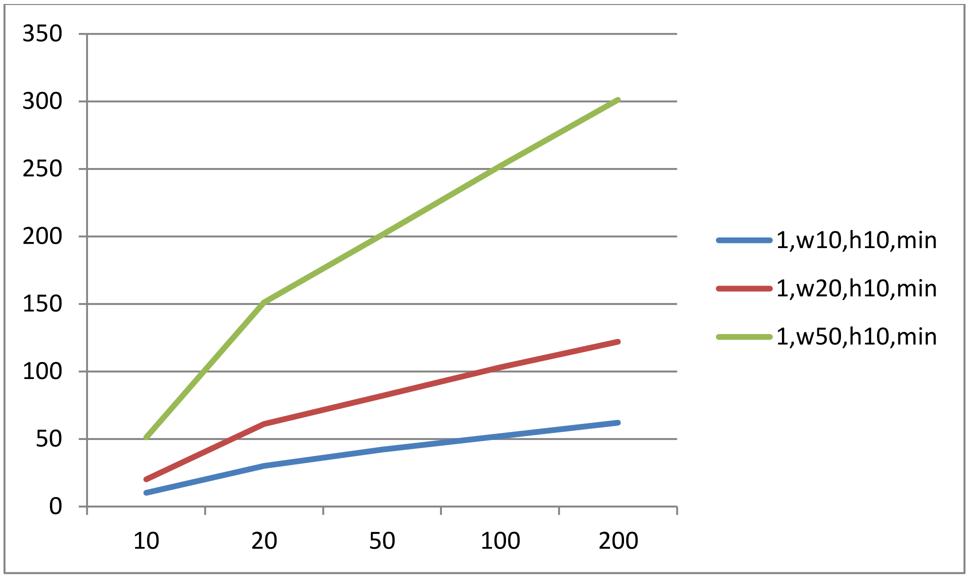
All, Work 20, Hash 500, Max



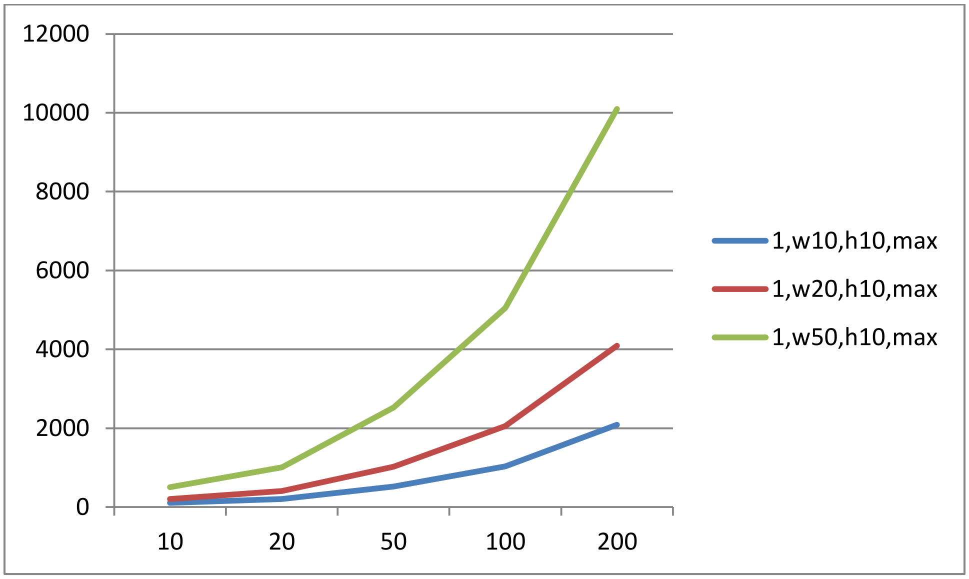
All, Work 20, Hash 500, Mean



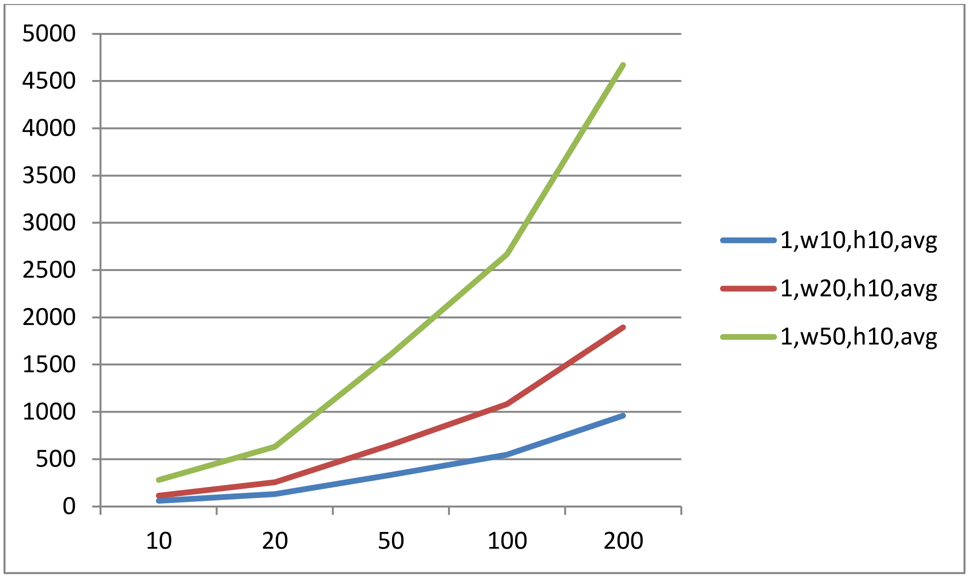
Strategy 1, Work 10, 20, 50, Hash 10, Min



