RAPPORT OBLIG 2 IN2010 H19

Tore Flobergseter

Utplukkssortering

Algoritme

```
public void sorter(int[] tall){
        skrivTall(tall);
        int i,j,k,bytt;

        for(i=0; i<tall.length-1; i++){
        skrivTall(tall);
        k=i;
        for(j=i+1; j<tall.length; j++){
        //System.out.println("sammenligner " + tall[k] + " med " + tall[j]);
        if(tall[j] < tall[k]){
        k=j;
        }
        }
        if(i != k){
        //System.out.println("Bytta " + tall[i] + " med " + tall[k]);
        bytt = tall[k];
        tall[k] = tall[i];
        tall[i] = bytt;
        }
        else {
        //System.out.println("Fant ikke noe mindre elemnet enn " + tall[i]);
        }
        }
        skrivTall(tall);
}
```

Tekst

I denne algoritmen er det en for løkke som går gjennom alle elementene fra start til slutt. Inni for løkka er det en ny for løkke som går gjennom de elementene som enda ikke er sortert og leter etter det minste elementet i denne sekvensen. Hvis elementet som det sammenlignes med er det minste, skjer det ingenting og vi går bare videre i den ytterste for løkka. Dersom et annet element er minst må de bytte plass. Sekvensen blir sortert en og en indeks videre for hver iterasjon i for løkken til vi har sammenlignet alle tall.

Denne var ganske rett frem å implementere, måtte leke litt med verdiene i ettersom at når jeg gikk mer direkte etter det som står i boka så ble ikke det første elementet sortert.

Mønstre

Tilfeldig: sammenligner et og et element og sjekker om det finnes noe mindre element i den usorterte delen av arrayen. Dersom det finnes bytter de plass.
Sortert: sammenligner et og et element, men ingen elementer i den usorterte delen er mindre enn elementet det sammenlignes med. Det skjer ingen bytter.
Synkende: sammenligner et og et element, finner det minste elementet og bytter plass for hver iterasjon.

Innstikksortering

Algoritme

```
public void sorter(int[] tall){
        //skrivTall(tall);
        int i,j,k;
        for(i=1; i<tall.length; i++){
        //skrivTall(tall);
        j = tall[i];
        k=i-1;
        System.out.println();
        while(k>=0 && j<=tall[k]){
        //System.out.println("Bytter " + tall[k] + " med " + tall[k+1]);
        //skrivTall(tall);
        tall[k+1] = tall[k];
        k--;
        }
        tall[k+1] = j;
        }
        //skrivTall(tall);
 }
```

Tekst

Går gjennom arrayen n ganger. Starter å sammenligne element på index 0 og element på index 1, deretter sammenligner den 0, 1 og 2, så 0,1,2,3 osv helt til for-løkka har gått gjennom hele arrayen.

Denne var ganske fin å implementere, måtte lete litt etter inspirasjon online for å implementere denne da verdiene som sto i boka ikke fungerte når jeg implementerte rett fra det som sto i pseudokoden. Starter sammenligningen fra venstre mot høyre.

Mønstre

Tilfeldig: Går gjennom arrayen n ganger, sjekker på samme måte som beskrevet og dersom et element til venstre er større enn det til høyre bytter de plass. Sammenligner 0,1 så 0,1,2 så 0,1,2,3 osv.
Stigende: Går gjennom arrayen n ganger, men ingen elementer bytter plass fordi alle elementene til venstre er mindre enn elementet til høyre for seg.
Synkende: Går gjennom arrayen n ganger, alle elementer bytter plass fordi alle elementene til venstre er større enn elementet til høyre for seg.

