Övning 4 - Tillämpad datalogi 2013

```
0 # coding: latin
```

courier times

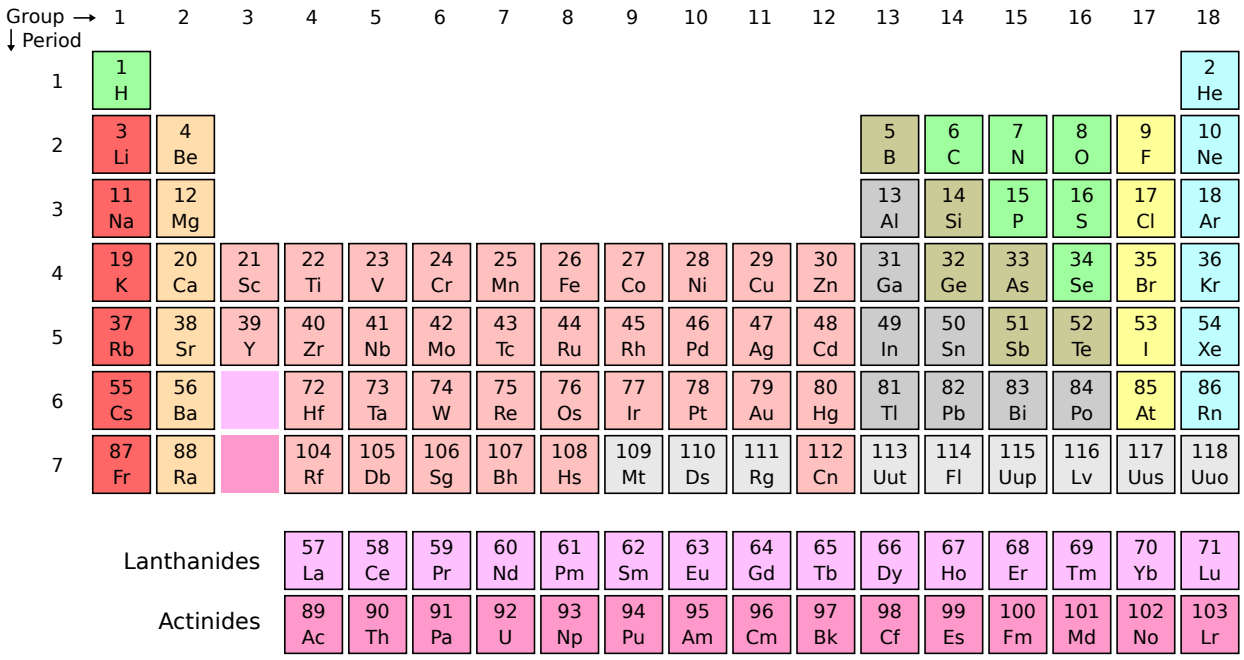
Sammanfattning

Idag gick vi igenom problem som handlade om **lagring** och **sortering**. Vi löste uppgifter som handlade om **hashning**, **binärträd**, **bloomfilter**, **trappor (heap)**, **komplexitet** och **sortering**. Vi gick igenom talen 1-5. Jag rekommenderar att kolla på de återstående talen 6-10. Återkomm gärna till mig med frågor.

Hashning, sortering, prioritetskö, bästaförstsökning

1. Perfekt hashfunktion

Hitta på en perfekt hashfunktion för atomer. Hur stor blir hashtabellen?



En perfekt hashfunktion mappar unikt alla hashvärden till en unik plats i en tabell, dvs det får inte förekomma några kollisione Vi antar

1. inga atombeteckningar är längre än två tecken, dvs tillfälliga namn för syntetiska transuraner som Ununtrium (Uut), Ununquadium (UUq), etc ignoreras
2. algoritmen ska klara alla möjliga tvåteckens-koder (dvs även "Ad" och "J" mfl, som inte finns)

Vi vill nu konvertera bokstäverna i an atombeteckning till ett hel tal. Låt oss vikta varje bokstav så att vi kan undvika att AB ger samma värde som BA. Antag att vi låter första bokstaven i namnet (versalen) viktas med 100, dvs.
Bokstav 1
$A = 1 * 100; B = 2 * 100, ..., Z = 26 * 100$

och ev. andra bokstav av heltalen
Bokstav 2

$a = 1; b = 2; c = 3, ... ,z = 26$
Då kan vi skapa hashkoden genom att lägga ihop siffrorna: Ag = 100 + 7. Det är dock onödigt att använda hundratal. Det blir ju många siffror som vi aldrig kommer att få 0, 27-99, 127-199, ..., 2527-2599. Därför låt oss ändra hashkoden för bokstav 1 till (i-1)*27 där i=nummer på bokstav 1-26. Vi multiplicerar med 27 därför att det finnd 27 kombinationer av A, Aa, ..., Az: Vi får:
Bokstav 1
$A = 0; B = 27; C = 54, ..., Z = 675 (25 * 27)$

