IDATT2101 Øving 3

Avansert sortering, forske på quicksort med forbedringer

Trang Minh Duong

Sorteringsoppgave 3, quicksort med min forbedring

I denne øvingen har jeg valgt å bruke quicksort (med Hoares oppdelingsalgoritme) med én pivot og bygget videre på denne algoritmen. Denne algoritmen benytter en pivot som er indeksen i midten av lista.

Forbedringer

Jeg har valgt å implementere noen metoder for å forbedre algoritmen, både fra forslag i oppgaveteksten og litt oppsøking på nett.

Som sagt tidligere bruker algoritmen Hoare-partisjon og midtpivot som kjerne. Dette kan vi se i koden: (`pivot = A[(l+h)/2]`).

```
1  int partition(vector<int>& A, int l, int h) {
2      int pivot = A[(l + h) / 2];
3      int i = l-1; int j = h+1;
4
5      while (true) {
6          do { i++; } while (A[i] < pivot);
7          do { j--; } while (A[j] > pivot);
8          if (i >= j) return j;
9          swap(A[i],A[j]);
10     }
11 }
```

Listing 1: Hoare-partisjon

I tillegg benytter programmet sentinel-trikset (sette min og maks verdiene ytterst). Før sortering flyttes minste element til `A[0]` og største til `A[n-1]`. Deretter sorteres kun det indre intervallet `[1, n-2]`. Effekten er at man i hvert rekursive kall trygt kan sjekke endepunktene utenfor deltabellen `A[l-1]` og `A[h+1]` uten ekstra grensetester. Bruken av sentineller unngår vi unødvendige sammneligninger og sorteringer i lister med mange duplikater.

```
1  void set_sentinels(vector<int>& A) {
2      int n = (int)A.size();
3      if (n <= 1) return;
4      int min_i = 0, max_i = 0;
5      for (int i = 1; i < n; ++i) {
6          if (A[i] < A[min_i]) min_i = i;
7          if (A[i] > A[max_i]) max_i = i;          // Holde
               styr over minst i og storst i
8      }
9
10     if (A[min_i] == A[max_i]) return;
11     swap(A[0], A[min_i]);                        // Alle
           elementene er like => putt minste i i begynnelsen
12
13     if (max_i == 0) max_i = min_i;               // Storst i
           i begynnelsen => bytt posisjon med minst i
14
15     swap(A[n-1], A[max_i]);                      // Flytt
           storst i mot slutten av lista
16 }
```

Listing 2: Sentinel

Programmet tar i bruk en duplikat-detektor som et forslag fra oppgaveteksten. Dette trikstet implementerer tidlig retur på like endepunkter. Når en deltabell ligger mellom to like delingstall, må alle elementene inni også være like, og deltabellen er allerede "sortert". Dette gir stor gevinst når det finnes mange duplikater. Eksempel ligger under i kodelistingen.

```
1  // Sikkerhetssjek slik at den leser ikke utenfor grensen
2          assert(l >= 1 && h + 1 < (int)A.size());
3          if (A[l-1] == A[h+1]) return;            // Om alle
               elementene er like
4
5          // Bruk quicksort for stor liste (indekser > cutOff)
               , insettingsort for deltabeller (indekser <
               cutOff)
6          if (h - l + 1 <= cutOff) {
7              insertion_sort(A, l, h);
8              return;
9          }
```

Listing 3: Duplikat-detektor

Programmet bruker også innsettingssortering for små deltabeller (cutoff ≈ 24). Når deltabellens størrelse er liten ($\leq$ cutoff), byttes det til innsettingssortering. Dette reduserer (rekrusive) kall- og partisjonerings indirekte konstnader (overhead) for bittesmå biter og forbedrer konstantleddet i praksis.

```
1  void insertion_sort(vector<int> &A, int l, int h) {
2      for (int i = l+1; i <= h; i++) {
3          int x = A[i], j = i - 1;                 // Sett x
               som nokkelverdi, start sammenligning med resten
4
5          while (j >= l && A[j] > x) {             // Sjekk
               grensen og om elementet er storre enn x
6
7              A[j+1] = A[j];                       // Flytt det
                   storre elementet til hoyre, apning ved j
8
9              j--;                                 // Fortsett
                   a flytte j til venstre til x er pa riktig
                   plass
10         }
11         A[j+1] = x;                              // Putt x i
               dens riktig plass
12     }
13 }
```

Listing 4: Insettingssort

Sortér minste side først og eliminer halerekursjon er også en annen metode brukt i dette programmet. Etter partisjonering sorteres den minste siden rekursivt, og den største siden håndteres iterativt i løkka (halerekursjon-eliminering). Dette begrenser stackdybden til $O(\log n)$ i stedet for potensielt $O(n)$ (selv når partisjonen er veldig forskyvet), og kutter noe kall-kostnader.

```
1  int j = partition(A, l, h);
2          if (j - l < h - j) {                    // Om
               venstre side er mindre
3
4               quicksort_core(A, l, j);           // Rekursere
                    den mindre siden [l...j]
5
6               l = j + 1;                         // Eliminere
                    halerekursjon p   den storre siden via
                   lokke gjennom [j+1...h]
7
8          } else {                                // Hoyre
               side mindre eller like stor
9
10              quicksort_core(A, j + 1, h);       // Rekursere
                    den mindre siden [j+1...h]
11
12
13              h = j;                             // Lokka
                   fortsetter p   st rre siden
14         }
```