Heap-sortering

Algoritme

```
public void sorter(int[] tall){
        //skrivTall(tall);
        if(tall.length<2){
        return;
        }
        int i,bytt;

        for(i=tall.length/2; i>=0; i--){
        heapify(tall, i, tall.length);
        }
        //System.out.println("\nFerdig med forste for-lokke");
        for(i=tall.length-1; i>=0; i--){
        //System.out.println();
        //skrivTall(tall);
        //System.out.println("Bytter " + tall[0] + " med " + tall[i]);
        bytt = tall[0];
        tall[0] = tall[i];
        tall[i] = bytt;
        //skrivTall(tall);
        heapify(tall, 0, i);
        }
        //skrivTall(tall);
 }

  private void heapify(int[] tall, int i, int n){
        //System.out.println("\nGjenoppretter heap-egenskaper - HEAP START");
        //skrivTall(tall);
```

```
        int rot = i;
        int venstre = i*2;
        int hoyre = i*2+1;
        int bytt;
        //System.out.println("Rot: " + rot + "\nV: " + venstre + "\nH: " + hoyre);

        if(venstre < n && tall[venstre] > tall[rot]){
        //System.out.println("tall[V] er storre en tall[ROT] " + tall[venstre] + " " + tall[rot]);
        rot = venstre;
        }
        if(hoyre < n && tall[hoyre] > tall[rot]){
        //System.out.println("tall[H] er storre enn tall[ROT] " + tall[hoyre] + " " + tall[rot]);
        rot = hoyre;
        }
        if(rot != i){
        bytt = tall[i];
        tall[i] = tall[rot];
        tall[rot] = bytt;
        heapify(tall, rot, n);
        }
        //skrivTall(tall);
        //System.out.println("HEAP END");
 }
```

Tekst

Denne var litt mer komplisert å implementere enn innstikksortering og utplukkssortering men når jeg så på forelesningene på nytt av hva som ble forelest så var den ganske enkel å implementere.
Sorter metoden sjekker først lengden til arrayen, dersom den er 1 eller 0 er den allerede ferdig sortert og bare returnerer.
Først går den gjennom arrayen og sjekker om den er på heap-format, hvis ikke retter den opp dette så arrayen oppfyller kravene til en heap.
For å sjekke heap-egenskaper bruker jeg metoden heapify som tar inn en array, og to tall, I og N. Deretter finner jeg venstre og høyre barn til i, sjekker om det venstre er mindre enn N og om tall[V] er større enn tall[ROT]. Dersom tallet på V er større enn tall på ROT blir roten satt til det venstre barnet. Deretter sjekker den om høyre er mindre enn N og om det høyre barnet er større enn roten og bytter hvis det er tilfelle.

For eksempel kan vi se på en array med tre elementer (2,1,0).
FOR-løkke nr 1:

1. Første gangen heapify blir kalt blir den kalt med i = tall.length/2, som i dette tilfellet er 3/2=1 altså er roten 1 i første kall på denne arrayen. Venstre barn blir satt til 1*2 = 2 og høyre barn blir 1*2+1 = 3. venstre er mindre enn n, tall[venstre] = 1 og tall[rot] = 2 og det blir ikke foretatt noe bytte med venstre barn. Tall[hoyre] = 0 og tall[rot] = 2, det blir ikke foretatt noe bytte. Heap-egenskapene er riktige.

2. I det andre kallet på heapify også fra den første for-løkka er i 0, venstre barn blir 0 og høyre barn blir 1. Tall[0] er det samme som tall[rot], og det blir ikke noe bytte av rot. Tall[1] = 1 er mindre enn tall[rot] = 2. Det blir ikke noe bytte av rot.

