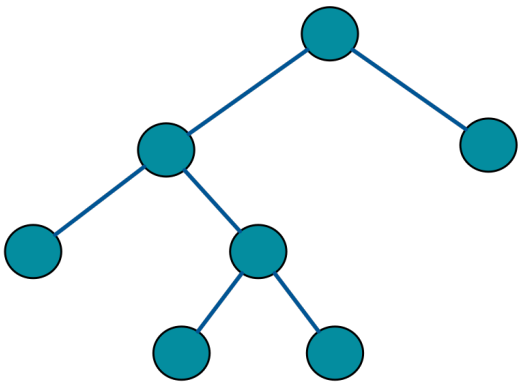
Lab 1



Group 20
Klas Mannberg klaman-8@student.ltu.se
Gustav Segerlind gusseg-8@student.ltu.se
D0012E Algorithms and data structures

LULEÅ UNIVERSITY OF TECHNOLOGY

November 25, 2019

Abstract

Analyzing modified versions of two different algorithms. We are investigating the time complex of a modified insertion and merge sort. After the analysis we proceed by implementing and testing the algorithm in Python. We reflect on our pseudo code and its calculated time complex and see how it matches in practice.

Contents

1 Introduction **1**

2 Theory **1**
2.1 bSort 1
2.2 Modified Mergesort 2

3 Implementation **2**
3.1 Experimentation 2

4 Result **4**

1 Introduction

We divide our project in two parts. In theory we study the algorithms to create a pseudo code representing how they operate. Vi then proceed to analyze our pseudo code and try to compute a time complex for each algorithm.
The implementation following the theory is our project in practice. We use our pseudo code to code the algorithms in Python 3. We then experiment and analyze results following different inputs to judge the conclusion of performance in different areas.

2 Theory

Lets investigate the theoretical part of these two algorithms using pseudo code. Hopefully we can deduct a fair assumption of the running times of the modified versions given any input n. The following segment will go through the unmodified version of insertion sort and binarysearch to then be modified and combined.

2.1 bSort

```
Standard Insertion sort algorithm pseudo code:
for j = 2 to A.length                 c1   n
    key = A[j]                        c2   n-1
    i = j - 1                         c3   n-1
    while i>0 and A[i] > key          c4   sum_{j=2}^{n} t_j
        A[i+1] = A[i]                 c5   sum_{j=2}^{n} (t_j - 1)
        i = i -1                      c6   sum_{j=2}^{n} (t_j - 1)
    A[i+1] = key                      c7   n-1
```

Time complex:
$C(n) = C(1) + \frac{(n-1)n}{2} = \frac{(n-1)n}{2} = \frac{n^2-n}{2}$
$\therefore$ The worst case cost for InsertionSort is $\theta(n^2)$

```
Binary search recursive pseudo code:
BinSearch(A,p,q,X)
    if p > q then return
    if x = A[(p+q)/2]
        return (p+q)/2
    else if x < A[(p+q)/2]
        BinSearch(A,p,(p+q)/2 - 1,x)
    else
        BinSearch(A,(p+q)/2+1,q,x)
```

Time complex:
$C(\frac{n}{2^{(k-1)}}) = C(\frac{n}{2^k}) + 1 \implies C(n) = C(1) + logn$
$\therefore$ The worst case cost for BinarySearch is $\theta(logn)$

So lets create a new pseudo code for insertion sort using binary search

```
for j = 2 to A.length:
    key = A[j]
    i = j -1
    pos = BinSearch(a,0,i,key)   //find what position value should be in. best case is θ(1)
    while i >= pos:              //still have to place it
        A[i+1] = A[i]
        i = i -1
    A[i+1]=key
```

Even though the binary search might only require logarithmic time. The time for the placement would still require a squared amount of time in the worst case scenario. Hence the worst case and average cost for bSort would still be $\theta(n^2)$ but the best case would be $\Omega(n)$ because binary search always gets the best case $\theta(1)$ and we skip the loop. This is the same time comlexity of insertionsort when we look at the bigger picture, but we suspect it will require more operations due to an additional step (search) and same number of loops.

