Thomas Strandlie Johannessen
Mathias Gretland Ellingsen
Parkavan Illavalagan

Oppgave 1.)

Vi tok i bruk følgende algoritmer:

Insertion-sort O(n^2)

Radix-sort O(d(N+n))

Selection-sort O(n^2)

Quick-sort (N * log(N)) // nesten alltid, hvis ikke, O(n^2) // Worst case

Vi har kjørt metodene igjennom run_algs_part1 metoden i oblig3runner, som lager en output fil, som er den sorterte versjonen av inputet. Vi vet derfor om algoritmene sorterer som de skal.

Oppgave 2.)

Hvordan er bytter og sammenligninger målt?

Dette er hentet fra prekoden til obligen. Det er en wrapper klasse for alle elementene i listen, som teller hver gang en sammenligning av to elementer i listen blir gjort av en gitt algoritme. Bytter er en metode man kaller på i en egen klasse som teller antall bytter og foretar bytter i listen på gitte indexer.

Thomas Strandlie Johannessen
Mathias Gretland Ellingsen
Parkavan Illavalagan

Oppgave 3.)

Random_10_result:

n	insertion_cmp	insertion_swaps	insertion_time	quick_cmp	quick_swaps	quick_time	selection_cmp	selection_swaps	selection_time	radix_cmp	radix_swaps	radix_time
10	21	13	17	41	15	19	45	7	20	0	0	20

Random_100_result:

n	insertion_cmp	insertion_swaps	insertion_time	quick_cmp	quick_swaps	quick_time	selection_cmp	selection_swaps	selection_time	radix_cmp	radix_swaps	radix_time
100	2664	2569	2511	867	236	344	4950	97	1646	0	0	192

Random_1000_result:

n	insertion_cmp	insertion_swaps	insertion_time	quick_cmp	quick_swaps	quick_time	selection_cmp	selection_swaps	selection_time	radix_cmp	radix_swaps	radix_time
1000				15053	3036	5734				0	0	2520

Nearly_sorted_100:

n	insertion_cmp	insertion_swaps	insertion_time	quick_cmp	quick_swaps	quick_time	selection_cmp	selection_swaps	selection_time	radix_cmp	radix_swaps	radix_time
100	137	38	86	3784	166	1005	4950	26	1691	0	0	196

Nearly_sorted_1000:

n	insertion_cmp	insertion_swaps	insertion_time	quick_cmp	quick_swaps	quick_time	selection_cmp	selection_swaps	selection_time	radix_cmp	radix_swaps	radix_time
1000	1465	467	942	343988	1638	87805				0	0	2479

Thomas Strandlie Johannessen
Mathias Gretland Ellingsen
Parkavan Illavalagan

I hvilken grad stemmer kjøretiden overens med kjøretidsanalysene (store O) for de ulike algoritmene?

Insertion-sort

Når vi kjører tilfeldig input, ser vi at kjøretiden stemmer overens med antall tall som skal sorteres, den kjører ca. like raskt som selection-sort på usortert input, som stemmer overens med at begge har O(n^2) i kjøretid.

Vi ser derimot at dette er den raskeste metoden på nesten sortert input, som stemmer overens med at denne metoden gjør få sammenligninger og bytter på nesten sortert input, og derfor blir veldig rask.

Selection-sort

Når vi kjører tilfeldig input, ser vi at kjøretiden stemmer overens med antall tall som skal sorteres, den kjører noe raskere enn insertion-sort på usortert input, som stemmer overens med at begge har O(n^2) i kjøretid. Loopen vil kjøre og foreta unødvendige sammenligninger ettersom listen er sortert og fører til mer tid ved kjøretiden. Ved "nearly_sorted_1000" og "Random_1000", får vi ikke oppgitt kjøretiden som skyldes av at kjøretiden har O(n^2) i kjøretidskompleksitet.

Dermed vil det ta lang tid for kjøretidskompleksiteten ved disse filene.

Quick-sort

Når vi velger pivot i quick sort, velger vi bare den høyeste indexen, det er nok derfor denne kjører forholdsvis tregt når vi jobber på nesten sortert input.

Radix-sort

Radix sort gjør ingen bytter og sammenligninger, dette kan gjøre at den vil få litt raskere tid sammenlignet med de andrealgoritmene vi kjører her, da den ikke vil gjøre noen ekstra operasjoner for å telle dette.