Nå har vi forsikret oss om at arrayen oppfyller maks-heap egenskaper.
FOR-løkke nr 2:

1. Vi går gjennom hele arrayen fra maks-indeks til min indeks og bytter plass på tall[0] og tall[i]. I første runde blir I=3-1=2. Vi bytter plass på tall[0] og tall[2] og vi får arrayen 012. Vi sjekker heap-egenskapene med N=2 og ROT = 0. V=0, H=1, begge er mindre enn N. Tall[0] er det samme som tall[0], ingen bytter. Tall[1] er større enn tall[0] og vi bytter og får arrayen 102.
2. Vi sjekker heap-egenskapene. ROT = 1, N=2, V=2, H=3. hverken V eller H er mindre enn N og vi går returnerer ut fra HEAP med array 102.
3. Så går vi videre i FOR-løkka og reduserer I med 1 og får I=1. Vi bytter tall(0) og tall(1) og får arrayen 012. Vi sjekker heap-egenskapene med ROT=0, N=1, V=0, H=1. V er mindre enn N, men tall[0] er det samme som tall[0] og ingen bytter. H er ikke mindre enn N.
4. Så går vi videre i for løkka med arrayen 012. Vi får I=0 og bytter tall[0] med tall[0] og sjekker heap-egenskapene en siste runde. Samme som sist og ingen bytter.

Ser ingen spesielle mønstre eller hva dere tenker på ved tilfeldig og sorterte arrayer.

Flettesortering

Algoritme

```
public void sorter(int[] tall){
        //skrivTall(tall);
        if(tall.length<2){
        return;
        }
        fletteSort(tall, tall.length);
 }

  private void fletteSort(int[] tall, int n){
        //skrivTall(tall);
        if(n<2){
        return;
        }
        int[] venstre = new int[(n+1)/2];
        int[] hoyre = new int[n-venstre.length];

        for(int i=0; i<n; i++){
        if(i<venstre.length){
        venstre[i] = tall[i];
        }
```

```
        else {
        hoyre[i-venstre.length] = tall[i];
        }
        }
        fletteSort(venstre, venstre.length);
        fletteSort(hoyre, hoyre.length);
        merge(venstre, hoyre, tall);
        //skrivTall(tall);
}

private void merge(int[] s1, int[] s2, int[] tall){
        /*
        System.out.print("\nS1: ");
        skrivTall(s1);
        System.out.print("S2: ");
        skrivTall(s2);
        System.out.print("Tall: ");
        skrivTall(tall);
        */
        int i=0;
        int j=0;
        int k=0;
        while(i<s1.length && j<s2.length){
        if(s1[i] <= s2[j]) {
        tall[k] = s1[i];
        i++;
        }
        else {
        tall[k] = s2[j];
        j++;
        }
        k++;
        }

        while(i<s1.length){
        tall[k] = s1[i];
        i++;
        k++;
        }
        while(j<s2.length){
        tall[k] = s2[j];
        j++;
        k++;
        }
        //skrivTall(tall);
        //System.out.println("MERGE END\n");
```

```
}
```

Tekst

Denne algoritmen så jeg litt i boka, og på forelesning også forsto jeg godt hva den skulle gjøre. Denne var veldig enkel å implementere og ligner veldig på pseudokoden som ligger i boka. Ingen problemer med å implementere denne.
FletteSortering sjekker om arrayen har mindre enn 2 elementer, hvis så er den allerede ferdig sortert og det er ingenting mer å gjøre.
Vi deler arrayen inn i to deler og kaller så rekursivt på begge delene helt til arrayen er 1 element langt.
Deretter kaller vi på merge som slår sammen to arrayer. Merge metoden tar inn tre arrayer og setter resultatet inn i tall. Vi går gjennom de to arrayene og sammenligner verdiene som vi så setter i stigende rekkefølge inn i tall. Etter while-løkka er alle elementene fra den ene arrayen satt inn i tall og vi kan derfor bare sette resten av den andre rett inn i tall ettersom den er ferdig sortert.