Listing 5: Sortere minste først

Tidsmålinger

Under er tabellene for 5 ulike tidsmålinger i nanosekund for n = 1 000 000:

For hvert unike n:

Quicksort	Quicksort med forbedring	Differansen mellom de
195 000 400	204 262 700	9 262 300
190 125 800	215 182 600	25 056 800
190 830 300	205 285 600	14 455 300
189 277 400	196 269 100	6 991 700
198 363 700	201 390 500	3 026 800

Tabell 1: Tidsmåling for unike n

Vi får en gjennomsnittstidsmåling på 192 719 520 ns (vanlig quicksort), 204 478 100 ns (forbedret quicksort) med en gjennomsnittsdifferanse på 11 758 580 ns.

For en allerede sortert liste med unike n og forholdet mellom gjensorteringen og den førstegangssorterte lista:

Vi får da gjennomsnittlig tidsmålinger på 65 794 820 ns (vanlig quicksort) med 0.3415x, 70 603 300 ns (forbedret quicksort) med 0.3455x. Dette betyr at resorteringen tok bare

Quicksort	**Forhold**	**Quicksort med forbedring**	**Forhold (forbedret)**
66 291 400	0.3400x	69 250 100	0.3390x
66 621 700	0.3504x	69 483 100	0.3229x
64 868 600	0.3399x	73 880 100	0.3599x
65 746 800	0.3474x	66 570 500	0.3392x
65 445 600	0.3299x	73 832 700	0.3666x

Tabell 2: Tidsmåling for allerede sortert liste med unike n og forholdene

34.15% (vanlig quicksort) og 34.55% (forbedret quicksort) av tiden det tok å sortere listen første gang

For en liste med n indekser hvor annenhvert tall er 42:

Quicksort	**Quicksort med forbedring**	**Differansen mellom de**
163 582 200	122 773 900	40 808 300
164 733 200	125 098 900	39 634 300
160 181 300	127 305 500	32 875 800
163 351 200	120 240 600	43 110 600
168 314 900	120 016 600	48 298 300

Tabell 3: Tidsmåling for liste med duplikater

Gjennomsnittstidsmålinger er 164 032 560 ns (vanlig quicksort), 123 087 100 ns (forbedret quicksort) med en gjennomsnittsdifferanse på 40 945 460 ns.

For en allerede sortert liste med duplikater og forholdet mellom gjensorteringen og den førstegangssorterte lista:

Quicksort	**Forhold**	**Quicksort med forbedring**	**Forhold (forbedret)**
105 875 600	0.6472x	52 153 200	0.4248x
103 081 300	0.6257x	50 661 600	0.4050x
105 972 600	0.6616x	51 445 000	0.4041x
104 762 700	0.6413x	51 159 300	0.4255x
106 612 400	0.6334x	51 378 800	0.4281x

Tabell 4: Tidsmåling for allerede sortert liste med duplikater og forholdene

Vi får gjennomsnittlig tidsmålinger på 105 260 920 ns (vanlig quicksort) med 0.6418x, 51 359 580 ns (forbedret quicksort) med 0.4175x. Dette betyr at resorteringen tok bare 64.18% (vanlig quicksort) og 41.75% (forbedret quicksort) av tiden det tok å sortere listen første gang

Disse tidsmålingene stemmer med forventningen utifra koden og metodene brukt. Vi kan se at:

Lister med unike tall (tilfeldig permutasjon) er ofte litt tregere med bruk av den forbedret varianten (204 478 100 ns) enn "vanlig"quicksort (92 719 520 ns). Dette skyldes én ekstra $O(n)$-pass for å finne min/max (sentinel), samt en ekstra sjekk per kall (`A[l-1] == A[h+1]`) som sjelden slår til uten duplikater. Cutoff til innsettingssort hjelper, men kompenserer ikke alltid helt for kostnaden/overhead. Små negative utslag er derfor forventet.

For **lister med mange duplikater** (f.eks. annenhver = 42) er den forbedrede varianten (123 087 100 ns) er klart raskere enn den "vanlige"quicksort-en (164 032 560 ns). «Like endepunkter»-testen slår ofte til og hopper over hele deltabeller som består av identiske verdier. Resultatet er færre rekursive kall og færre sammenligninger.