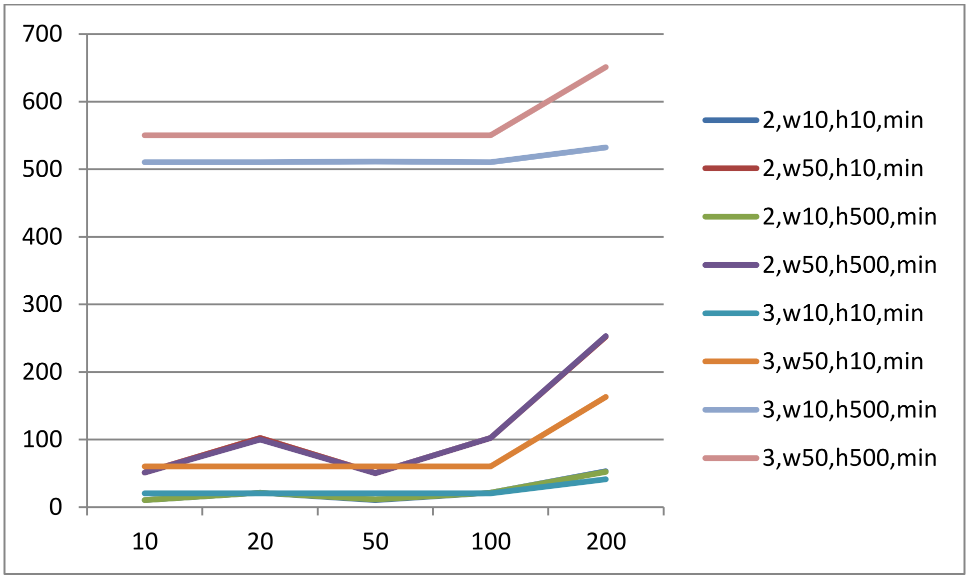
Strategy 1, Work 10, 20, 50, Hash 10, Max



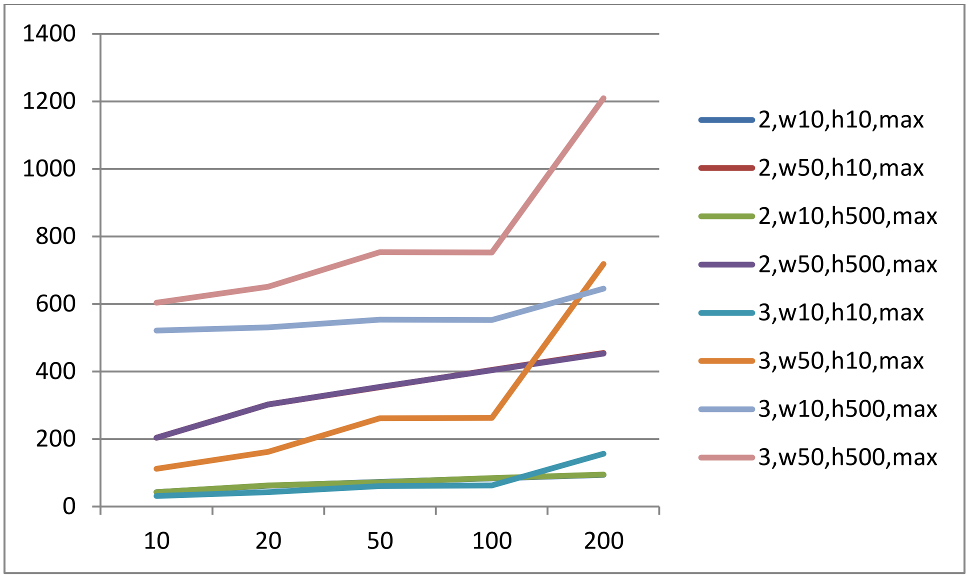
Strategy 1, Work 10, 20, 50, Hash 10, Mean



