aronbj@uio.no taeriks@uio.no

IN2010 Innleveringsoppgave 2

Deloppgave 3: Eksperimentèr

For denne andre innleveringsoppgaven har vi implementert: Quick Sort, Insertion sort, Merge sort og Bubble sort.

Vi har benyttet oss av den offentlig tilgjengelige prekoden fra emnesiden, med noe modifikasjon til hvordan tid beregnes.

```
# Run a sorting and return a description of the execution as a CSV row
def runalg(alg, A, i, discarded):
    fmt = runstringfmt(alg)

    if alg in discarded:
        res = fmt % (0, 0, 0)
        return res.replace('0', ' ')

    countingA = CountSwaps([CountCompares(x) for x in A[:i]])
    now = time.perf_counter_ns()

    alg(countingA)
    timeus = time.perf_counter_ns() - now

    timems = timeus / 1000000

    if timems > TIME_LIMIT_MS:
        discarded.add(alg)
        print('\nGiving up on ' + algname(alg) + '\n')

    comparisons = sum(x.compares for x in countingA)
    swaps = countingA.swaps

    return fmt % (comparisons, swaps, timems)
```

I den originale prekoden benyttet man time.time(), der tidsberegninger ble utført. Dette har vi endret til time.perf_counter_ns() ettersom det gav uriktige svar tidligere. Det er mulig at den offentlige prekoden på emnesiden har blitt endret i ettertid, men slik var det i alle fall da vi først benyttet den. For å gjøre om fra nanosekunder til millisekunder delte vi resultatet på en million. Foruten om dette benyttet vi metodene fra prekoden for å beregne antall bytter og sammenlikninger i algoritmene. Merk at slik vi tolket bytter beregnet vi ikke bytter i Merge sort algoritmen, da man strengt tatt ikke gjennomfører bytter av ulike elementer som i de andre algoritmene. Derfor står det 0 i kolonnen for «swaps» i Merge sort algoritmens resultater. For at testprogrammet skulle gi opp på enkelte algoritmer med for lang kjøretid beholdt vi grensen på 100 millisekunder satt i prekoden.

aronbj@uio.no taeriks@uio.no

Algoritmene i eksperimentet har store O notasjon på følgende form:

- **Quick Sort:** Verste tilfelle O(n^2), men avhengig av pivot elementet kan det i beste tilfelle være O (n log(n)).
- **Insertion Sort:** Verste tilfelle O(n^2), men dersom alle input er allerede sortert, O(n).
- **Merge Sort:** Verste tilfelle O (n log(n)).
- **Bubble Sort:** Verste tilfelle O(n^2).

Jevnt over vil vi påstå at kjøretiden stemmer overens med kjøretidsanalaysen av de ulike algoritmene, men gitt ulik form på input var det definitivt noen overraskelser som viser anvendeligheten av algoritmene gitt ulike inputs.

For å kunne gjennomføre testene, slik de er gjort nedenfor må følgende del av koden manipuleres

```
# Put the sorting algorithms under test for part 1 here
ALGS1 = [insertion.sort,quick.sort,bubble.sort,merge.sort]
#  quick.sort, bubble.sort,
# Put the sorting algorithms under test for part 2 here
ALGS2 = [insertion.sort,quick.sort,bubble.sort,merge.sort]
```

I innlevering2runner.py vil man kunne endre følgende kodesnutt for å teste ut ulike algoritmer oppimot hverandre. De fleste testene er gjennomført ved at alle algoritmene testes oppimot hverandre og derfor vil man ikke måtte endre på koden. Dette er med unntak av siste testen der man i det ene eksperimentet tester Quick sort oppimot Merge sort, og Merge sort oppimot Insertion sort.

