Labbrapport Labb 2 D0012E

Författare
Adam Gatmon | adagat-1@student.ltu.se
David Malmblad | maldav-1@student.ltu.se
Albin Kullberg | albkul-0@student.ltu.se

LULEÅ TEKNISKA UNIVERSITET

3 januari 2024

Innehåll

1 Inledning

I denna laboration kommer vi designa 3 algoritmer.

De två första algoritmernas syfte är att returnera de tre minsta elementen i storleksordning men dock använda sig av två olika meteoder (incremental) och (divide and conquer)

Den sista algoritmen löser problemet att hitta den maximala summan av en subarray i en array (A) av icke noll reella tal. Genom en (divide and conquer) metod.

2 Teori

2.1 Inkrementell metod

Att anväda en inkrementell metod betyder att man steg för steg löser ett problem, ofta genom användning av loopar eller nästlade loopar. Man delar alltså upp problemet i små delar och hanterar delarna en och en genom att bygga vidare på sitt resultat, så att när alla delar av input har hanterats kommer slutresultatet vara en korrekt för hela input.

Ett enkelt exempel är att beräkna summan av en lista av element; genom att gå genom listan och addera varje element man passerar till en kumulativ variabel kommer den variablens värde vara den totala summan av alla elementen när hela listan har behandlats.

2.2 Divide and conquer

Divide-and-conquer-metoden är en strukturerad metod för att hantera komplexa problem. Grundtanken är att bryta ner ett stort problem i mindre subproblem och därefter hantera dem induviduellt. Sedan kombineras resultaten för att skapa en lösning för orginalproblemet. Denna process fortsätter rekursivt tills basfallet (den enklaste formen av problemet) som kan lösas.

3 Diskussion

3.1 Uppgift 1

Uppgift 1 handlar om att ta fram de tre lägsta värdena, i sorterad ordning, ur en lista av storlek $n = 3 \cdot 2^{k-1}, k \in Z^+$. Vi har löst problemet med två olika metoder: en inkrementell/iterativ lösning, och en Divide&Conquer-baserad lösning. Problemet är i grunden ett sorteringsproblem, som alltså i princip inte går att lösa snabbare än $O(n \cdot \log n)$, men eftersom vi bara söker de tre lägsta värdena och inte alla värden, kan problemet lösas i $O(3 \cdot n) \sim O(n)$.

3.1.1 Iterativ implementation

Uppgiften kan lösas iterativt genom en simpel for-loop som mäter varje värde i våran input lista med en nyskapad array med tre platser, där de 3 lägsta värderna från våran input lista kommer att sparas.

Till att börja med kommer arrayen att fyllas med tre stycken inf värden, som kommer att vara större än något annat random värde som någonsin kommer att kunna jämföras med dessa startvärden. Efter detta startas en for loop som går igenom hela input listan 1 gång.

Vår algoritm kommer att genom loopen först jämföra det aktuella värdet i listan med värdet från den första positionen i vår output array. Om värdet från input listan är mindre än det värde som redan finns i vår output lista kommer vårat nuvarande lägsta tal i output listan att flyttas till nästa position i output listan så att det nya minsta värdet vi fann i input arrayen kan ta dess plats. Innan dess kommer även det nuvarande värdet från den andra postionen i output listan flyttas till position 3 i output listan, vilket skriver över vårat tidigare tredje största tal som var sparat.

Om det nuvarande värdet i vår input lista inte visar sig vara lägre än det nuvarande lägsta talet i våran output lista kommer vi även att mäta talet med det näst lägsta talet i output listan för att se om talet i fråga är mindre. I detta fall förflyttas det tidigare näst största talet till position 3 i output listan så att vårat nya minst största tal kan ta dess plats i output arrayn.

Om värdet åter igen visar sig vara större än vad vi redan sparat i våran output array testas det aktuella talet mot värdet i den tredje och sista positionen i output listan. Om talet visar sig vara mindre kommer värdet att skriva över det nuvarande tredje sista värdet i arrayn. Annars händer ingenting med talet.