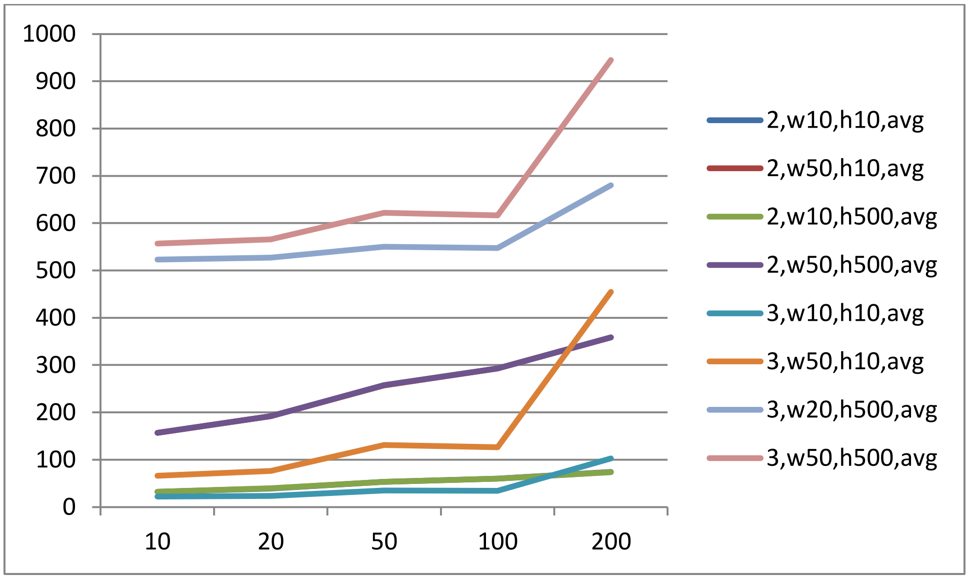
Strategy 3, 2, Work 10, 50, Hash 500, 10, Min



Strategy 3, 2, Work 10, 50, Hash 500, 10, Max



Strategy 3, 2, Work 10, 50, Hash 500, 10, Mean



Q3.

I’d expect Strategy 1 to have a growth rate of c(n), where c is a positive constant. Strategy 2 should have a growth rate of clog(n), and Strategy 3 of c(load factor + 1).

Depending on the workload, the higher it is, the more time it will take, however by a constant factor.

Q4.

Strategy 1’s complexity is as expected, it grows linearly by a constant factor. Strategy 2 grows logarithmically, and Strategy 3 has a low growth rate as well. However, when the hash time is high, Strategy 3 becomes less effective.

Q5.

Strategy 1 is fine if you have a small input size to search from. But as it becomes large, it is the most inefficient algorithm.

Strategy 2 is superior to Strategy 1, usually as long as our input size is over 35.

The most efficient algorithm is Strategy 3, as long as our hash function has a low hash time, it beats Strategy 2.

But as we generally care about our efficiency as our input grows arbitrarily large, we would ditch Strategy 1, and use Strategy 3 if we have a fast hash function; and if not, Strategy 2.

Part 2 - Analysis of sort algorithms

Q1.
sort_strategy_1() uses the principle of selection sort, where it repeatedly finds the minimum Entry, and swap it will the current pointer index, utilizing two for loops of complexity so it is:

Within $O(n^2)$ // The loops will iterate n^2 times regardless

Within $\Omega(n^2)$ // The loops will iterate n^2 times regardless

Within $\theta(n^2)$ // The loops will iterate n^2 times regardless

sort_strategy_2() uses the principle of bubble sort

so it is: Ω are different, we do not have a Big θ

Within $O(n^2)$ // The loops will iterate n^2 times regardless

Within $\underline{\Omega(n)}$ // Our entries are already sorted, or after one iteration of the outer loop, it is sorted

As our Big O and Ω are different, we do not have a θ. However, an average list will perform about n^2 operations, so our growth rate is around n^2

sort_strategy_3() uses the principle of quick sort, divide and conquering and uses pivots to move each Entry to the correct position. So it is:

Within $\underline{O(n^2)}$ // The pivots are always chosen to be either the largest or smallest value in the list.

Within $\underline{\Omega(n\log n)}$ // Our entries are sorted after choosing nlogn pivots

As our Big O and Ω are different, we do not have a θ. However, an average list will perform about nlogn operations, so our growth rate is around nlogn

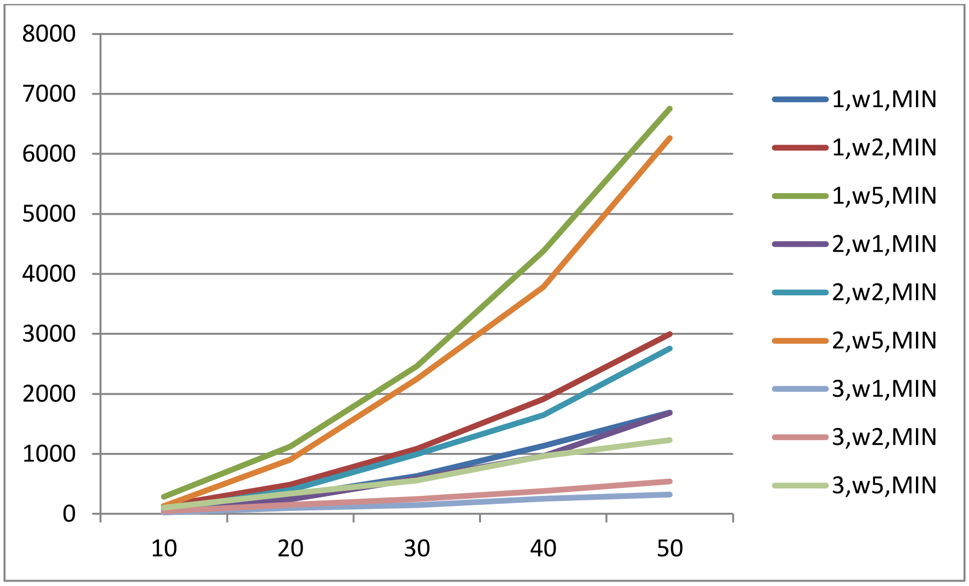
Q2.

