582359 Algoritmit ongelmanratkaisussa (kevät 2013)
Viikon 11 ratkaisuja

1. Tehtävänä on tarkistaa, onko merkkijonon A osana merkkijonoa B. Oletetaan, että A:ssa on n merkkiä ja B:ssä on m merkkiä. Toteuta tehtävään brute-force-algoritmi, jonka aikavaativuus on $O(nm)$.

   Suunnittele kaksi syötettä, joissa $n = 1000000$ ja $m = 10000$. Valitse syötteet niin, että toisessa tapauksessa algoritmi on hidas ja toisessa salamannopea.

   Ratkaisu:

   Tiedostossa `Merkkijonohaku1.java` on brute-force-algoritmin toteutus. Algoritmi käy kaikki A:n merkit läpi ja yrittää täsmätä mahdollisimman monta B:n merkkiä joka kohdasta aloittaen.

   Valitaan molemmissa syötteissä $A =$ `aaa...a` eli merkkijono, jossa on pelkkää merkkiä `a`. Algoritmi toimii nopeasti kun $B =$ `bbb...b`. Tällöin jokaisessa kohdassa B:n vertailu päättyy heti, koska ensimmäinen merkki on väärä. Algoritmi toimii hitaasti kun $B =$ `aaa...ab`, koska nyt algoritmi joutuu tarkistamaan jokaisessa kohdassa kaikki `a`:t, kunnes viimeinen `b` ei täsmää.

2. Toteuta samaan tehtävään haluamasi tehokas $O(n + m)$-aikainen algoritmi. Kuinka nopeasti se käsittelee valitsemasi syötteet?

   Ratkaisu:

   Tiedostossa `Merkkijonohaku2.java` on Knuth-Morris-Pratt-algoritmin toteutus. Algoritmi vastaa muuten brute-force-algoritmia, mutta kun B ei täsmää tiettyyn kohtaan, niin sitä siirretään eteenpäin ensimmäiseen seuraavaan kohtaan, johon B:n alkuosa täsmää.

   Tiedostossa `Merkkijonohaku3.java` on vertailun vuoksi Javan valmista metodia `contains` käyttävä ohjelma. Javan `contains` tuntuu olevan yhtä hidas kuin itse tehty brute-force-algoritmi.

3. (a) Laske merkkijonojen KØBENHAVN ja KÖÖPENHAMINA editointietäisyys.

   (b) Toteuta editointietäisyyden laskenta dynaamisella ohjelmoinnilla. Millainen taulukko syntyy kohdan (a) tapauksessa?

   Ratkaisu:

   (a) Editointietäisyys on 6. Esimerkiksi: KØBENHAVN → KÖBENHAVN → KÖÖBENHAVN → KÖÖPENHAVN → KÖÖPENHAMN → KÖÖPENHAMIN → KÖÖPENHAMINA.

   (b) Tiedostossa `Editointietaisyys.java` on editointietäisyyden laskeva ohjelma. Kohdan (a) tapauksessa syntyy seuraava taulukko:

	K	Ö	Ö	P	E	N	H	A	M	I	N	A
K	0	1	2	3	4	5	6	7	8	9	10	11
Ø	1	1	2	3	4	5	6	7	8	9	10	11
B	2	2	2	3	4	5	6	7	8	9	10	11
E	3	3	3	3	3	4	5	6	7	8	9	10
N	4	4	4	4	4	3	4	5	6	7	8	9
H	5	5	5	5	5	4	3	4	5	6	7	8
A	6	6	6	6	6	5	4	3	4	5	6	7
V	7	7	7	7	7	6	5	4	4	5	6	7
N	8	8	8	8	8	7	6	5	5	5	5	6

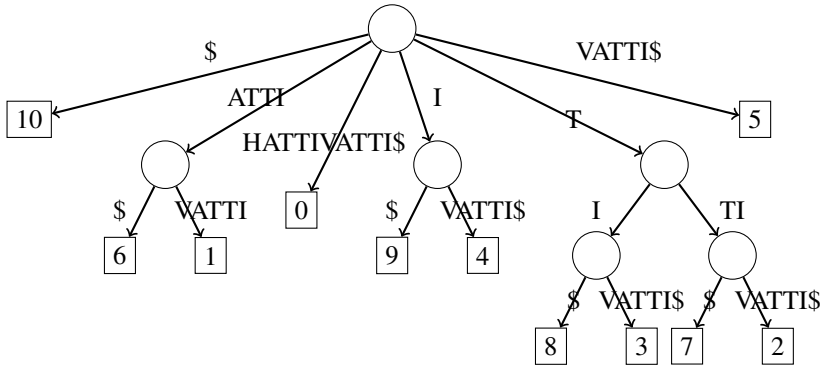
4. Muodosta merkkijonon HATTIVATTI suffiksipuu ja suffiksitaulukko.