2.2 Modified Mergesort

```
Mergesort(A,k):
    if k < A.length:
        middle = A.length/2
        L = A[0:middle]
        L = Mergesort(L,k-1)
        R = A[middle:]
        R = Mergesort(L,k-1)
        return merge(L,R)
    else:
        A = InsertionSort/bSort(A)
        return A
```

Worst case:
1. Base case - Constant time
2. Dividide into arrays - Constant time
3. Computing (sorting) n^2 times for both we get $\frac{n}{k} * k^2 = n * k$
4. Merging - Linear time n
5. Return - Constant time

The sublists are sorted in k^2 time n times. Our total sorting time would equal $n/k * k^2 = n * k$. The merge time would remain linear and our total time complexity would be $\theta(nlog(\frac{n}{k}) + nk)$.

Best case:
1. Base case - Constant time
2. Dividide into arrays - Constant time
3. Computing (sorting) n^2 times for both we get $\frac{n}{k} * k = n$
4. Merging - Linear time n
5. Return - Constant time

For the best case the sorting time would be $\frac{n}{k} * k$ instead of $\frac{n}{k} * k^2$ since the sorting alogrithms now only takes linear time. The merging would still take $n * log(\frac{n}{k})$ time and in total we instead get $\theta(nlog(\frac{n}{k}) + n)$

3 Implementation

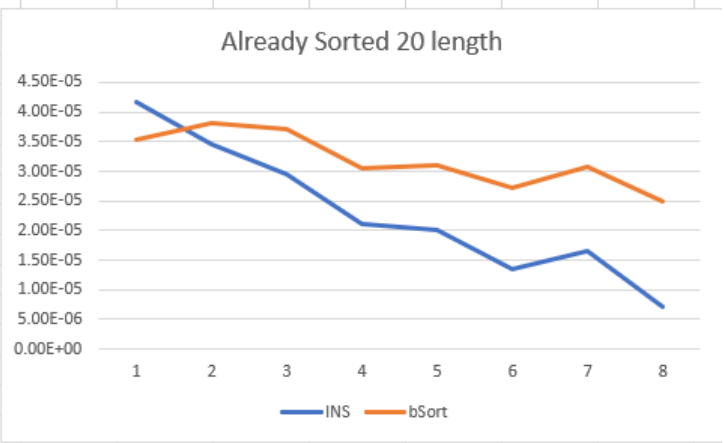
The implementation of the algorithms were created in Python 3. The source code for all algorithms are included in the file lab1.py

3.1 Experimentation

The tests are carried out with set array length and varying k values. The tests were performed on a high end consumer CPU. INS/bSort represents the variant of mergesort. The running time Y is given in approximate seconds. X is the next k value in increments shown on the corresponding data table.

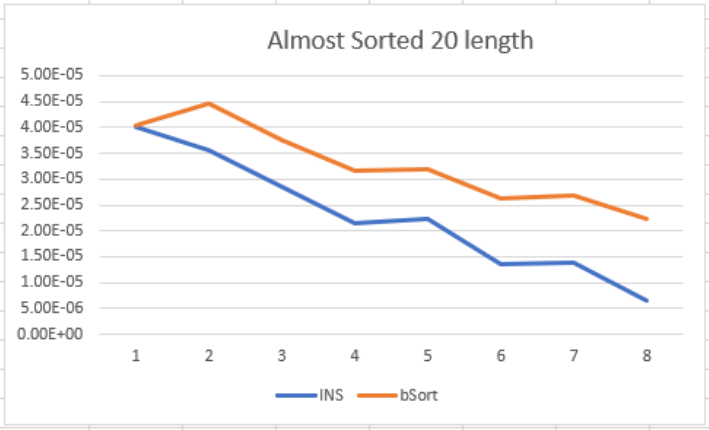
Test 1: Sorted array of length 20
A = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]

k	INS	bSort
0	4.18E-05	3.54E-05
2	3.45E-05	3.82E-05
4	2.94E-05	3.72E-05
6	2.12E-05	3.05E-05
8	2.02E-05	3.09E-05
10	1.35E-05	2.71E-05
16	1.65E-05	3.08E-05
20	7.20E-06	2.50E-05