Om värdet visat sig vara större än det nuvarande minsta, näst minsta och tredje minsta talet i vår output lista kommer värdet att ignoreras. Oavsett resultatet av jämförelsen kommer loopen sedan att startas om för att jämföra nästa tal i input listan, tills det att hela listan körts igenom en gång.

På grund av hur våran algoritm är strukturerad kommer vår input lista endast loopas igenom en gång, och endast 1-3 jämförelser kommer att göras per värde. I värsta fall kommer alla värden att jämföras 3 gånger vilket skapar en tidskomplexitet som grafiskt kan beskrivas som $Y = 3X$. Detta visar att våran funktion även i ett worst case scenario är linjär och har en tidskomplexitet av $O(n)$.

Givet från uppgiften $n \geq 3$.

1. Induktionsbas: För det minsta fallet där $n = 3$, har vi endast tre element. I detta fall görs tre jämförelser, och algoritmen returnerar de tre minsta elementen.

2. Induktionsantagande: Antag att algoritmen fungerar korrekt för ett givet $n = 3 \cdot 2^p - 1$, där p är en positiv heltalskonstant.

3. Induktionsteg: Vi vill visa att algoritmen också fungerar för $n = 3 \cdot 2^{(p+1)} - 1$.

3.1 Dela upp i två delar: För $n = 3 \cdot 2^{(p+1)} - 1$, dela upp inputen i två delar: den första delen innehåller de första $3 \cdot 2^p - 1$ elementen och den andra delen innehåller de återstående elementen.

3.2 Använd induktionsantagandet: Tillämpa induktionsantagandet på den första delen och få de tre minsta elementen (x', y', z').

3.3 Jämför med de nya elementen: Jämför (x', y', z') med de första tre elementen i den andra delen (a, b, c). Uppdatera de tre minsta elementen (x, y, z) om det behövs.

Jämför x' med a, b, c och uppdatera x, y, z om det behövs. Jämför y' med a, b, c och uppdatera y, z om det behövs. Jämför z' med a, b, c och uppdatera z om det behövs.

3.4 Beräkna totalt antal jämförelser: Totalt antal jämförelser är antalet jämförelser för de första $3 \cdot 2^p - 1$ elementen

3.1.2 Divide&Conquer-implementation

Eftersom uppgiften tillåter antagandet att $\frac{n}{3} = 2^{k-1}, k \in Z^+$ kan uppgiften lösas rekursivt genom att dela input i två tills basfallet $n = 3$ nås. När basfallet nåtts är förväntat resultat bara en sorterad version av input, vilket i vårt fall implementerades med en simpel hårdkodad insertionsort. I alla andra fall krävs då att vi väljer de lägsta tre talen från två sorterade listor av längd 3, vilket enkelt kan göras genom att jämföra de lägsta talen i vardera lista och plocka utdet lägsta (detta är samma metod som används i mergesort). Det första talet kommer då vara det lägsta, det andra det näst lägsta osv. Slutligen returneras denna lista av de tre lägsta talen. Ett enkelt induktionsbevis följer nedan:[1]

1. Basfall: $k = 1 \Leftrightarrow n = 3 \Rightarrow insertionsort(x, y, z)$. Detta kommer alltid producera en lista med tre värden x, y, z så att $x < y < z$.
   $\therefore$ Basfallet är korrekt.

