IDATA2302 Algoritmer og Datastrukturer

Hans Georg Schaathun

Ordinær eksamen 25. mai 2021
Revidert 14. juni 2021

Sensorvegleiing 25. mai 2021 **IDATA2302 Algoritmer og Datastrukturer**
Eksamenstid $4\frac{1}{2}$ timar *Nynorsk*

1. Me har to *array*ar A og B som kvar inneheld n heiltal.

   (a) Lag ein effektiv algoritme for å sjekka om der finst eit element e som er innehalde i den eine *array*en og ikkje i den andre (dvs. $e \in A - B$ eller $e \in B - A$). Algoritmen skal ha køyretidskompleksitet $O(n \cdot \log n)$. Forklar kort kvifor algoritmen er korrekt.

   Solution: The time complexity gives us time to sort both the arrays using say merge sort or heap sort. (QuickSort may be better in practice but does not satisfy the requirement in the worst case.)
   If there is no element e in one but not both arrays, it means that A and B are identical. If e exists, there is an index i so that $A[i] \neq B[i]$. This is checked by a linear search, in linear time, as follows.

```
for i = 1...n, do
  if A[i] ≠ B[i],
      return True
return False
```

   (b) Lag ein annan effektiv algoritme som finn ein *array* C som omfattar m distinkte element som utgjer unionen av A og B. Forklar kort kvifor algoritmen er korrekt.

   Solution: This is similar to the previous algorithm. First, we sort the arrays, and then we merge the two arrays dropping duplicates as we merge. The latter is essentially the merge subroutine from merge sort, but it must be tweaked to drop duplicates.

```
a := 1 // index for A
b := 1 // index for B
c := 0 // index for C
while a ≤ n and b ≤ n
  if A[a] = B[b],
      c := c + 1
      C[c] = A[a]
      a := a + 1
      b := b + 1
  else if A[a] < B[b],
      c := c + 1
      C[c] = A[a]
      a := a + 1
  else if A[a] > B[b],
      c := c + 1
      C[c] = B[b]
      b := b + 1
while a ≤ n
      c := c + 1
      C[c] = A[a]
      a := a + 1
while b ≤ n
      c := c + 1
      C[c] := B[b]
      b := b + 1
return C
```

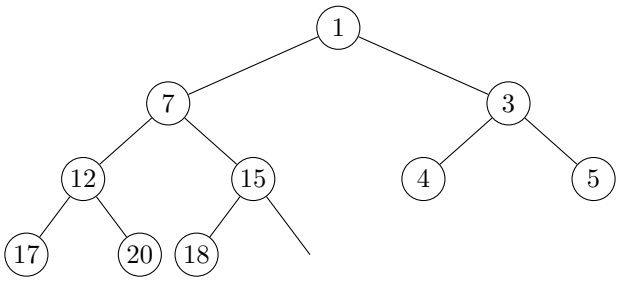
   (c) Finn køyretidskompleksiteten åt algoritmen.

Solution: The merging simply loops through the $2n$ elements, which gives a run time complexity of $\Theta(n)$. Thus the sorting dominates the run time, and we can choose between algorithms with worst case complexity $O(n \log n)$ such as MergeSort, or algorithms which tend to do better in practice in spite of $\Theta(n^2)$ worst case complexity such as QuickSort. If the integers in the arrays are constrained, RadixSort may be able to sort in linear time.

(d) Drøft kort køyretidskompleksiteten. Er det råd å løysa problemet asymptotisk raskare, anten i verste fall eller i gjennomsnitt?

Solution: We have already discussed the tuning possible by the choice of sorting algorithm. To improve any further we would need to do away with the sorting.

One way or another, the algorithm has to identify duplicates and omit them from the output. If the arrays are not sorted, this would require a linear search in one array, for every element of the other array, i.e. n linear searches each of complexity $\Theta(n)$, for a total of $\Theta(n^2)$. This is worse than what we achieve by sorting first, and we conclude that the algorithm cannot be improved any further.



Figur 1: A heap for Question 2.

2. Figur 1 viser ei dyngje (*heap*), visualisert som eit binærtre.

(a) Forklar kva dyngjeeigenskapen er, og vis at datastrukturen i figuren tilfredsstiller denne eigenskapen.

Solution: The heap property is the property that for every subtree, the root element is smaller (resp. larger for a Max Heap) than any of its descendants.

It can be proved (by mathematical inducation) that it suffices to check that each element is smaller than each of its children. In the heap in the figure, we can quickly check each element and see that both children are larger. Hence the heap property is satisfied.

(b) Forklar kort kvifor dyngjeeigenskapen kan vera nyttig i praktisk bruk, gjerne med døme.