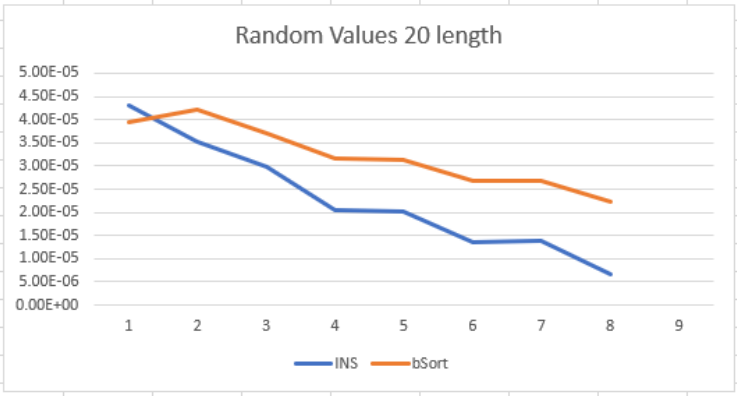
Test 2: Almost sorted array of length 20
A = [1, 2, 18, 4, 5, 6, 7, 8, 9, 20, 11, 12, 13, 14, 15, 16, 17, 3, 19, 10]

k	INS	bSort
0	4.00E-05	4.05E-05
2	3.57E-05	4.46E-05
4	2.84E-05	3.76E-05
6	2.15E-05	3.17E-05
8	2.22E-05	3.20E-05
10	1.36E-05	2.62E-05
16	1.39E-05	2.67E-05
20	6.40E-06	2.24E-05



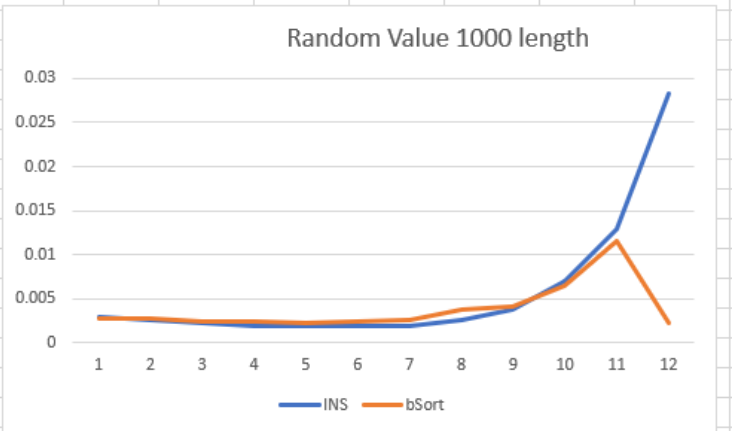
Test 3: Randomly generated array of length 20
A = [2, 2, 0, 8, 17, 15, 19, 13, 12, 17, 5, 17, 2, 14, 1, 10, 14, 7, 19, 9]

k	INS	bSort
0	4.30E-05	3.94E-05
2	3.52E-05	4.21E-05
4	2.97E-05	3.69E-05
6	2.05E-05	3.15E-05
8	2.02E-05	3.12E-05
12	1.37E-05	2.67E-05
16	1.40E-05	2.68E-05
20	6.50E-06	2.23E-05