Resultat ved nesten sortert input 10:

n,	insertion_cmp,	insertion_swaps,	insertion_time,	quick_cmp,	quick_swaps,	quick_time,	bubble_cmp,	bubble_swaps,	bubble_time,	merge_cmp,	merge_swaps,	merge_time
0,	0,	0,	0,	0,	0,	0,	0,	0,	0,	0,	0,	0
1,	0,	0,	0,	0,	0,	0,	0,	0,	0,	0,	0,	0
2,	1,	0,	0,	1,	0,	0,	1,	0,	0,	1,	0,	0
3,	2,	0,	0,	5,	0,	0,	3,	0,	0,	2,	0,	0
4,	3,	0,	0,	10,	0,	0,	6,	0,	0,	4,	0,	0
5,	4,	0,	0,	16,	0,	0,	10,	0,	0,	5,	0,	0
6,	6,	1,	0,	14,	1,	0,	15,	1,	0,	8,	0,	0
7,	10,	4,	0,	18,	2,	0,	21,	4,	0,	9,	0,	0
8,	11,	4,	0,	27,	2,	0,	28,	4,	0,	15,	0,	0
9,	12,	4,	0,	37,	2,	0,	36,	4,	0,	16,	0,	0
10,	13,	4,	0,	39,	2,	0,	45,	4,	0,	17,	0,	0

Resultat ved random input 10:

n,	insertion_cmp,	insertion_swaps,	insertion_time,	quick_cmp,	quick_swaps,	quick_time,	bubble_cmp,	bubble_swaps,	bubble_time,	merge_cmp,	merge_swaps,	merge_time
0,	0,	0,	0,	0,	0,	0,	0,	0,	0,	0,	0,	0
1,	0,	0,	0,	0,	0,	0,	0,	0,	0,	0,	0,	0
2,	1,	0,	0,	1,	0,	0,	1,	0,	0,	1,	0,	0
3,	2,	0,	0,	5,	0,	0,	3,	0,	0,	2,	0,	0
4,	4,	1,	0,	7,	1,	0,	6,	1,	0,	4,	0,	0
5,	9,	5,	0,	13,	3,	0,	10,	5,	0,	7,	0,	0
6,	14,	9,	0,	17,	5,	0,	15,	9,	0,	10,	0,	0
7,	16,	10,	0,	24,	6,	0,	21,	10,	0,	13,	0,	0
8,	18,	11,	0,	29,	7,	0,	28,	11,	0,	15,	0,	0
9,	21,	13,	0,	31,	7,	0,	36,	13,	0,	17,	0,	0
10,	22,	13,	0,	47,	7,	0,	45,	13,	0,	22,	0,	0

For lave inputs, uavhengig av om tallene er sortert eller ikke, var algoritmene ganske så likt stilt i fysisk kjøretid. Når såpass få elementer skal sorteres er utslaget lite i den fysiske kjøretiden, men effektiviteten av algoritmene kan man klart se ut ifra antallet bytter og sammenlikninger som ble gjennomført. Bubble sort skiller seg klart ut som den minst effektive algoritmen for både sorterte og

tilfeldige inputs. Dette stemmer overens med kjøretidsanalysen der beste, gjennomsnittlige og verste kjøretid for algoritmen er på formen O(n^2).

For sammenligningens skyld har vi tatt med bilde av de 10 siste resultatene i eksperimentene nedenfor for å lettere kunne sammenligne ved større input:

Resultat ved nesten sortert input 100:

n,	insertion_cmp,	insertion_swaps,	insertion_time,	quick_cmp,	quick_swaps,	quick_time,	bubble_cmp,	bubble_swaps,	bubble_time,	merge_cmp,	merge_swaps,	merge_time
88,	121,	34,	0,	3061,	22,	0,	3828,	34,	0,	290,	0,	0
89,	122,	34,	0,	3151,	22,	0,	3916,	34,	0,	292,	0,	0
90,	124,	35,	0,	3153,	23,	0,	4005,	35,	0,	298,	0,	0
91,	125,	35,	0,	3245,	23,	0,	4095,	35,	0,	300,	0,	0
92,	126,	35,	0,	3338,	23,	0,	4186,	35,	0,	308,	0,	0
93,	128,	36,	0,	3340,	24,	0,	4278,	36,	0,	312,	0,	0
94,	129,	36,	0,	3435,	24,	0,	4371,	36,	1,	320,	0,	0
95,	130,	36,	0,	3531,	24,	0,	4465,	36,	1,	319,	0,	0
96,	131,	36,	0,	3628,	24,	0,	4560,	36,	1,	329,	0,	0
97,	132,	36,	0,	3726,	24,	0,	4656,	36,	1,	331,	0,	0
98,	134,	37,	0,	3728,	25,	0,	4753,	37,	1,	340,	0,	0
99,	136,	38,	0,	3829,	26,	0,	4851,	38,	1,	343,	0,	0
100,	137,	38,	0,	3930,	26,	0,	4950,	38,	1,	337,	0,	0