For eksempel kan vi se på en array med tre elementer (2,1,0).
Den blir først delt opp i 21, 21 blir så delt opp i 2 og 1 og satt sammen til 12, deretter har vi 0 som vi må sette sammen med 12 og setter den først fordi den er minst.
Her har vi to sammenslåinger 2 og 1, og 12 og 0.
I den første sammenslåingen har vi s1(2) og s2(1). Vi sammenligner arrayene og ser at 2 er større enn 1 og setter tall[0] = 1. Da er s2 tom, vi går ut av for løkka og setter inn elementene fra s1.
I den andre sammenslåingen har vi S1(12) s2(0) og TALL(210). Vi sammenligner arrayene og ser at s1[0] er større enn s2[0] og setter derfor tall[0] = 0. Da er s2 tom og vi kan sette inn resten av elementene fra s1 og får TALL(0,1,2).
Ser ingen spesielle mønstre eller hva dere tenker på ved tilfeldig og sorterte arrayer.

Hastigheter

Utplukkssortering

Verdi	Tilfeldig unik	Tilfeldig ikke unik	Stigende unik	Stigende ikke unik	Synkende unik	Synkende ikke unik	O(n²)
10 000	42	40	23	21	28	21	100,000,000
110 000	4827	4840	2543	2542	3320	2578	12,100,000,000

210 000	9555	17593	9259	9257	12062	9370	44,100,000,000
310 000	37559	38050	20330	20244	26261	20464	96,100,000,000
410 000	68745	36030	35312	35515	46088	36013	168,100,000,000
510 000	108357	55866	54643	54691	70610	57483	260,100,000,000
610 000	150733	79562	78241	78251	101132	80327	372,100,000,000
710 000	209241	107719	105911	105881	137444	106621	504,100,000,000
810 000	148086	137845	138098	138231	177920	138087	656,100,000,000
910 000	341571	174279	174064	178351	226870	187110	828,100,000,000
1 010 000	414535	214540	214828	217598	278292	218124	1,020,100,000,000

Innstikksortering

Verdi	Tilfeldig unik	Tilfeldig ikke unik	Stigende unik	Stigende ikke unik	Synkende unik	Synkende ikke unik	O(n²)
10 000	62	49	45	46	62	41	100,000,000
110 000	1722	1711	454	455	3029	457	12,100,000,000
210 000	4607	5503	832	866	10161	878	44,100,000,000
310 000	11504	11576	1259	1288	21533	1296	96,100,000,000
410 000	19655	19236	1708	1700	36892	1707	168,100,000,000
510 000	29963	29274	2119	2106	56342	2116	260,100,000,000

610 000	41213	41178	2520	2523	79873	2584	372,100,000,000
710 000	56659	44889	2936	2945	109058	2985	504,100,000,000
810 000	73271	70494	3361	3360	139691	3441	656,100,000,000
910 000	93419	88710	3761	3773	174500	3809	828,100,000,000
1 010 000	110068	108340	4158	4185	215270	4307	1,020,100,000,000

Heap-sortering

Verdi	Tilfeldig unik	Tilfeldig ikke unik	Stigende unik	Stigende ikke unik	Synkende unik	Synkende ikke unik	O(n log n)
10 000	2	0	0	0	0	0	40,000
110 000	13	12	7	7	7	7	554,553
210 000	25	25	14	15	15	15	1,117,666
310 000	38	39	21	23	22	23	1,702,322
410 000	54	53	29	30	30	30	2,301,241
510 000	68	66	36	38	38	40	2,910,860
610 000	96	81	44	46	47	46	3,529,051
710 000	104	96	52	54	56	54	4,154,393
810 000	115	110	60	62	62	62	4,785,872
910 000	127	127	68	73	70	73	5,422,727
1 010 000	145	141	75	81	81	82	6,064,364

Flette-sortering

Verdi	Tilfeldig unik	Tilfeldig ikke unik	Stigende unik	Stigende ikke unik	Synkende unik	Synkende ikke unik	O(n log n)
10 000	2	1	0	0	0	0	40,000
110 000	22	17	10	9	9	9	554,553
210 000	42	34	18	18	19	19	1,117,666
310 000	50	54	31	28	28	30	1,702,322
410 000	69	69	40	37	41	40	2,301,241
510 000	94	89	50	51	51	57	2,910,860
610 000	123	111	59	68	61	61	3,529,051
710 000	135	127	70	69	73	70	4,154,393
810 000	163	147	93	83	81	83	4,785,872
910 000	173	177	100	94	92	95	5,422,727
1 010 000	195	195	127	106	107	113	6,064,364