2. Induktionssteg: Anta att algoritmen är korrekt för alla $k = 1, \ldots, p$.

3. Visa att algoritmen är korrekt för $k = (p + 1)$.

   $$n = 3 * 2^{k-1} = 3 * 2^{(p+1)-1} = 3 * 2^p = 2 * (3 * 2^{p-1})$$

   n kan alltså delas upp i två lika stora listor, som båda har storleken $3*2^{p-1} \Rightarrow k = p$. Enligt induktionsprincipen kan funktionen exekveras rekursivt på dessa listor och producera två sorterade listor innehållande tre element vardera.
   Eftersom listorna är sorterade kommer det minsta elementet vara det första värdet i någon av listorna, detta värdet plockas då ut och sätts som x. Av samma princip måste det näst minsta elementet vara det nya minsta elementet av de två första elementen. Detta plockas ut och sätts som y. Samma procedur upprepas för z. Det som återstår blir $x < y < z < \{resten\}$.
   $\therefore$ Genom axiomet för induktion är algoritmen därmed korrekt för alla $n = 3 \cdot 2^{k-1}, k \geq 1$

Det exakta antalet jämförelser (*if-statements*) som exekveras i vår implementation är alltid exakt $\frac{8}{3}n - 4$. Detta kan observeras genom att titta på mönstret för antalet exekveringar av funktionen. Då funktionen (bortsett från rekursioner, alltså i basfallet) alltid gör 4 jämförelser behöver vi bara beräkna antalet exekveringar av funktionen för ett givet n.

$$\frac{n}{3} = 1 \Rightarrow 1 \text{ exekvering (basfallet)}$$
$$\frac{n}{3} = 2 \Rightarrow 3 \text{ exekveringar (1 merge + 2 basfall)}$$
$$\frac{n}{3} = 4 \Rightarrow 7 \text{ exekveringar (1 merge + 2 merges + 4 basfall)}$$
$$\ldots$$

Snabbt dyker ett mönster upp: antal exekveringar $= \frac{n}{3} \cdot 2 - 1$ vilket ger antal jämförelser $= 4 \cdot (\frac{n}{3} \cdot 2 - 1) = 4 \cdot (\frac{2}{3}n - 1) = \frac{8}{3}n - 4$. Detta kan även bevisas med ett induktionsbevis. Vi låter definiera antalet jämförelser som $C(n) = \frac{8}{3}n - 4$ och bevisar (vi bevisar endast för positiva heltal k, dvs enligt uppgiften giltiga storlekar av input):

1. Basfall: $n = 3 \Rightarrow C(3) = \frac{8}{3}3 - 4 = 8 - 4 = 4$. I basfallet sker endast en exekvering vilket innebär en total av 4 jämförelser; basfallet är korrekt.

2. Induktionssteg: Anta att funktionen är korrekt för alla $n = 3, \ldots, p$.

3. Visa att algoritmen är korrekt för $n = p \cdot 2$. Totala jämförelser är 4 för den översta exekveringen adderat med summan av båda underliggande rekursioners jämförelser, alltså:

   $$C(n) = 4 + 2 \cdot C(n/2) = 4 + 2 \cdot C(p) = 4 + 2 \cdot (\frac{8}{3} \cdot \frac{n}{2} - 4) = 4 - 8 + \frac{8}{3} \cdot n = \frac{8}{3}n - 4$$

   $\therefore$ Genom axiomet för induktion är funktionen därmed korrekt för alla $n = 3 \cdot 2^{k-1}, k \geq 1$

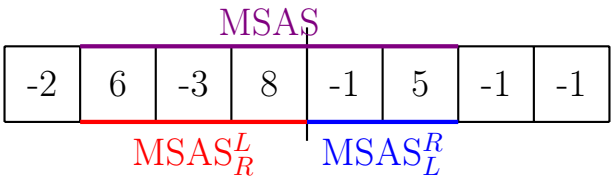
[1] Detta bevis antar för enkelhetens skull att insertionsort är korrekt. Eftersom $n = 3$ alltid är sant där insertionsort exekveras kan korrektheten enkelt verifieras genom att pröva alla fall.

3.2 Uppgift 2

Uppgift 2 var lite klurigare. Den naiva lösningen är att prova alla möjliga startpunkter med alla möjliga slutpunkter och för varje par av dessa beräkna summan. Denna lösning kommer dock i värsta fall köra i $O(n^2)$ eller t.o.m. $O(n^3)$, så lite optimeringar måste göras.