Resultat ved tilfeldig input 100:

n,	insertion_cmp,	insertion_swaps,	insertion_time,	quick_cmp,	quick_swaps,	quick_time,	bubble_cmp,	bubble_swaps,	bubble_time,	merge_cmp,	merge_swaps,	merge_time
90,	2189,	2100,	1,	848,	133,	0,	4005,	2100,	1,	457,	0,	0
91,	2202,	2112,	1,	854,	137,	0,	4095,	2112,	1,	470,	0,	0
92,	2216,	2125,	1,	858,	140,	0,	4186,	2125,	1,	490,	0,	0
93,	2259,	2167,	1,	797,	144,	0,	4278,	2167,	1,	496,	0,	0
94,	2299,	2206,	1,	826,	150,	0,	4371,	2206,	1,	507,	0,	0
95,	2307,	2213,	1,	883,	140,	0,	4465,	2213,	1,	507,	0,	0
96,	2383,	2288,	1,	895,	140,	0,	4560,	2288,	1,	514,	0,	0
97,	2425,	2329,	1,	870,	150,	0,	4656,	2329,	1,	520,	0,	0
98,	2521,	2424,	1,	1068,	149,	0,	4753,	2424,	1,	533,	0,	0
99,	2606,	2508,	1,	926,	150,	0,	4851,	2508,	1,	533,	0,	0
100,	2668,	2569,	1,	938,	147,	0,	4950,	2569,	1,	549,	0,	0

For input 100 er det fortsatt relativt få elementer man sorterer, og man kan tyde lite fra den fysiske kjøretiden. I henhold til antallet bytter og sammenlikninger er det derimot interessante resultater.

Det er påfallende forskjell for Insertion sort ved sorterte input og tilfeldige input. Algoritmen er svært effektiv når input er tilnærmet ferdig sortert. Dette er på grunn av formen på algoritmen, der få operasjoner må gjennomføres dersom input allerede er sortert. Dersom det er få elementer som ikke ligger på riktig plass i arrayet vårt, så gjennomføres det hverken mange sammenlikninger eller bytter.

Bubble sort er lite effektiv for sorterte og ikke sorterte mengder, men hovedforskjellen er egentlig bare antallet bytter som må gjennomføres. Algoritmen vil uavhengig av antallet bytter sjekke like mange elementer.

Resultatene for Quick sort er svært interessante ettersom resultatene av de ulike formene for input er svært avhengig av hva som er pivot punktet. For at Quick sort algoritmen skal ha optimal kjøretid på O(n log(n)) så må valget av pivot punkt være den verdien som ville utgjort median verdien for alle elementene vi sorterer. Pivot punktet for vår algoritme er ikke valgt på bakgrunn av en slik median verdi, men man kan se for seg at dersom man valgte elementet som var midt iblant alle de sorterte elementene i input, ville man ha mulighet for å treffe ganske godt for å nå O(n log(n). Dette er ikke tilfellet i vår implementasjon, og gitt at den velger innputtet som er ved starten av lista vår som pivot, så betyr det at man må gjennomføre svært mange sammenlikninger selv om man har en nesten

sortert liste. Pivot punktet ser ut til å ha truffet bedre for den tilfeldige input filen og forskjellen i antallet sammenlikninger mellom de to inputfilene støtter opp om denne teorien.

Resultat ved nesten sortert input 1000:

n,	insertion_cmp,	insertion_swaps,	insertion_time,	quick_cmp,	quick_swaps,	quick_time,	bubble_cmp,	bubble_swaps,	bubble_time,	merge_cmp,	merge_swaps,	merge_time
990,	1452,	463,	0,	338377,	275,	47,	,	,	,	5106,	0,	1
991,	1453,	463,	0,	339369,	275,	46,	,	,	,	5102,	0,	1
992,	1454,	463,	0,	340362,	275,	47,	,	,	,	5121,	0,	1
993,	1456,	464,	0,	340364,	276,	47,	,	,	,	5122,	0,	1
994,	1457,	464,	0,	341359,	276,	47,	,	,	,	5136,	0,	1
995,	1458,	464,	0,	342355,	276,	47,	,	,	,	5145,	0,	1
996,	1459,	464,	0,	343352,	276,	48,	,	,	,	5157,	0,	1
997,	1460,	464,	0,	344350,	276,	48,	,	,	,	5159,	0,	1
998,	1461,	464,	0,	345349,	276,	48,	,	,	,	5159,	0,	1
999,	1464,	466,	0,	344354,	278,	48,	,	,	,	5170,	0,	1
1000,	1466,	467,	0,	345356,	279,	48,	,	,	,	5169,	0,	1

