**Oblig 2 - Gruppe 18**

**Medlemmer:**

Eirik Innselset Bjørdal

Brage Bakken

**Del 1 – Oppgave 1**

**Del 2 – Oppgave 1**

1. **Når vi flytter det minste elementet til indeks 0 før vi starter insertion sort, kan vi forenkle betingelsen i while-løkken:** **Siden vi har garantert at det minste elementet er på plass ved a[0], trenger vi ikke å sjekke j >= 0. Dette reduserer antall sammenligninger i while-løkken og gir en liten, men merkbar forbedring i hastighet.**

**Resultat: koden sorterar meir effektivt.**

1. **Dette reduserer antall iterasjoner av hovedløkken, og fordi vi fortsatt bruker insertion sort, beholder vi stabiliteten.**
2. package v2;
3. import java.util.Random;
4. import java.util.Arrays;
5. public class Oppgave1 {
6. public static void improvedInsertionSort(int[] a) {
7. int n = a.length;

10. int minIndex = 0;
11. for (int i = 1; i < n; i++) {
12. if (a[i] < a[minIndex]) {
13. minIndex = i;
14. }
15. }
16. *swap*(a, 0, minIndex);
18. for (int i = 2; i < n; i += 2) {
19. int smaller, larger;

22. if (a[i] < a[i - 1]) {
23. smaller = a[i];
24. larger = a[i - 1];
25. } else {
26. smaller = a[i - 1];
27. larger = a[i];
28. }
30. int j = i - 1;
31. while (j > 0 && a[j] > larger) {
32. a[j + 2] = a[j];
33. j--;
34. }
35. a[j + 2] = larger;
36. while (j > 0 && a[j] > smaller) {
37. a[j + 1] = a[j];
38. j--;
39. }
40. a[j + 1] = smaller;
41. }
42. }
43. private static void swap(int[] a, int i, int j) {
44. int temp = a[i];
45. a[i] = a[j];
46. a[j] = temp;
47. }
48. public static void main(String[] args) {
49. int size = 150000;
50. int[] array = new Random().ints(size, 1, 150000).toArray();
51. int[] arrayCopy = Arrays.*copyOf*(array, array.length);
52. long startTime = System.*nanoTime*();
53. *improvedInsertionSort*(arrayCopy);
54. long endTime = System.*nanoTime*();
55. System.***out***.println("Sorteringstid: " + (endTime - startTime) / 1e9 + " sekunder");
56. boolean sorted = *isSorted*(arrayCopy);
57. System.***out***.println("Sortert korrekt: " + sorted);
58. System.***out***.println("Første 20 elementer etter sortering: " + Arrays.*toString*(Arrays.*copyOf*(arrayCopy, 20)));
60. System.***out***.println("Siste 20 elementer etter sortering: " + Arrays.*toString*(Arrays.*copyOfRange*(arrayCopy, arrayCopy.length - 20, arrayCopy.length)));
61. }
62. private static boolean isSorted(int[] a) {
63. for (int i = 1; i < a.length; i++) {
64. if (a[i - 1] > a[i]) {
65. System.***out***.println("Feil sortering ved indeks " + i + ": " + a[i - 1] + " > " + a[i]);
66. return false;
67. }
68. }
69. return true;
70. }
71. }

**Et bilde som inneholder tekst, skjermbilde, programvare, display

KI-generert innhold kan være feil.**

**Del 2 – Oppgave 2**

**a)**

A screenshot of a computer screen

AI-generated content may be incorrect.

Den teoretiske tiden kan avvike fra den målte tiden på grunn av flere faktorer. Dette kan være ting som f. eks. hastigheten på datamaskinen, om det er flere programmer som kjører i bakgrunnen og andre ting som kan senke prosessorhastigheten. Det kan også være forårsaket av uopptimal implementering av sorteringsalgoritmene. Hvordan tabellene er utformet kan også ha en effekt som f.eks. hvis all elementen er like kan quicksort bli tregere enn om de er tilfeldig plasert i tabellen.

**b)**

**A screen shot of a computer program

AI-generated content may be incorrect.**

Når du sorter en liste med bare like elementer med quicksort så kan den bruke lengre tid fordi den leter etter et sted å partisjonere listen. Dette fører til mange unødvendige sammenligninger og lengre kjøretid. I en tilfeldig liste så vil quicksort ha en kjøretid på O(n^2) men i verste fall kan den bli O(n log(n)).