In the following graphs, the lines of graph are formatted as follows, delimited by commas:

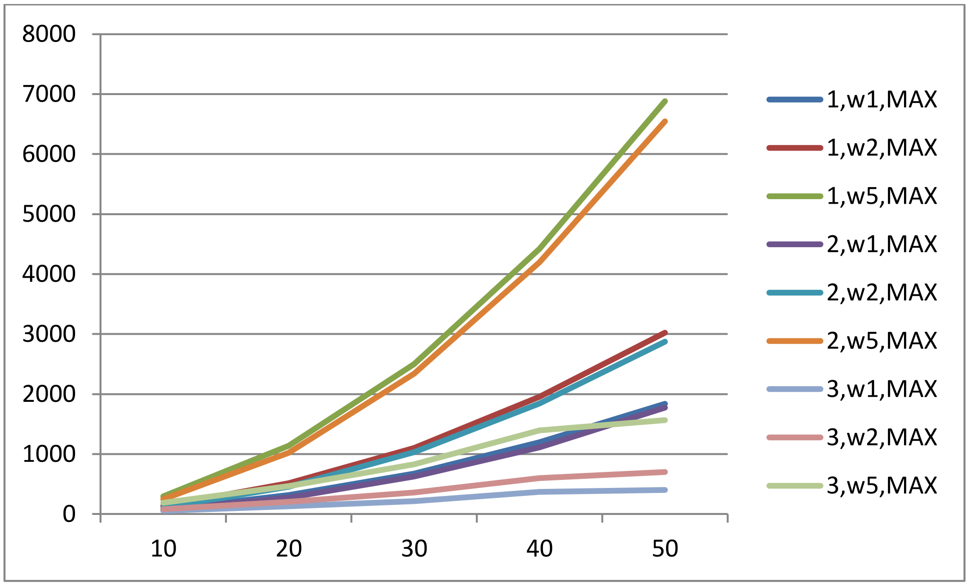
Type of sort, work time, and whether it is the upper, lower, or average bound of its offsets

For example, “2,w5,MAX” denotes using sort_strategy_2(), has a work time of 5, and its graph is the upper bound of its offsets.