   Mitä sovelluksia rakenteilla on ja mitä hyviä ja huonoja puolia niissä on toisiinsa verrattuna?

   Ratkaisu:

   Lisätään merkkijonoon lopetusmerkki \$, jolloin tuloksena on HATTIVATTI\$.

Merkkijonon suffiksipuu:



Merkkijonon suffiksitaulukko:

10	6	1	0	9	4	8	3	7	2	5

5. Palindromi on merkkijono, joka pysyy samana, vaikka sen kääntäisi väärinpäin. Suunnittele brute-force-algoritmi, joka etsii pisimmän palindromin merkkijonon sisällä. Mikä on algoritmin aikavaativuus? Millaiset syötteet ovat algoritmille hankalia?

   Ratkaisu:

   Yksi mahdollisuus on käydä läpi kaikki merkkijonon osajonot ja tarkistaa jokaisesta, onko se palindromi. Tämän ratkaisun aikavaativuus on $O(n^3)$. Ratkaisua voi tehostaa käymällä haun edetessä läpi vain osajonoja, joiden pituus on pidempi kuin pisin tähän mennessä löydetty palindromi. Tällöin haku toimii nopeasti, jos heti alussa löytyy pitkä palindromi, mutta muuten haku on hidas.

   Tehokkaampi menetelmä on etsiä palindromia keskikohdasta lähtien. Nyt riittää käydä läpi kaikki mahdolliset keskikohdat ja tutkia jokaisesta, kuinka leveäksi palindromin saa muodostettua. Tämän algoritmin aikavaativuus on $O(n^2)$. Käytännössä algoritmi toimii usein tehokkaasti, mutta jos merkkijonossa on paljon pitkiä palindromeja, niin algoritmi on hidas.

6. (a) Hahmottele suffiksipuuta käyttävä tehokas ratkaisu pisimmän palindromin etsimiseen.

   (b) Suffiksipuun heikkoutena on sen monimutkaisuus. Suunnittele yksinkertainen $O(n)$-aikainen algoritmi pisimmän palindromin etsimiseen.

   Ratkaisu:

   (a) Puussa kahden solmun alin yhteinen vanhempi on sellainen solmu, josta pääsee kumpaankin solmuun ja joka on mahdollisimman syvällä puussa. Suffiksipuussa kahden lehtisolmun alin yhteinen vanhempi on solmu, joka vastaa kyseisissä kohdissa alkavien suffiksien yhteistä alkuosaa. Alimman yhteisen vanhemman selvittäminen on mahdollista tehokkaasti sopivan esikäsittelyn jälkeen. Tehdään suffiksipuu merkkijonosta $T\#T'\$$, jossa T on alkuperäinen merkkijono ja T' on sama merkkijono käännettynä. Nyt riittää käydä läpi kaikki palindromin keskikohdat ja laskea, mikä on T:ssä ja T':ssä kyseisten kohtien alin yhteinen vanhempi. Tämä on puolet pisimmän palindromin pituudesta kyseisessä kohdassa.

   (b) Tehdään $O(n^2)$-aikaiseen brute-force-ratkaisuun seuraava tehostus: Tallennetaan jokaisen merkin kohdalle pisimmän palindromin pituus, jonka keskikohtana on kyseinen merkki. Lisäksi pidetään kirjaa, mikä on tällä hetkellä pisimmälle merkkijonossa yltävän palindromin sijainti. Nyt jos tutkittava keskikohta on pisimmälle yltävän palindromin sisällä, alkuarvo palindromin pituudelle saadaan lukemalla tallennettu palindromin pituus pisimmälle yltävän palindromin keskikohdan toiselta puolelta. Tämän ansiosta hakua voi jatkaa suoraan vielä tutkimattomasta merkkijonon osasta (jos palindromin alkuosa on kunnossa), jolloin aikavaativuudeksi tulee $O(n)$.