Programmeringsuppgift: mergesort och quicksort

Den här uppgiften handlar om mergesort och quicksort. Som utgångspunkt får ni en exempelklass `MyInsertionTest` implementerad både i Java och C#, som det är meningen att ni ska läsa och klipp-och-klistra kod från till era egna program för att slippa spendera tid på det som inte är väsentligt i uppgiften. (Skriver ni i något annat språk än Java eller C# får ni dock själva fixa hur man gör till exempel filläsning/-skrivning och slumptal.)

Det rekommenderas att ni utför programmeringsuppgifter två och två, men det är tillåtet att jobba ensam eller i grupper om tre. (Faran med att vara tre är att inte alla hänger med på allt, och får sämre förutsättningar för att ta till sig resten av kursen, men jag lämnar det avgörandet åt er.)

Algoritmbeskrivningar

De här beskrivningarna kan ni utgå från när ni tänker ut er programkod:

Mergesort på del av array från *lo* till *hi*:

- Om *hi* inte är större än *lo* har vi högst ett element och då är denna del av arrayen redan sorterad, så klart!
- Annars fortsätt med att sätta *mid* till mittpunkten (avrundat neråt) mellan *lo* och *hi*.
- Kör mergesort rekursivt på delen från *lo* till *mid*.
- Kör mergesort rekursivt på delen från *mid* + 1 till *hi*.
- Nu har vi två halvor som vardera är sorterade. Kör *merge* på dem.

Merge på delar av array (*lo* … *mid*) och (*mid* + 1 … *hi*):

- Kopiera över allt – från position *lo* till *hi* – till extralagringsarray.
- Låt *i* börja på *lo* (början på den vänstra delen) och *j* på *mid* + 1 (början på den högra).
- Låt *k* gå igenom alla värden från *lo* till *hi*, och utför följande för varje:
    - Vi ska nu ta ett värde från extraarrayen, antingen från position *i* (vänstra delen) eller från position *j* (högra delen), och lägga på position *k* i originalarrayen.
    - Om *i* är större än *mid* (alltså vänstra delen är slut), tar tar vi värdet från position *j* (och räknar upp *j*).
    - Om *j* är större än *hi* (alltså vänstra delen är slut), tar tar vi värdet från position *i* (och räknar upp *i*).

    - Annars jämför vi värdet på position *i* med värdet på position *j*, och tar det minsta (och räknar upp antingen *i* eller *j*, det vi tog från).

Quicksort på hel array:

- Blanda om arrayen (för att undvika urartningsfall).
- Kör quicksort på ”delen” av arrayen från 0 till $N - 1$ (alltså hela arrayen).

Quicksort på del av array från *lo* till *hi*:

- Om *hi* inte är större än *lo* har vi högst ett element och då är denna del av arrayen redan sorterad, så klart!
- Annars, välj elementet på position *lo* som pivotelement. Vi kallar det *p*. Vi ska partitionera så att inga element som är större än *p* står till vänster om något som är större än *p*.
- Låt *i* börja på *lo* och *j* på *hi* + 1. Vi ska räkna upp *i* och räkna ner *j* tills de möts. Upprepa:
    - Räkna upp *i*, först en gång, och fortsätt så länge *i* är mindre än *hi* och elementet på position *i* är mindre än *p*.
    - Räkna ner *j*, först en gång, och fortsätt så länge elementet på position *j* är större än *p*. (Vi behöver faktiskt inte kolla att *j* inte går förbi *lo*, eftersom vi vet att elementet på position *lo* är *p* och därför inte större än *p*.)
    - Om *i* och *j* har mötts, alltså $i \geq j$, bryt loopen.
    - Annars, byt värdena på position *i* och *j*, och fortsätt.
- Slutför partitioneringen genom att lägga in *p* på rätt plats: byt värdena på position *lo* och *j*. Nu vet vi att inga värden till vänster om *j* är mindre än *p*, och inga värden till höger om *j* är större än *p*.
- Kör mergesort rekursivt på delen från *lo* till $j - 1$.
- Kör mergesort rekursivt på delen från $j + 1$ till *hi*.

Uppgifter

1. Implementera mergesort och quicksort specifikt för att sortera en array av `byte` (som `MyInsertionTest` gör). Kopiera inte bara kod från boken eller från nätet – förutom det ni får från `MyInsertionTest`(observera att den innehåller en shuffle-metod ni kan använda till quicksort) och ändra, utan skriv själv från scratch, med utgångspunkt från följande beskrivningar. (Om ni väljer att ”fuska” genom att ta kod från andra håll är det inget vi kan göra något åt, men det förtar en stor del av poängen med uppgiften och gör att ni inte lär er så mycket. Om det är någon detalj i beskrivningen ni inte förstår utan att titta i bokens kod får ni naturligtvis göra det, men skriv aldrig av utan att förstå.)