Arrays.sort()

Verdi	Tilfeldig unik	Tilfeldig ikke unik	Stigende unik	Stigende ikke unik	Synkende unik	Synkende ikke unik	O(n log n)
10 000	2	2	0	0	0	0	40,000
110 000	9	10	1	0	1	0	554,553
210 000	19	20	0	0	0	0	1,117,666
310 000	17	18	1	1	0	1	1,702,322
410 000	24	41	0	0	0	0	2,301,241
510 000	31	34	0	0	0	0	2,910,860
610 000	39	42	0	0	0	0	3,529,051
710 000	45	49	0	0	0	0	4,154,393
810 000	51	56	0	0	0	0	4,785,872

910 000	57	64	0	3	0	0	5,422,727
1 010 000	65	71	0	4	0	0	6,064,364

Sammenligning

Tilfeldig unik

Verdi	Utplukk	Innstikk	Heap-sort	Flette-sort	Arrays.sort
10 000	42	62	2	2	**2**
110 000	4827	1722	13	22	**9**
210 000	9444	4607	25	42	**19**
310 000	37559	11504	38	50	**17**
410 000	68746	19655	54	69	**24**
510 000	108357	29963	68	94	**31**
610 000	150733	41213	96	123	**39**
710 000	209241	56659	104	135	**45**
810 000	148086	73271	115	163	**51**
910 000	341571	93419	127	173	**57**
1 010 000	414535	110068	145	195	**65**

Tilfeldig ikke unik

Verdi	Utplukk	Innstikk	Heap-sort	Flette-sort	Arrays.sort
10 000	40	49	0	1	2
110 000	4840	1711	12	17	10
210 000	17593	5503	25	34	20
310 000	38050	11576	39	54	18
410 000	36030	19236	53	69	41

510 000	55866	29374	66	89	34
610 000	79562	41178	81	111	42
710 000	107719	55889	96	127	49
810 000	137845	70494	110	147	56
910 000	174279	88710	127	188	64
1 010 000	214540	108340	141	195	71

Stigende unik

Verdi	Utplukk	Innstikk	Heap-sort	Flette-sort	Arrays.sort
10 000	23	45	0	0	0
110 000	2543	454	7	10	1
210 000	9259	832	14	18	0
310 000	20330	1259	21	31	1
410 000	35312	1708	39	40	0
510 000	54643	2119	36	50	0
610 000	78241	2520	44	59	0
710 000	105911	2936	52	70	0
810 000	138098	3361	60	93	0
910 000	174064	3761	68	100	0
1 010 000	214828	4158	75	127	0

Stigende ikke unik

Verdi	Utplukk	Innstikk	Heap-sort	Flette-sort	Arrays.sort
10 000	21	46	0	0	0
110 000	2542	455	7	9	0

210 000	9257	866	15	18	0
310 000	20244	1288	23	28	1
410 000	35515	1700	30	37	0
510 000	54691	2106	38	51	0
610 000	78251	2523	46	68	0
710 000	105881	2945	54	69	0
810 000	138231	3360	62	83	0
910 000	178351	3773	73	94	3
1 010 000	217598	4185	81	106	4

Synkende unik

Verdi	Utplukk	Innstikk	Heapsort	Flette-sort	Arrays.sort
10 000	28	62	0	0	0
110 000	3320	3029	7	9	0
210 000	12062	10161	15	19	0
310 000	26261	21533	22	28	0
410 000	46088	36892	30	41	0
510 000	70610	56342	38	51	0
610 000	101132	79873	47	61	0
710 000	137444	109058	56	73	0
810 000	177920	139691	62	81	0
910 000	226870	174500	70	92	0
1 010 000	278292	215270	81	107	0