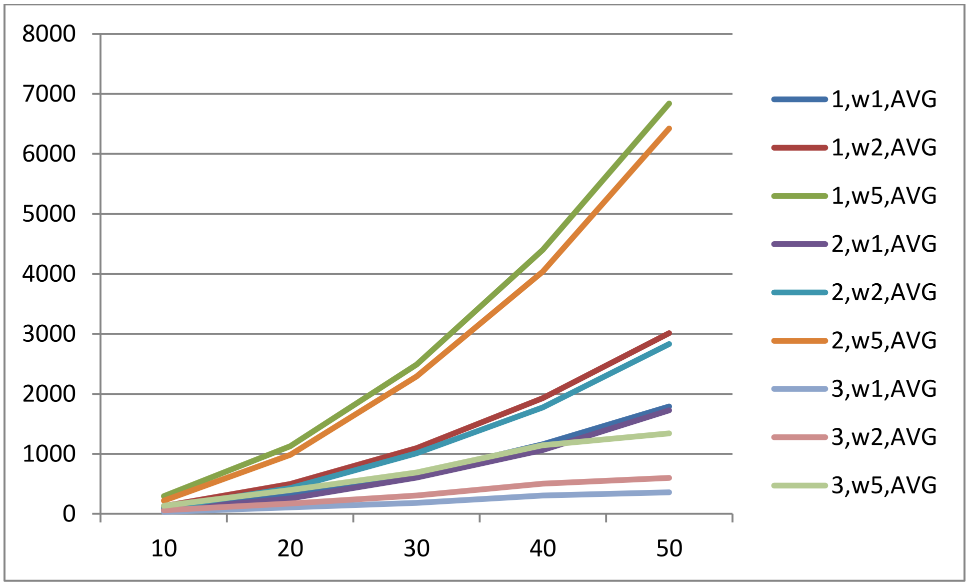
All, Min



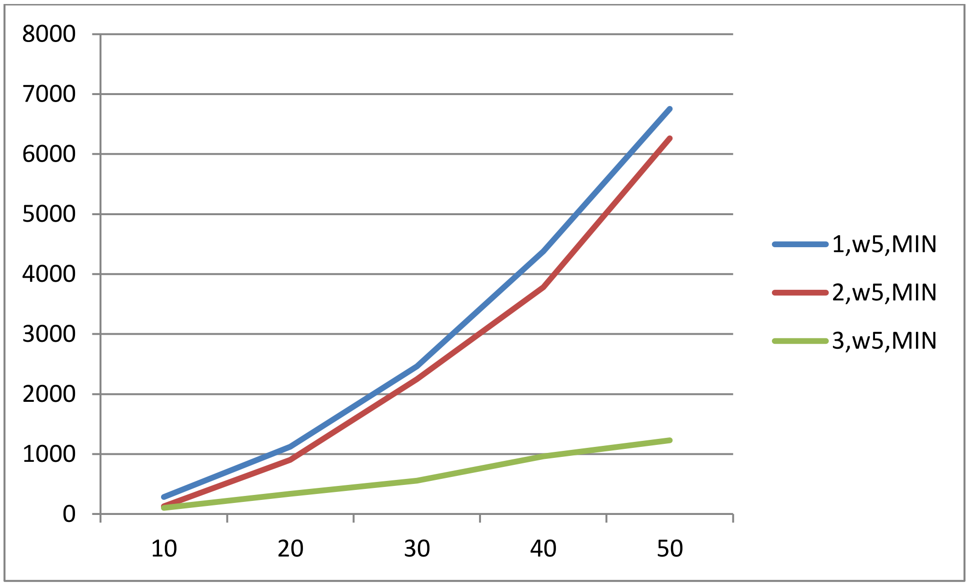
All, Max



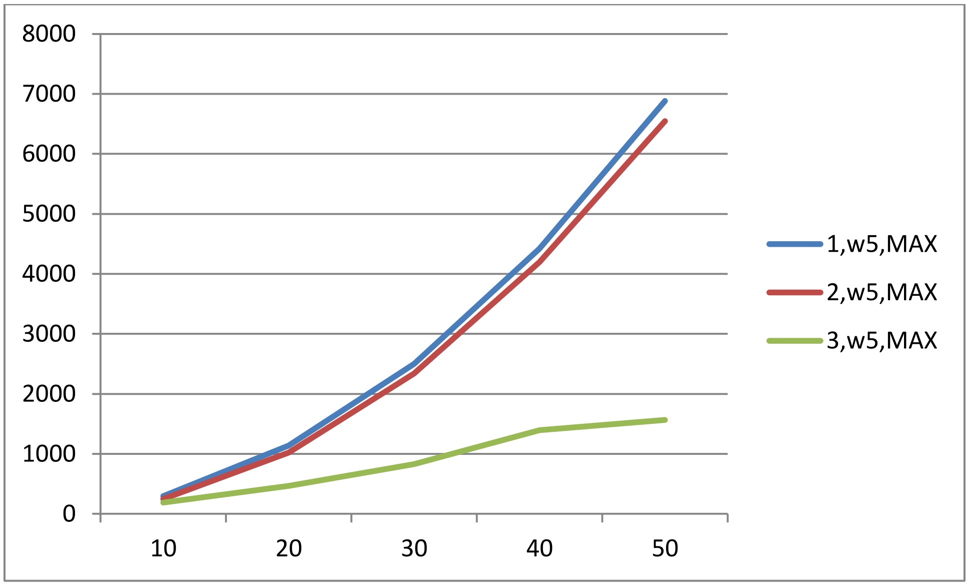
All, Mean



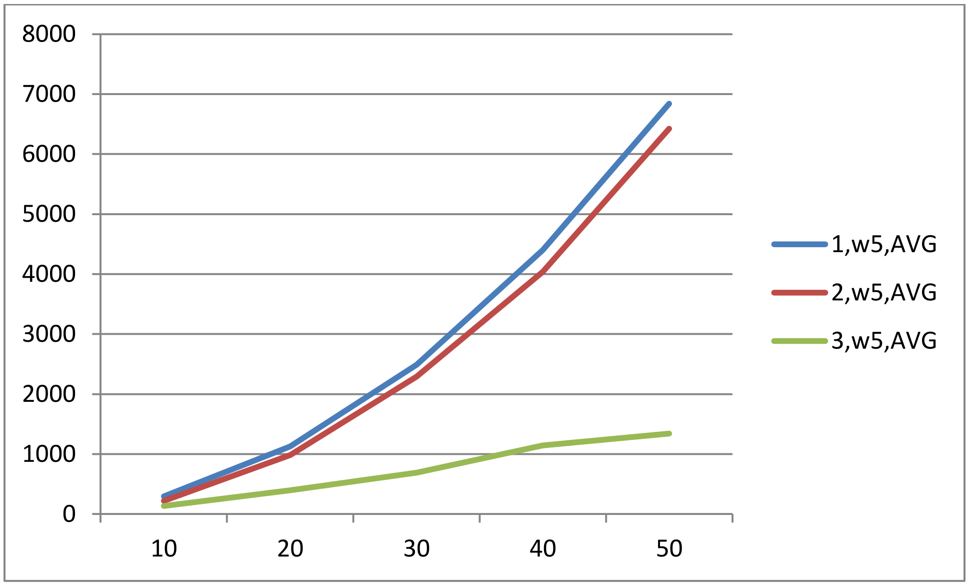
All, Work 5, Min



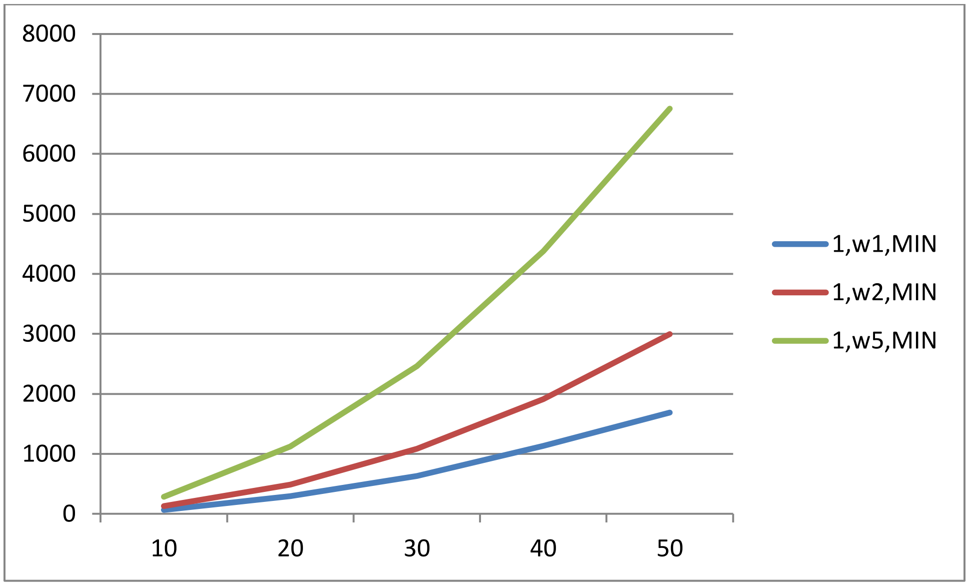
All, Work 5, Max



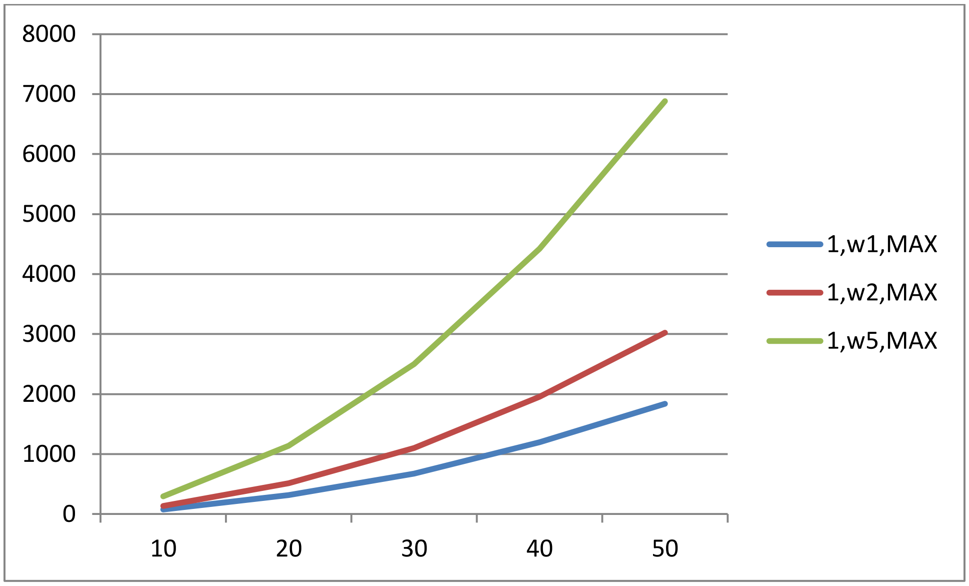
All, Work 5, Mean



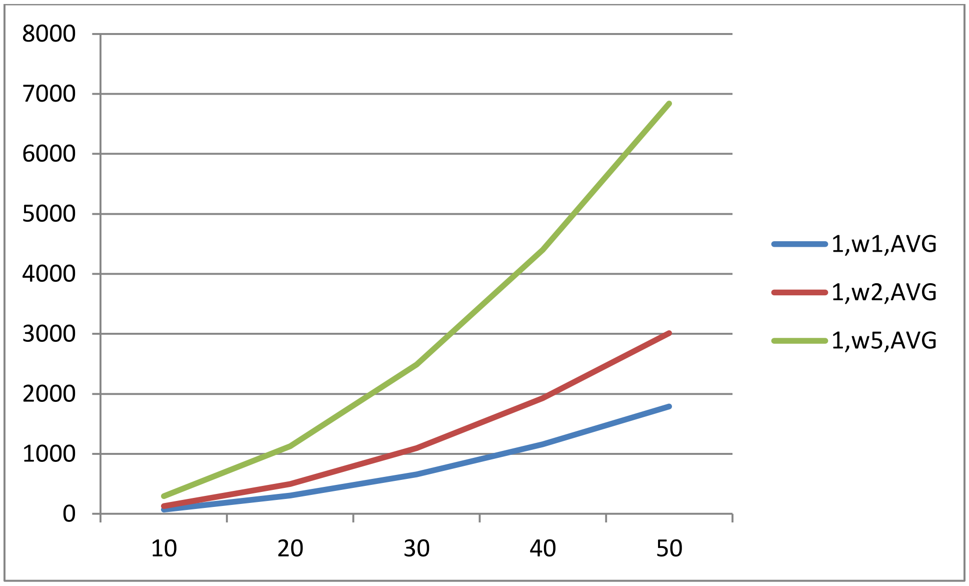
Strategy 1, Work 1, 2, 5, Min



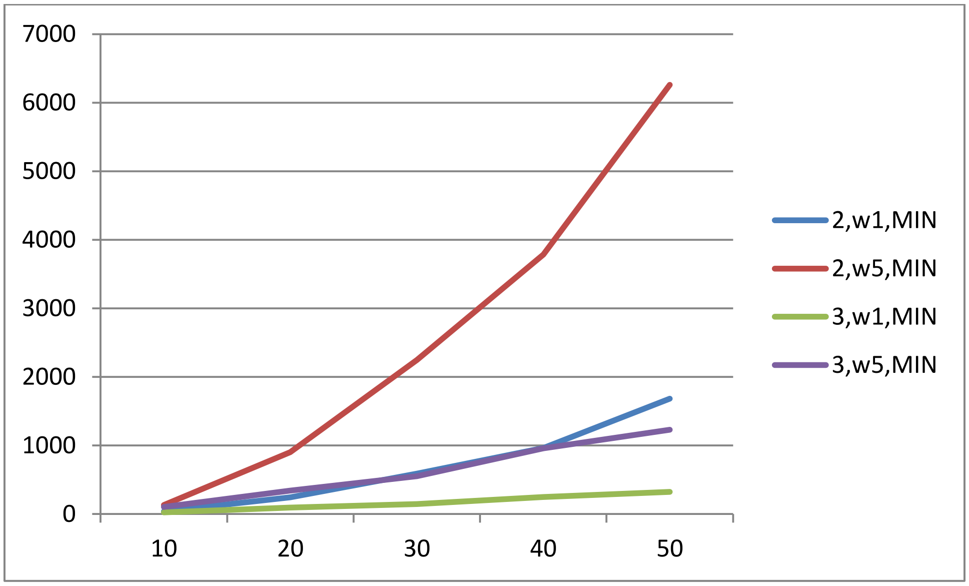
Strategy 1, Work 1, 2, 5, Max



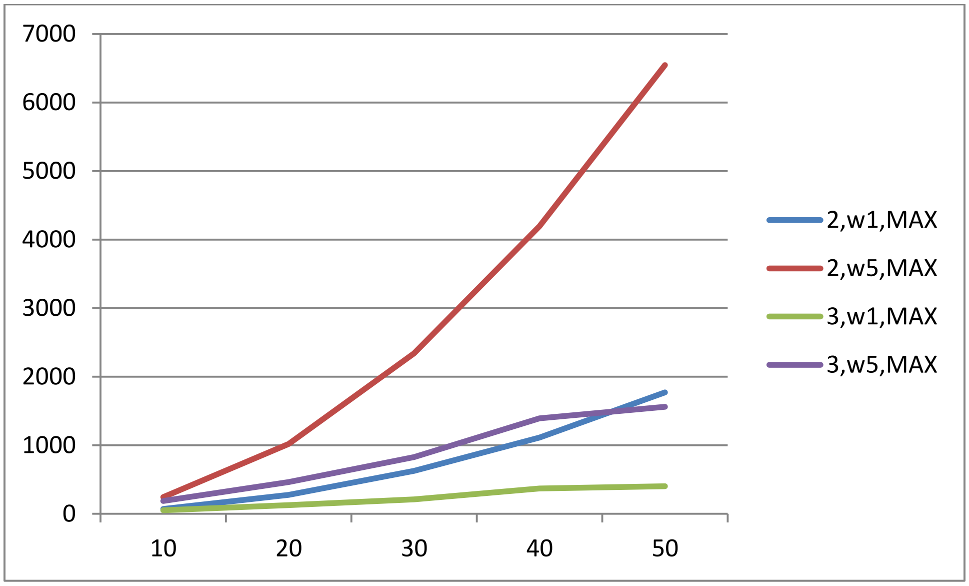
Strategy 1, Work 1, 2, 5, Mean



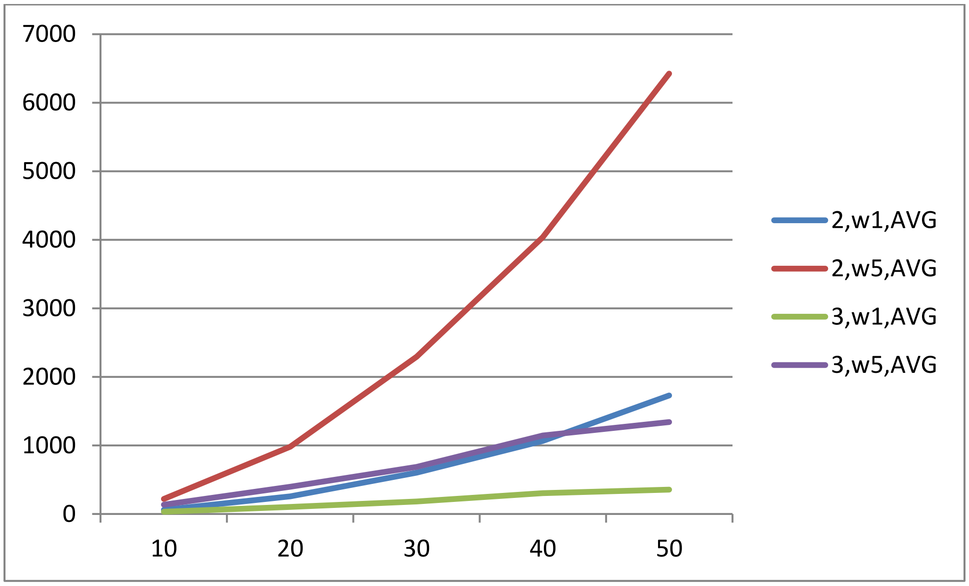
Strategy 3, 2, Work 1, 5, Min



Strategy 3, 2, Work 1, 5, Max



Strategy 3, 2, Work 1, 5, Mean



Q3. Make some Predictions about what value you would be getting for a particular scenario according to your theoretical analysis. (**2pts**)

At a work of 5, I’d expect Strategy 1 and 2 to have a growth rate of c(n^2), and Strategy 3 to have a growth rate of c(nlogn), where c is a positive constant. E.g. at an n of 30, Strategy 1 and 2 will be around 30^2, and Strategy 3 is expected to be around 30log30.