Testkör koden enligt samma modell som i `MyInsertionTest` för att se till att det funkar korrekt.

2. Välj ett eller flera av dessa alternativ för att testa för att undersöka eller försöka förbättra er mergesort eller quicksort (eller båda), och gör en utvärdering av hur algoritmen beter sig i förhållande till den oförändrade. Observera att det inte alls är säkert att allt faktiskt medför någon märkbar förändring i exekveringstiden i det praktiska fallet, det är bland annat det ni ska utvärdera!
    a. Mergesort och quicksort: byt till insertion sort för små delar. Insertion sort är ofta snabbare än mer avancerade metoder när antalet element är litet. Ändra i koden för de båda algoritmerna så att rekursionen inte går hela vägen ner till delarrayer av storlek 1, utan byter till insertion sort när antalet element är mindre än något *M*. Experimentera för att ta reda på vad som är ett bra värde på *M* (är det kanske 10, 100, 1000? binärsök gärna, manuellt eller automatiskt, för att hitta en mer precis bästa brytpunkt).

        I quicksort innebär denna förändring att det är lite onödigt att göra *shuffle* på alla element i arrayen: det borde räcka att räkna ut ett slumptal varje gång man kör en partitionering, vilket nu blir mindre än det totala antalet element. Därför kan detta experiment gärna kombineras med c (*ingen shuffle*). Blir det någon skillnad i körtid tack vare det minskade antalet ”slumptal”? Blir det samma förändring för alla arraystorlekar, eller varierar det med antalet element (fundera och testa)?

    b. Mergesort: minimera kopieringar till extralagring. Den givna, enklaste, formuleringen av mergesort kopierar en del av arrayen till extralagringen vid varje *merge*. Det är egentligen onödigt. Om den näst översta rekursionsnivån istället lägger *resultatet* i extraarrayen, så kan den sista merge-operationen hämta från extraarrayen och lägga resultatet i originalarrayen. Så om vi flyttar på ena hållet när rekursionsdjupet är jämnt och på andra när rekursionsdjupet är udda, bör vi i bästa fall kunna helt eliminera kopieringarna. Det kan dock behövas en kopieringsomgång, eftersom vi inte kan vara säkra på att alltid få ett udda antal rekursionsnivåer totalt.

        En enkel idé, som kan vara klurig att få rätt på. Blir det någon skillnad i körtid tack vare det minskade antalet kopieringar? Blir det samma förändring för alla arraystorlekar, eller varierar det med antalet element (fundera och testa)?

    c. Quicksort: Ingen shuffle. Gör inte den inledande omblandningsoperationen i quicksort, utan testa att istället välja

pivotelement med några olika strategier. Prova följande idéer. I samtliga fall blir det antagligen lättast att implementera genom att man byter elementet man valt mot det första, och sedan kör algoritmen precis som i originalkoden.

- Originalkoden väljer det första. När får vi ett urartningsfall som ger kvadratisk tidskomplexitet (fundera och testa)?
- Välj det sista. När får vi ett urartningsfall (fundera och testa)?
- Välj det mittersta. Kan ni hitta ett urartningsfall?
- Välj ett slumpmässigt element. (Se i shuffle-implementationen hur man får fram ett slumptal mellan två gränser.)
- Om ni vill kan ni också prova att välja medianen av det första, sista och mittersta. Kan ni hitta ett urartningsfall?

Fundera och gör mätningar. För att testa på fallet när input redan är sorterad, eller sorterad i omvänd ordning, använd era egna sorteringsalgoritmer för att placera elementen i den ordningen. Ni kan se i `MyInsertionTest` hur man kan spara ner en vektor till en fil, om ni vill göra det för att få en annan fil att använda som input. Andra specialfall är klurigare att tänka ut och generera; inget krav att ni lyckas med det, men fundera gärna på det.

Inlämning

Ni lämnar in programkoden ni har skrivit, tillsammans med ett PDF-dokument där ni beskriver resultatet av utvärderingen ni gjort för den eller de algoritmniförändringar ni valt att göra under punkt 2.

Författare: Jesper Larsson
Created: 2017-09-18 Mon 12:14