Resultat ved tilfeldig input 1000:

n,	insertion_cmp,	insertion_swaps,	insertion_time,	quick_cmp,	quick_swaps,	quick_time,	bubble_cmp,	bubble_swaps,	bubble_time,	merge_cmp,	merge_swaps,	merge_time
990,	,	,	,	15118,	2216,	3,	,	,	,	8597,	0,	2
991,	,	,	,	14624,	2186,	3,	,	,	,	8601,	0,	2
992,	,	,	,	15255,	2200,	3,	,	,	,	8615,	0,	2
993,	,	,	,	15861,	2163,	3,	,	,	,	8622,	0,	2
994,	,	,	,	15556,	2177,	3,	,	,	,	8643,	0,	2
995,	,	,	,	15221,	2151,	3,	,	,	,	8649,	0,	2
996,	,	,	,	15537,	2183,	4,	,	,	,	8656,	0,	2
997,	,	,	,	15073,	2205,	3,	,	,	,	8663,	0,	2
998,	,	,	,	15867,	2187,	4,	,	,	,	8671,	0,	2
999,	,	,	,	15962,	2205,	3,	,	,	,	8680,	0,	2
1000,	,	,	,	15734,	2177,	3,	,	,	,	8693,	0,	2

For input 1000 har vi fått enda større utslag i fysisk kjøretid.

Som nevnt tidligere er det svært overraskende hvor effektivt det er å bruke Insertion sort når elementene er nesten sortert. Det kan se ut som at vi her oppnår en kjøretid som nærmer seg lineær tid, altså den beste mulige kjøretiden for Insertion sort algoritmen.

Resultat ved tilfeldig input 1000 viser at kjøretiden for Insertion algoritmen overskred 100 millisekunder ved gjennomkjøring 749.

749,	137125,	136377,	102,
750,	,	,	,

Dette er i motsetning til resultatet ved sortering for nesten sortert input, der antallet bytter og sammenlikninger er svært lavt relativt sett. Det virker å være den mest effektive algoritmen når vi har å gjøre med nesten sorterte input dersom vi sammenligner det totale antallet sammenlikninger og bytter for algoritmene.

aronbj@uio.no taeriks@uio.no

Det samme resultatet som tidligere gjentar seg for Quick sort algoritmen. Gitt vår implementasjon så har pivot punktet truffet dårlig for nesten sortert input, men bedre for tilfeldig input.

Bubble sort gav opp først iblant algoritmene i eksperimentet, slik man ville anta ut ifra kjøretidskompleksiteten til algoritmen.

Merge sort viser seg å jevnt over være den algoritmen med størst allsidighet og har en mer stabil kjøretid. Tilfeldige inputs tar noe lengre tid enn nesten sorterte, men det er fordi flere operasjoner må gjennomføres for å oppnå riktig rekkefølge når det ikke allerede er sortert.

Nedenfor har vi kjørt en siste test for å sammenlikne de mest effektive algoritmene gitt tilfeldige input og nesten sorterte input.

Resultat ved tilfeldig input 10000:

n,	quick_cmp,	quick_swaps,	quick_time,	merge_cmp,	merge_swaps,	merge_time
9990,	222954,	29290,	50,	120403,	0,	30
9991,	217234,	29790,	49,	120516,	0,	30
9992,	210873,	29816,	48,	120405,	0,	29
9993,	219832,	29588,	50,	120416,	0,	31
9994,	214962,	29745,	49,	120415,	0,	30
9995,	213129,	29705,	49,	120397,	0,	30
9996,	231720,	29389,	52,	120394,	0,	30
9997,	218309,	29589,	49,	120421,	0,	30
9998,	215660,	29622,	49,	120415,	0,	30
9999,	219643,	29428,	54,	120425,	0,	30
10000,	212350,	29954,	49,	120488,	0,	30