Vi låter definiera termen MSAS för Maximum SubArray Sum, det vill säga den största summan av en kontinuerlig sublista av input. Med att MSAS "ligger i" ett visst område menas därför att den börjar och slutar innanför dessa gränser. Att studera var MSAS börjar och slutar är högst relevant för problemet, då vi till största mån vill undvika att behöva iterera genom hela input och beräkna en viss summa. Eftersom lösningen ska vara av typen Divide&Conquer måste algoritmen ta hänsyn till följande: För varje input ligger MSAS antingen bara i den vänstra halvan, bara i den högra halvan eller i båda halvorna genom att passera mittpunkten. MSAS för hela input är därmed $max(MSAS^L, MSAS^R, (\text{MSAS som korsar mittpunkten}))$, där $MSAS^L$ och $MSAS^R$ representerar MSAS för endast den vänstra respektive högra halvan av input. MSAS för vänster och höger halva kan enkelt beräknas rekursivt genom samma algoritm ner till basfallet $n = 2$, där MSAS helt enkelt är lika med $max(L, R, L + R)$.

Utmaningen ligger därmed i hur man beräknar den MSAS som täcker båda halvor av input och alltså "korsar" mittpunkten, d.v.s. den MSAS sådan att de två element närmast mitten[2] alltid är inkluderade. Om vi fortsätter i D&C-tankesättet är det rimligt att dela upp detta i två delar, så att MSAS som korsar mittpunkten blir summan $MSAS^L_R + MSAS^R_L$, där $MSAS^H_K$ är MSAS i halvan H sådan att det element som ligger på kant K alltid är inkluderat. Se illustrationen nedan.



I högra halvan av illustrationen tydliggörs varför dessa kantvärden är viktiga, då $MSAS^R = 5$ medan $MSAS^R_L = 4$. Eftersom summan av *alla* tal från start till slut måste inkluderas i MSAS kan det bli som i exemplet där $MSAS^L = 11 > MSAS^R = 5$ men $MSAS^L_R + MSAS^R_L = 15$. 5an är störst "själv" i högra halvan men genom att "bygga en bro" till mitten kan den inkluderas i den stora sekvensen i vänsterhalvan och ändå bidra positivt till den totala summan.

Vi får då titta på hur man kan beräkna $MSAS_L$ respektive $MSAS_R$, så att dessa värden kan hämtas rekursivt från lägre nivåer av rekursionen ner till basfallet $n = 2$. I basfallet är beräkningen enkel, då det endast finns tre möjliga kombinationer av två element, samt för $MSAS_L$, exempelvis, *måste* det vänstra elementet inkluderas (den sublista med endast det högra elementet måste exkluderas). Detta ger $MSAS_L(L, R) = max(L, L + R)$ respektive $MSAS_R(L, R) = max(R, L + R)$. I alla fall högre än basfallet ($n \geq 4, 2 \mid n$) kan det verka svårare, men en viktig observation kan göras: eftersom något av kantelementen måste inkluderas, kommer samtliga element från den ena kanten till någon punkt i listan inkluderas. Eftersom den punkten kan vara var som helst i listan kommer den vara i antingen den högra eller vänstra halvan av listan. För halvan H ger detta endast två möjliga utfall: slutpunkten ligger i $\begin{cases} H & \Rightarrow MSAS_H = MSAS^H_H \\ \neg H & \Rightarrow MSAS_H = MSAS^{\neg H}_H + SUM^H \end{cases}$ Där $\neg H$ är den andra halvan (inte H) och SUM_H är summan av alla element i H.