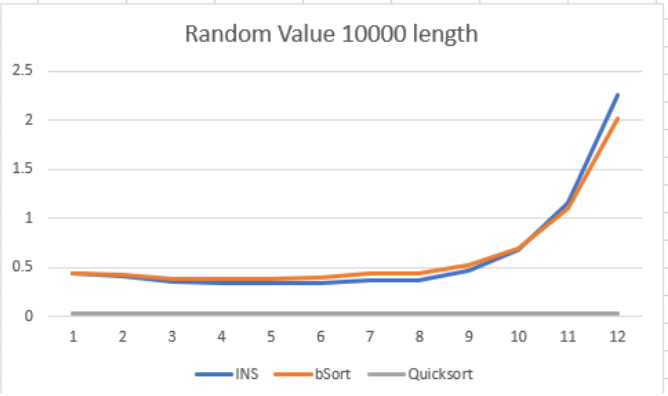
Test 4: Randomly generated array of length 1000
A = every entry a random number from 1..1000, length = 1000

k	INS	bSort
0	0.002895	0.002804
2	0.002607	0.002749
6	0.002247	0.002448
12	0.001885	0.002321
20	0.001926	0.002285
25	0.001917	0.002346
50	0.001913	0.002476
90	0.0025	0.003743
150	0.003795	0.004125
350	0.006988	0.006405
600	0.012853	0.011448
1000	0.028338	0.002155



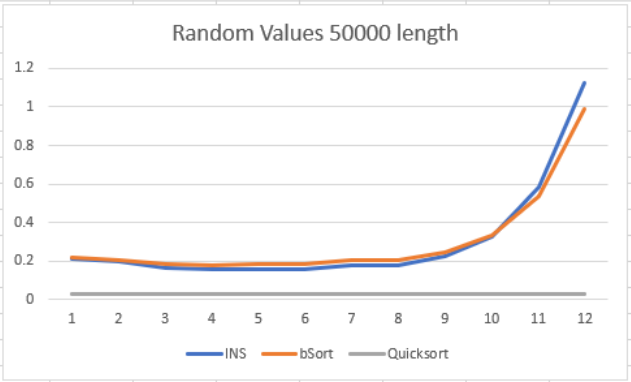
Test 5: Randomly generated array of length 10000

k	INS	bSort	Quicksort
0	0.440073	0.439191	0.0271
2	0.416439	0.427035	0.0271
6	0.350611	0.385573	0.0271
12	0.337926	0.384677	0.0271
20	0.335905	0.382803	0.0271
25	0.338902	0.398561	0.0271
50	0.37212	0.439417	0.0271
90	0.374037	0.433636	0.0271
150	0.465477	0.51805	0.0271
350	0.67148	0.692446	0.0271
600	1.165196	1.102197	0.0271
1000	2.266237	2.013318	0.0271



Test 6: Randomly generated array of length 50000

k	INS	bSort	Quicksort
0	0.21337	0.217478	0.027978
2	0.198602	0.200547	0.027978
6	0.165085	0.180181	0.027978
12	0.157412	0.179954	0.027978
20	0.155678	0.180361	0.027978
25	0.158671	0.184891	0.027978
50	0.173956	0.206541	0.027978
90	0.174062	0.205089	0.027978
150	0.220723	0.245924	0.027978
350	0.325028	0.335514	0.027978
600	0.582128	0.536677	0.027978
1000	1.121903	0.992935	0.027978



4 Result

Both versions of mergesort seems to have worse running times the larger the value of k is. We see on an already sorted array with low k values that bSort seems to be fastest. But in general on smaller array sizes insertionsort seems to beat bSort in almost all other cases. When we use larger arrays insertionsort seems to be fastest on low k values, but eventually bSort catches up once k starts getting high enough and becomes the faster algorithm which is very evident in test 4 especially. So when the sublists become large enough when using big arrays bSort takes the cake, but in almost all other cases insertionsort seems more efficient. Our implementation of quicksort seems to outshine both algorithms no matter the value of k.

Theoretically we suspected insertionsort to be in general faster than bSort altough they have almost the same time complexity, the bSort requires more operations. But there could have been error in our readings a few main causes could be:

1. The CPU does other work while program is running, effecting the reported running time.
2. The time is gathered by pulling the system time after algorithm is finished - when it started. And dosen't give an exact time of the actual running of the code. (Connected to the above point).
3. Our implementation might work but there is a possibility of it not being optimized to its fullest giving worse results then there should be.