Resultat av nesten sortert 10000:

n,	insertion_cmp,	insertion_swaps,	insertion_time,	merge_cmp,	merge_swaps,	merge_time
9990,	14509,	4520,	5,	67331,	0,	21
9991,	14510,	4520,	4,	67379,	0,	21
9992,	14511,	4520,	4,	67346,	0,	21
9993,	14512,	4520,	6,	67354,	0,	27
9994,	14513,	4520,	5,	67373,	0,	23
9995,	14514,	4520,	4,	67351,	0,	20
9996,	14515,	4520,	4,	67360,	0,	21
9997,	14516,	4520,	7,	67379,	0,	31
9998,	14519,	4522,	8,	67394,	0,	20
9999,	14521,	4523,	5,	67427,	0,	21
10000,	14522,	4523,	4,	67389,	0,	21

Disse testene ble kun utført av interesse og for å videre sammenlikne de algoritmene som presterte best for ulik input, gitt vår implementasjon. Her kan vi tyde, gitt vår implementasjon av algoritmene, at Insertion sort er den mest effektive algoritmen for nesten sorterte arrays og at Merge sort er mest effektiv for tilfeldige arrays.

aronbj@uio.no taeriks@uio.no

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

Vi vil påstå at kjøretidsanalysen for de ulike algoritmene, gitt vår implementasjon, stemmer godt overens med antatt kjøretid, men at formen på inputfilene og våre implementasjoner av algoritmene gir betydelige utslag på kjøretiden. Dette påvirker om algoritmene havner i verste, gjennomsnittlige eller beste mulige kjøretid. Dette var tilfelle for Insertion sort algoritmen og Quick sort, der formen på inputfil, samt vår spesifikke implementasjon av algoritmene påvirket kjøretiden betraktelig.

• Hvordan er antall sammenligninger og antall bytter korrelert med kjøre- tiden?

Det ser ut til å være en korrelasjon der et høyere antall sammenlikninger og bytter bidrar til økt kjøretid. Det er vanskelig å se konkret på størrelsesforholdet de imellom og hvordan de separat bidrar til kjøretid, men man tyde anta at enhver bytte operasjon vil påvirke kjøretiden mer, enn hva en sammenlikning vil gjøre. Dette er fordi man her utfører flere operasjoner relativt sett for å bytte to elementær, enn ved å aksessere og sammenlikne.

• Hvilke sorteringsalgoritmer utmerker seg positivt når n er veldig liten? Og når n er veldig stor?

Vi vil påstå at Insertion sort utmerker seg positivt når n er veldig liten, og Merge sort når n er veldig stor. Merge sort har jevnt over kjøretidsanalyse på formen O(n log(n)) og vil prestere slik uavhengig av inputformen og størrelse, men merk at den er bedre egnet for store datasett enn små, der den gjør noe unødig arbeid oppimot f.eks insertion sort, ved å sammenligne hver verdi i arrayet. Insertion sort presterer godt med små datamengder eller når input er nesten sortert fordi den automatisk vil gå forbi allerede sorterte verdier og utfører færre operasjoner relativt sett.

• Hvilke sorteringsalgoritmer utmerker seg positivt for de ulike inputfile- ne?

Dette har vi bemerket tidligere, men vi vil påstå at for tilfeldige inputfiler presterte Quick sort overraskende godt, men kjøretiden vil avhenge svært mye av hva man har valgt som pivot punkt. Det ville ha kunnet være tilnærmet like god prestasjon for nesten sorterte inputfiler dersom pivot punktet hadde vært valgt midt i arrayet, og at dette elementet tilfeldigvis ikke var det elementet som var malplassert. Merge sort presterte jevnt over godt for både sorterte og ikke sorterte inputs. Bubble sort var generelt den minst effektive algoritmen iblant de. Ellers presterte Insertion sort ekstremt bra for nesten sorterte inputfiler, uavhengig av størrelse.

• Har du noen overraskende funn å rapportere?

Vi vil ikke fremme noe annet enn hva som har blitt diskutert tidligere i oppgaven. Det var særlig overraskende å se hvor effektiv Insertion sort faktisk var, gitt at inputfilen allerede var nesten sortert. Det var også overrasende å se på hvor stor forskjell det har med kjøretid dersom valget av pivot punkt i en Quick sort algoritme treffer dårlig.