Q4.

My predictions seem to be correct. Sort Strategy 1 and 2 have a similar growth rate, with Strategy 2 slightly outperforming it. Strategy 3 has a different growth rate altogether, being over a 400% faster than the others at an input of size 50. At a work of 5, Strategy 3 has a similar time to Strategy 2 at a work of 1.

At a work of 5, Strategy 1 and 2 have a growth rate of around $2.7n^2$. Strategy 3 seems to have a growth rate of 5.5($n\log_2 n$).

Q5.

Sort Strategy 1 always performed the worst at every input size, with the same growth rate as Strategy 2 but performing with a constant more time. This is because it will always perform both its loops, and performing the operations within those loops without fail at every call.

Strategy 2 will perform the best in the rare case that after bubbling one iteration, its condition will be met and it will break; or if we redundantly wanted to sort an already sorted list. However, generally, it is always more efficient to use Strategy 3, especially as our input size grows arbitrarily large.

It seems like it is rarely a good idea to use selection, insertion, or bubble sort outside of academically learning sorting algorithms.

Part 3 - Analysis of Algorithms

Q1.

We can guess the time complexity by calculating how many operations are performed, which in this case, is the number of times the function is called times the number of operations. Because the number of operations is a constant, we can ignore it for our Big O.

As every time the function is called until we reach our base case, it calls two more of itself, it is within O(2^n).

Q2. Is this the most efficient way to do the task it is accomplishing? (If the task is still a mystery observe the test case of the task to get a clue). (**14 points**)

The inefficiency arises from having to always recursively call to reach the base case.

Instead of always performing recursive calls, we can add the values of our not_really_a_mystery(n) to an array, and reference that array for further calls to save a lot of run time. This would be within O(n), as it would be a matter of looking up the value at each call of the method after we have added them all.

Part 4 - More practice on theory

Q1.

O($\log_2 n$), within O(logn)

As n becomes arbitrarily large, our number of loops remain in a logarithmic growth rate, as “i” doubles every iteration.

Q2.

O(1)

As this algorithm runs without a relationship to an input size, it is always a constant run time, performing a fixed amount of operations.

**edit after clarification

If plimit is n, we have O(n^2)

Q3.

O(n^2)

Our inner loop runs (n – i) times, while our outer loop runs n times. So, multiplying, we get around (n / 2)(n), we have a complexity of O(n^2)

Q4.

O(100n), within <u>O(n)</u>

Our inner loop always runs 100 times, while our outer loop runs n times.