Detta är för att om slutelementet för $MSAS_H$ inte ligger i H, *måste* mitten vara inkluderad i $MSAS_H$, och därmed måste *alla element i* H vara inkluderade i $MSAS_H$. De element som "hänger med" från $\neg H$ kommer enligt definitionen av MSAS vara den största möjliga sublistan i $\neg H$ som inkluderar mittenelementet, vilket per definition är $MSAS^{\neg H}_H$. Om slutelementet inte passerar mitten betyder det att $MSAS_H$ är den största summan av element endast i H som inkluderar det yttersta elementet, dvs $MSAS^H_H$. Eftersom båda dessa utfall är möjliga beräknas båda och det största väljs: $MSAS_H = max(MSAS^H_H, MSAS^{\neg H}_H + SUM^H)$.

För att rekursionen ska fungera krävs alltså att vi beräknar $\{MSAS, MSAS_L, MSAS_R, SUM\}$ vid varje nivå.

[2]
$\because n = 2^k, k \in Z^+ \Rightarrow 2 \mid n$
$\therefore$ 'Mittpunkten' ligger mellan två element

Funktionen definieras då som följande:

$$\begin{cases} \text{basfall } (n=2): & \begin{cases} MSAS &= max(L, R, L+R) \\ MSAS_L &= max(L, L+R) \\ MSAS_R &= max(R, L+R) \\ SUM &= L+R \end{cases} \\ \text{övriga fall } (n>2): & \begin{cases} MSAS &= max(MSAS^L, MSAS^R, MSAS_R^L + MSAS_L^R) \\ MSAS_L &= max(MSAS_L^L, MSAS_L^R + SUM^L) \\ MSAS_R &= max(MSAS_R^R, MSAS_R^L + SUM^R) \\ SUM &= SUM^L + SUM^R \end{cases} \end{cases}$$

där värdena för vänster/höger halva fås genom att köra funktionen rekursivt på dessa halvor. Eftersom vi nu täcker alla möjliga fall och alltid returnerar det största möjliga värdet av MSAS, måste algoritmen vara korrekt för alla $n = 2^k, k \in Z^+$.

Vår Python-implementation exekverar exakt $(n-1) \cdot 5$ jämförelser (if-statements) för en input av storlek n. På ett liknande sätt som i del 3.1.2 gör vi nu 5 jämförelser per exekvering av funktionen, så det som är relevant är återigen totala antalet exekveringar av funktionen. Vi tittar på mönstret:

$n = 2 \Rightarrow 1$ exekvering (basfallet)
$n = 4 \Rightarrow 3$ exekveringar (1 merge + 2 basfall)
$n = 8 \Rightarrow 7$ exekveringar (1 merge + 2 merges + 4 basfall)
...

Alltså är antalet exekveringar lika med $n-1$ och därmed antalet jämförelser lika med $5 \cdot (n-1)$. Detta kan bevisas med ett induktionsbevis eller väldigt enkelt genom att titta på en klassisk geometrisk serie: $\sum_{i=1}^{\infty} 2^{-i} = 1/2 + 1/4 + 1/8 + \ldots = 1$. Om vi tittar på mönstret ovan kan vi snabbt se att antalet exekveringar liknar den geometriska serien. För exemplet $n = 8$ kan vi se att antalet exekveringar blir:

$$8/8 + 8/4 + 8/2 = 8 \cdot (1/2 + 1/4 + 1/8) = n \cdot (1/2 + 1/4 + 1/8) \approx n$$

Anledningen till att det inte blir exakt n är för att vi inte följer den geometriska serien helt, utan tar bort alla termer < 1 (vi går ju bara ner till basfallet som körs en gång och inte "delar" på sig själv mer). Vi kan alltså skriva antalet exekveringar som:

$$n \cdot (1/2 + 1/4 + 1/8 + \ldots) - (1/2 + 1/4 + 1/8 + \ldots) = n \cdot \sum_{i=1}^{\infty} 2^{-i} - \sum_{i=1}^{\infty} 2^{-i} = n \cdot 1 - 1 = n - 1$$

Slutligen görs 5 jämförelser per exekvering av funktionen och det totala antalet jämförelser blir då $5 \cdot (n-1)$.