Solution: A heap is also known as a priority queue, where the elements in the queue should be served, not on a first come first served basis, but based on some heuristic which is used as key to calculate smaller and larger in the queue.

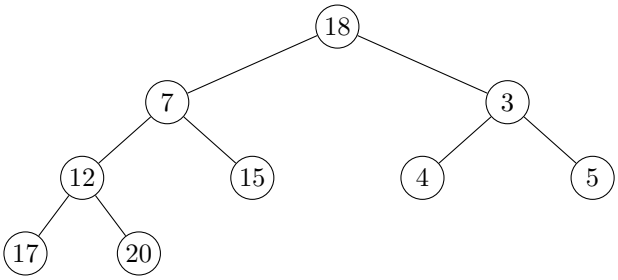
This is used in task scheduling in multitasking operating systems and batch systems, where many different factors should influence the priority, including type (user or system task), time since last execution, 'nice' value set by the user. A similar situation can be found in customer management systems, where one want to give premium customers prompter service than regular customers.

We also see the heap used in many algorithms, particularly optimisation problems, such as Dijkstra's shortest path algorithm.

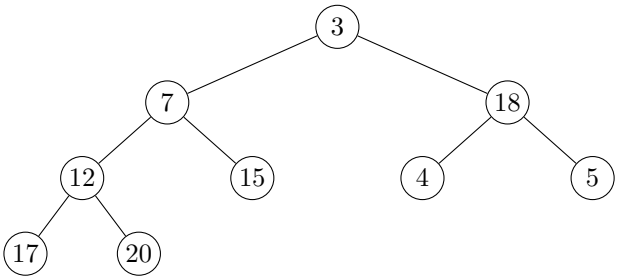
(This should not be a competition in finding as many and as elaborate examples as possible. A top student should be expected to find two rather different examples (e.g. task scheduler and Dijkstra), but one example is very good.)

(c) Gå ut frå at me skal henta ut det minste elementet frå dyngja. Vis, steg for steg, korleis dyngja vert endra for å gjenoppretta dyngjeeigenskapen når det minste elementet vert fjerna. Bruk ein generell og effektiv algoritme.

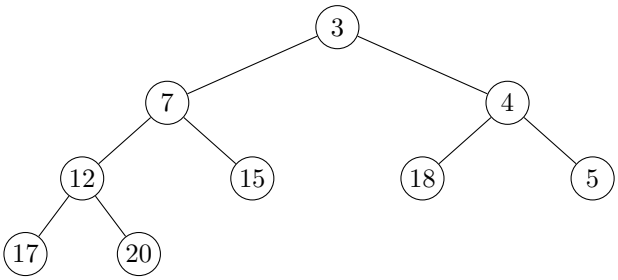
Solution: Step 1. Remove the root element, and replace it with the last element in the tree (reading top-down, left-right).



Step 2. Push the large element down the heap, until the heap property is satisfied. Here the right hand child is the new smallest element.



Step 3. Repeat. Now the left child is the new smallest element.



Done. The heap property has been restored.

(d) Gjev pseudokode for ExtractMin-algoritmen som du brukte i førre oppgåve (inklusive hjelperutinar).

Solution: The pseudo-code is easiest to write if we represent the heap as an array, where the children of item i are items $2i$ and $2i+1$. This is to get easy access to the last element in the heap.

Algorithm ExtractMin(A)
r := A[1] // **return** value
$A[1] := A[A.\mathsf{size}]$
$\mathsf{size} := \mathsf{size} - 1$
\textsc{MinHeapify}(A,1)

```
Algorithm MinHeapify(A, i)
l = 2i
r = 2i + 1
if l ≤ A.size and A[l] > A[i], then
    m := l
else
    m := i
if r ≤ A.size and A[r] > A[m], then
    m := r
if m ≠ i, then
  swap A[i] and A[m]
  \textsc{MinHeapify}(A,m)
```

(e) Kva er køyretidskompleksiteten åt ExtractMin()?

Solution: MinHeapify recurses for each level of the tree, i.e. $\Theta(\log n)$ calls. Except for the recursive call, every instruction takes constant time. Thus the complexity is $\Theta(\log n)$.

In ExtractMin, the only instruction which takes more than constant time is MinHeapify. Hence ExtractMin also has complexity $\Theta(\log n)$.

3. Sett at me har ei usortert liste a med n element. Me skal finna det $i$te største elementet. Dersom me sorterer lista i synkande orden fyrst, kan me sjølvsagt henta ut det $i$te elementet relativt enkelt.

(a). Kva er køyretidskompleksiteten på denne naïve løysinga inklusive sortering? Grunngje svaret kort.

Solution: Sortering i mogleg i $O(n \log n)$, ogso i verste fall. Berre med spesielle føresetnader er det mogleg å sortera raskare.