Thomas Strandlie Johannessen
Mathias Gretland Ellingsen
Parkavan Illavalagan

Hvordan er antall sammenligninger og antall bytter korrelert med kjøretiden?

Insertion-sort

På nesten sortert input gjør ikke denne metoden mange sammenligninger og bytter, fordi der vil den kun gjøre bytter der det er nødvendig, og vi ser derfor at kjøretiden til denne algoritmen stemmer overens med dette på nesten sortert input.

Selection-sort

Ved "nearly_sorted_100" og "nearly_sorted_1000", ser vi at selection-sort foretar mange sammenligninger som stemmer siden algoritmen ikke bryter ut av loopen hvis listen er sortert.

Ved "Random_100-result" og "Random_1000-result", ser vi at selection-sort, foretar mange sammenligninger som stemmer med samme grunn som tidligere.

Quick-sort

Her ser vi at når listen er nesten sortert så vil den utføre utrolig mange sammenligninger, fordi her vil tallet som er til høyre for det tallet man er på, som oftest være mindre. Da utfører den ikke en så stor andel av swaps fordi det ikke er noe å bytte på, men den må ofte gå ut for å starte en ny sammenligning. Hvis tallene er i tilfeldig rekkefølge så slipper man å gjøre så mange sammenligninger, fordi da er ikke tallet til høyre så ofte mindre enn tallet man er på. Men da blir man nødt til å gjøre flere swaps, slik at den blir sortert.

Quick-sort vil dermed bruke lenger tid på en liste som nesten er sortert, enn en liste som er tilfeldig.

Radix-sort

Litt usikker på dette, men radix sort gjør ingen bytter eller sammenligninger, da denne algoritmen bare pusher og poper fra arrays. Derfor ser vi ingen sammenligninger eller bytter i resultatene våre.

Thomas Strandlie Johannessen
Mathias Gretland Ellingsen
Parkavan Illavalagan

Hvilke sorteringsalgoritmer utmerker seg positivt når n er veldig liten?

Og når n er veldig stor?

I våre resultater var det ikke veldig stor forskjell mellom tidene når "n" var veldig liten når vi hadde 10 tilfeldige tall i input, algoritmene med log(N^2) i kjøretidskompleksitet er kanskje litt raskere da de ikke trenger å foreta like mange operasjoner. Når vi begynte med 100 tilfeldige tall, ser vi at det begynner å bli litt større forskjeller, og at algoritmene våre med (n * log(n)) i kjøretidskompleksitet (radix og quick) begynner å bli raskere.

Når vi har 1000 tilfeldig tall i input ser vi at algoritmene våre med log(N^2) i kjøretidskompleksitet bruker for lang tid (da vi har makstid på 1000 millisekunder), og det er derfor bare algoritmene våre med n * log(n) i kjøretidskompleksitet får resulterer med så mange tall i input.

Hvilke sorteringsalgoritmer utmerker seg positivt for de ulike inputfilene?

Insertion-sort

Insertion sort utmerker seg positivt på nesten sortert input, som nevnt tidligere trenger ikke denne algoritmen og foreta seg like mange bytter og sammenligninger når dette er tilfellet.

Selection-sort

Selection sort skiller seg ut, da den bruker ca. like lang tid på å sortere lister som er tilfeldige og lister som er nesten sorterte. Men siden den kjører i kvadratisk tid så vil den bruke utrolig lang tid på å sortere inputfiler med mange elementer, og egner seg derfor ikke for sortering av lister med mange elementer.

Quick-sort

Quick sort gjør det veldig bra ved større input filer som ikke nesten er sortert, som nevnt er nok dette grunnet pivot metoden vår.

Thomas Strandlie Johannessen
Mathias Gretland Ellingsen
Parkavan Illavalagan

Radix-sort

Radix sort gjør det veldig bra ved større input filer, spesielt da hvis det er få siffer i det største tallet.

Har du noen overraskende funn å rapportere?

Radix har ingen sammenligninger og bytter, dette kan kanskje ha noe med hvordan vi teller, men er vel fordi sorteringen blir gjort ved pushing og popping av arrays. La forresten inn denne metoden i countcompares for at radix sorten vår skulle fungere, dette var nødvendig pga måten vi henter siffer i radix sort metoden, ved get_digit():

```
def __floordiv__(self, other):
    return self.elem // other
```