Synkende ikke unik

Verdi	Utplukk	Innstikk	Heapsort	Flette-sort	Arrays.sort
10 000	21	41	0	0	0
110 000	2578	457	7	9	0
210 000	9370	878	15	19	0
310 000	20464	1296	23	30	1
410 000	36013	1707	30	40	0
510 000	57483	2116	40	57	0
610 000	80327	2584	46	61	0
710 000	106621	2985	54	70	0
810 000	138087	3441	62	83	0
910 000	187110	3809	73	95	0
1 010 000	218124	4307	82	113	0

Utplukkssortering og innstikksortering skal begge bruke O(n²) - kvadratisk tid.

Prosentvis økning basert på O(n²)	Faktisk økning utplukkssortering tilfeldig unik	Faktisk økning innstikksortering tilfeldig unik
12000% økning fra 10.000 til 110.000 elementer.	11392%	2677%
264.5% økning fra 110.000 til 210.000	100%	167%
117.9% økning fra 210.000 til 310.000	293%	150%
74.9% økning fra 310.000 til 410.000	83%	70%
54.8% økning fra 410.000 til 510.000	57%	52%
43.1% økning fra 510.000 til 610.000	39%	37%
35.5% økning fra 610.000 til 710.000	38%	37%

30.2% økning fra 710.000 til 810.000	-29%	29%
26.2% økning fra 810.000 til 910.000	130%	27%
23.2% økning fra 910.000 til 1.010.000	21%	17%

Utplukkssortering har veldig høye verdier for sorteringen, men sammenligner vi disse med hva som var forventet av O-notasjonen stemmer de likevel ganske bra. Eneste er at det skjer noe rart noen ganger så en sortering som er stor tar mindre tid enn sorteringer som var mindre. Innstikksortering har relativt lave verdier sammenlignet med utplukkssortering noe som skyldes at den første økningen fra 10,000 tall til 110,000 tall er mye mindre, med unntak av verdiene for synkende unike tall hvor de går relativt tett på hverandre.

Heapsortering, flettesortering og Arrays.sort skal alle bruke O(n log n) - logaritmisk tid.

Prosentvis økning basert på O(n log n)	Faktisk endring heap-sortering tilfeldig unik	Faktisk endring flette-sortering tilfeldig unik	Faktisk endring Arrays.sort() tilfeldige unike
1286.3% økning fra 10.000 til 110.000 elementer.	550	1000	350
101.6% fra 110.000 til 210.000	92	90	111
52.3% økning fra 210.000 til 310.000	52	19	-10
35.2% økning fra 310.000 til 410.000	42	38	41
26.5% økning fra 410.000 til 510.000	25	36	39
21.2% økning fra 510.000 til 610.000	41	30	25
17.7% økning fra 610.000 til 710.000	8	9	15
15.2% økning fra 710.000 til 810.000	10	20	13

13.3% økning fra 810.000 til 910.000	10	6	11
11.8% økning fra 910.000 til 1.010.000	14	12	14

Heapsortering, flettesortering og Arraysortering går mye raskere enn de to foregående og ser vi på verdiene forventet av O-notasjonen passer dette ganske godt med hvor mye tiden faktisk økte mellom de ulike verdiene. Noe som overrasker meg er hvor rask Arraysortering er på stigende og synkende verdier hvor den har 0 på tid uansett hvor stort array

Tekst

Er det noen av resultatene som overrasker deg?
Det overrasket meg litt hvor lang tid dette tok på store input og hvor stor forskjell det faktisk var mellom n^2 og n log n. Det overrasket meg også at det var så stor forskjell mellom utplukkssortering og innstikksortering selv om begge skal bruke $O(n^2)$ tid. Det overrasket meg også hvor effektiv Arrays.sort() var på de fleste typer input.