Når ein fyrst har sortert, kan ein slå opp det $i$te elementet i konstant tid (i ein *array*) eller i lineær tid (i ei liste). Uansett er dette neglisjerbart samanlikna med sorteringa.

Det tek altso $\Theta(n \log n)$.

(b). Der er fleire moglege løysingar som gjev lågare kompleksitet (anten i verste fall eller i gjennomsnitt). Gje pseudo-kode for minst éi og forklar kvifor ho gjev korrekt svar.

Solution: (Her er det sannsynleg at studentane søkjer seg fram til algoritmar som dei ikkje har studert på førehand, og ein kan ikkje venta at dei vel den same algoritmen. Det viktigste er at dei kan vurdera algoritmen sjølv.

Den enklaste algoritmen med utgangspunkt i pensum er QuickSelect.)

Me skal finna det i største elementet i ei usortert liste. QuickSelect byggjer på QuickSort, men utnyttar det faktumet at ein ikkje treng å sortera den delen av lista der ein ikkje finn det $i$te elementet.

Lat $A_{1..n}$ vera ein usortert array, og i som brukt over.

```
QuickSelect(A_{1..n}, i)
if n = 1, return A_1.
Pick a random pivot p ∈ A
Sort (in linear time) the elements of A into three groups
  L = {x > p | x ∈ A}
  M = {x = p | x ∈ A}
```

$R = \{x < p | x \in A\}$
if $k \leq |L|$ **return** QuickSelect(L, i)
else if $k \leq |L| + |M|$ **return** p
else return p QuickSelect$(R, i - |L| - |R|)$

Me ser at dei $|L|$ største elementa hamnar i L, slik at dersom $i \leq |L|$ er det berre i L me treng søkja vidare. Dei neste $|M|$ elementa hamnar i M, so dersom i ligg til venstre for det siste elementet i M, er det i M me skal leita, men alle elementa i M er like, so me kan ta einkvan. Dersom me må leita i den høgre mengda, R, skal me leita etter $(i + |L| + |M|)te$ største, fordi der er større element i dei to venstre mengdene.

(c). Vurder kompleksiteten på løysinga som du har vald.

Solution: (Her er det usannsynleg at studentane klarer å gjera ein fullstendig, formell analyse, sjølv om ein kan finna han i læreboka.)

Me kan samanlikna med QuickSort. Me veit at QuickSort har $O(n \log n)$ i gjennomsnitt (og i beste fall) fordi ein i gjennomsnitt vil om lag halvera n for kvar rekursjon og dermed få $\log n$ rekursjonar totalt. I kvar steg må ein sortera n element, som gjev $n \log n$ operasjonar totalt.

I QuickSelect har me omtrent same prinsipp, bortsett frå at me berre treng å sortera halvparten av elementa i kvar runde (ideelt). I det ideelle tilfellet endar me då opp med

$$n \cdot (1 + \frac{1}{2} + \frac{1}{4} + \cdots \frac{1}{2^n} < 2n = O(n).$$

Det er rimeleg å tru at QuickSelect er lineær i gjennomsnitt, etter same argument som for at QuickSort er $O(n \log n)$ i gjennomsnitt.

(Fullstendig prov finst i læreboka, men er ikkje gjennomgått på førelesing. Det vesentlege er at studentane ser samanhengen med og forbetringa over QuickSort.)

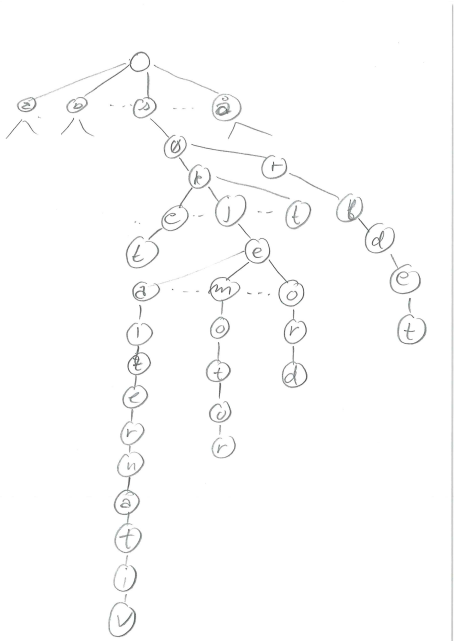
4. Gje eit grovdesign til ein søkjemotor basert på søkjetre, dvs. sokalla *tries*. Søkjemotoren vil indeksera sidene på førehand. Når brukaren søkjer på eit ord, skal systemet returnera ei liste med sider sortert etter relevans. Ta med fylgjande:

   (a). Ein kort drøfting av moglege søkjeheuristikkar. Korleis måler me relevansen for ei side?