For **allerede sorterte tabeller** (resortering) blir begge varianter mye raskere ved gjensortering. Midtpivot + Hoare gir nær balanserte delinger på sortert input, og cutoff reduserer kostnader. Forbedret versjon har fortsatt sentinel- og sammenligningskostnad, så den vinner ikke nødvendigvis her, men vi kan garantere at det er ingen n^2-atferd (forholdet mellom er godt under $2\times$), i tråd med kravene.

Kompleksitetsanalyse

Empirisk modellering av kjøretider

Når vi skriver

$$T_{\text{unique}}(n) \;\approx\; c\,n\log_2 n \;+\; a\,n \;+\; b \qquad \text{og} \qquad T_{\text{dups}}(n) \;\approx\; k\,n \;+\; b,$$

lager vi en enkel empirisk modell for målt tid $T(n)$ og estimerer konstantene $c, a, k, \ldots$ fra måledata.

1) Unike elementer ($\approx n\log n$)

Del med n:

$$\frac{T(n)}{n} \;\approx\; c\,\log_2 n \;+\; a \;+\; \frac{b}{n}.$$

For $n \to \infty$ er $\frac{b}{n} \to 0$, så vi kan skrive om og tilnærme en linje

$$Y \;=\; c\,X + a, \qquad \text{der} \quad Y = \frac{T(n)}{n},\; X = \log_2 n.$$

Rask rimelighetsvurdering for $n = 10^6$.

$$\log_2(10^6) \approx 19.9315686, \qquad n \log_2 n \approx 19\,931\,568.6.$$

- Vanlig quicksort (unike): $T \approx 184\,792\,400$ ns $\Rightarrow\ c \approx \dfrac{T}{n \log_2 n} \approx 9.27$ ns.
- Forbedret quicksort (unike): $T \approx 204\,962\,200$ ns $\Rightarrow\ c \approx 10.28$ ns.

Dette stemmer med at forbedringen gjør litt ekstra arbeid på helt unike data (sentinel-sjekk, cutoff-gren, osv.), altså et høyere konstantledd.

2) Mange duplikater ($\approx n$)

Modellen er

$$T(n) \ \approx\ k\,n + b \quad \Rightarrow \quad \frac{T(n)}{n} \ \approx\ k + \frac{b}{n}.$$

For $n \to \infty$ er $T(n)/n \approx k$ (nesten konstant på tvers av n).

Rask rimelighetsvurdering for $n = 10^6$.

- Vanlig quicksort (dups): $T \approx 160\,179\,700 \ \Rightarrow\ c \approx \dfrac{T}{n \log_2 n} = \frac{160\,179\,700}{19\,931\,568.6} \approx 8.04$ ns.
- Forbedret quicksort (dups): $T \approx 124\,314\,600$ ns $\Rightarrow\ k \approx T/n \approx 124.31$ ns/element.

Dette viser målingene for den forbedrede varianten at $T(n)$ vokser omtrent proporsjonalt med n. Dette stemmer med mekanikken i forbedringen. Når et delintervall er omgitt av likeverdige naboelementer, kan vi anta at alle elementene innenfor også er like. Dette gjør det mulig å eliminere videre rekursjon for dette intervallet. I tillegg håndterer cutoff (`insertion_sort` for små delintervall) kortere rester effektivt. Summen blir at mye av $n \log n$-arbeidet forsvinner, og vi får empirisk $T(n) \approx k\,n$.

3) Gjensortering

For **unike elementer (allerede sortert)** har begge varianter fortsatt $\Theta(n \log n)$ på sortert input (midtpivot + Hoare gir nær-balanserte delinger). Like endepunktertesten hjelper sjelden når alle verdier er unike. Begge unngår n^2-atferd på sortert input (midtpivot + Hoare + halerekursjonseliminering).

Konstantleddet blir lavt for begge, men den forbedrede varianten betaler litt ekstra overhead (sentinel-flytt, grensesjekk, cutoff-tester). Vi kan se dette i forholdet mellom gjensortering med førstegangsortering:

$$\text{vanlig} \ \sim 0.3415\times \quad \text{vs.} \quad \text{forbedret} \ \sim 0.3455\times,$$

altså ofte litt raskere uten forbedring når alt allerede er sortert og unikt.

For **mange duplikater (allerede sortert)** kan delintervaller droppes (i forbedret varianten) når elementet før og etter intervallet er like, noe som gir beste fall $\Theta(n)$ (f.eks. alle like). Uten forbedring forblir det i praksis $\Theta(n \log n)$.

Her gir forbedringen tydelig gevinst, som vi observerte:

$$\text{vanlig } \sim 0.6418\times \quad \text{vs.} \quad \text{forbedret } \sim 0.4175\times$$

ved gjensortering av duplikatrike tabeller. Dette skyldes hyppige kortslutninger av hele delintervaller + effektiv `insertion_sort` for små rester.

Teoretisk oversikt

Variant	**Beste**	**Forventet**	**Verste**
Quicksort (vanlig)	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n^2)$
Quicksort (forbedret)	$\Theta(n)$	$\Theta(n \log n)$	$\Theta(n^2)$