   Solution: Dette spørsmålet er ein utfordring i praksis, og kandidaten må fyrst og framst visa at han er kjend med utfordringa. Det mest typiske heuristikkane talet på førekomstar av ordet på sida og kor ofte og/eller kor nylig andre henter sida. Det fyrste er lett å implementera. Bruksstatistikkar kan vera, men er ikkje alltid tilgjengelege.

   (b). Skildring av datastrukturen, gjerne vha. klasse- og objektdiagram.

   Solution:



Ein *trie* er eit søkjetre der kvar node er ein bokstav og kvar stig frå roten til eit løv representerer eit ord, stava med bokstavane på stigen. Kvar node kan ha inntil eitt barn per bokstav i alfabetet.

(c). Algoritmen for å byggja indeksen, med køyretidskompleksitet og argumentasjon.

Solution: Lat $s = s_1 s_2 \dots s_n$ vera eit søkjeord, t rotnoden i søkjetreet og p ei side som skal indekserast.

Algorithm TrieInsert(t, s, p)
while $s \neq$
  if t has child c with label s_1 then
    $t := c$
  else
    $t :=$ new child of t with label s_1
  remove first letter from s
Add p to t

Algoritmen fylgjer stigen som søkjeordet definerer og oppretter nye nodar der dei manglar. Analysen fylgjer dermed analysen for neste algoritme.

Pseudokoden føreset at mengda av alle søkjeord som vert brukte er prefiksfri og der er ingen feilsjekk i denne versjonen. Den enklaste måten å løysa dette problemet er (truleg) å innføra eit ekstra symbol som markerer slutt på ord og aldri finst i andre posisjonar.

(d). Algoritmen for å finna ei side, køyretidskompleksitet og argumentasjon.

Solution: Lat $s = s_1 s_2 \dots s_n$ vera eit søkjeord og t rotnoden i søkjetreet

Algorithm TrieFind(t, s)
while $s \neq$
  if t has child c with label s_1 then
    $t := c$
  else

```
      return ∅
    remove first letter from s
  if t is a leaf, then
    return contents of t
  else
    return ∅
```

Algoritmen fylgjer den stigen representert ved s, og implementerer dermed avbildinga etter definisjonen. Dersom ein bokstav manglar vil han returnera ei tom mengd. Dersom han finn heile strengen er der to alternativ. Dersom han ender opp i eit løv returnerer han innhaldet i løvet som representerer søkjeordet. Elles er søkjeordet berre ein prefiks av andre søkjeord.

Køyretida er lineær i lengda på søkjeordet (talet på iterasjonar i *while*-løkka). Ho kan vera konstant eller lineær i talet på bokstavar i alfabetet (born per node) avhengig av underliggjande implementasjon.

Vurderingsrubrikk

Rubrikken vert delt til studentane før eksamen.
Får å stå på eksamen må ein ha **minst eitt (1) poeng på kvart kriterium og minst seks (6) poeng totalt.**
Ved vurdering av validering og kompleksitetsanalyse legg me i hovudsak vekt på den oppgåva som er best løyst.
Andre oppgåver vert evt. vektlagde i tvilsfall.

Kriterium Poeng	Ikkje tilfredsstillande 0 poeng	Tilfredsstillande 1 poeng	Bra svar 2 poeng	Perfekt 3 poeng
Presentasjon	Svaret er blankt, usamanhengande eller tvetydig og gjev ikkje noko klart inntrykk av algoritmen.	Algoritmen er tilstrekkeleg formelt presentert som pseudo-kode, slik at hovudprinsippa kjem klart fram, sjølv om detaljane er skjult av usamanhengande eller upresise formuleringar.	Løysinga er overtydande og utvetydig presentert som pseudo-kode, sjølv om småfeil kan førekoma.	Lytefritt.
Validering	Ingen av dei mest vesentlege poenga kjem tydeleg fram i svaret	Dei kritiske idéane til eit bevis er korrekt identifiserte og presentasjonen er eit godt utgangspunkt for diskusjon	Valideringa er overtydande og lemnar ingen særleg tvil om at algoritmen er korrekt	Feilfritt logisk bevis
Kompleksitetsanalyse	Ingen av dei mest vesentlege poenga kjem tydeleg fram i svaret	Kritiske idéar er identifiserte og presentasjonen er eit godt utgangspunkt for diskusjon	Korrekt køyretid er funnen og analysen og lemnar ingen særleg tvil om at han er korrekt	Feilfritt logisk bevis
Algoritmisk løysing	Ingen presenterte algoritmar løyser problemet fullstendig	Nokon av oppgåvene er løyste hovudsakleg korrekte	Fleirtalet av oppgåvene er løyste hovudsakleg korrekt.	Korrekte algoritmar/-datastrukturar er gjevne på alle oppgåvene.