Bokstav 2
$a = 1; b = 2, ..., z = 26$

$h(A) = 0; h(Ag) = 7(0+7); h(Zz) = 701 (25 * 27 + 26 = 675 + 26)$ Så att "minsta" möjliga atom-hashkod 0, och största möjliga 701. Med 702 platser (6 ggr antal element) så blir det inga krockar, och ingen luft förutom det som orsakas av att element inte finns... *I labb 5 ska ni inte göra en så här stor tabell utan istället använda kvadratisk probning.*

2. Håll reda på media (Tildatenta 030308)

Under gulfkriget var det väldigt svårt för arméstaben att hålla reda på alla TV-bolag som for omkring och rapporterade i öknen. För att hålla reda på dem användes en hashvektor. Koden fungerade inte som avsett och man har nu gett i uppdrag åt en f.d. tildastudent att titta på en misstänkt del av koden:

```
79 from string import find
80 p = 100;
81 hash_vector = [0]*p
82 alfabet = "abcdefghijklmnopqrstuvxyz"
83
84
85 def put(name, company):
86     hash_code = 0
87
88     # Skapa ett heltalsvärde för objektet
89     for i in range(len(name)):
90
91         # Tar ut bokstavens position if alfabetet
92         pos = find(alfabet, name[i])
93         hash_code += pos
94
95     # Modulus längden av hashvektorn
96     hash_code = hash_code % p
97
98     # Placera objektet i hashvektorn
99     hash_vector[hashcode] = company
```

Vad är det för fel på koden? Beskriv hur man kan förbättra den. Namnen på TV-bolagen kan antas bestå av högst tre bokstäver. Det kommer inte att förekomma mer än 75 TV-bolag.

Problem: Saknas krockhantering, och samma bokstäver ger samma kod: hash("B") = hash("AA"). hash("AB") = hash("BA"). hash("ABC") = hash("CBA").

Lösningar:
Krocklistor - varje element i hashvektorn är en lista.

Linjär probning - ifall positionen är upptagen, gör en rehash till **(pos+skip)** vi hittar en tom plats eller kommer tillbaka till ursprungsvärdet. Skip är ett konstant värde. Finns även kvadratisk probning, då ändrar vi skip för varje rehash; 1,3 5,7, osv

Hashvektorns längd - Större hashvektor vore bättre, och dessutom bör längden vara primtal. Försäkrar att alla positioner i talbellen kommer at besökas vid kockhantering. (Går att visa matematiskt att det ger bättre spridning.) Lämplig längd är 151, ett primtal dubbelt så stort som förväntat antal element.

Vikta mot bokstavsposition En bättre hashkod skulle man få genom att vikta varje bokstav vid uträkningen beroende på posision i strängen. T.ex. med pos som värdet på bokstaven (0-25), då kan vi vikta första bokstaven genom at multiplicera pos med 1, andra genom att multiplicera pos+1 med 26 och för tredje pos+1 med 26^2. Då kan **bokstav 1** bara ta värden mellan 0-25, **bokstav 2** mellan $26, 26*2, ..., 26*25$ och **bokstav 3** $26^2, 26^2*2, ..., 26^2*25$.
Det ger att

$A \Rightarrow 0\ (0+0*1)$
$B \Rightarrow 1\ (0+1*1)$
$C \Rightarrow 2\ (0+2*1)$
...
$Z \Rightarrow 25\ (0+25*1)$
AA $\Rightarrow 26\ (0+1*26+0*26^2)$
...
$ZA \Rightarrow 51\ (25+1*26+0*26^2)$
...
$BA \Rightarrow 52\ (0+2*26)$
...
$AAA \Rightarrow 702\ (0+1*26+1*26^2)$
...
$ZZZ \Rightarrow 18277\ (25+26*26+26*26^2)$
Så nu kan aldrig två nummer anta samma värde. (Modulo-operationen gör att man ändå hamnar inom vektorn.)
Vi får hashfunktionen: $h(ord) = nr_{bokstav1} + (nr_{bokstav2})*26 + (nr_{bokstav3})*26^2$. där ord =bokstav1+bokstav2+bokstav3.

3. Nix till telefonförsäljning (Tildatenta 000603)

Föreningen för konsumentskydd vid marknadsföring per telefon har startat ett register dit den som inte vill bli uppringd av telefonförsäljare kan anmäla sig.
Till att börja kommer kontrollen att ske genom att företaget sänder sin telefonlista till nix och får tillbaka en lista där de nixade numren markerats. ***Vilka av följande metoder kan föreningen använda sig av? Vilken är bäst?***

Binärträd
Bloomfilter
Hashtabell

Ett framtida mål är att kontroll också skall kunna ske över internet. Då måste kontrollen ske snabbt men man vill också försäkra sig om att ingen ska kunna få ut en lista över alla nixade telefonnummer.
Vilken metod passar bäst för internet-kontrollen?

OBS: Just nu ringer många och anmäler sig till NIX-registret så det måste gå lätt att lägga till nya.

Binärträd går snabbt att söka i men man måste se till att det inte blir obalanserat när nya telefonnummer läggs till.

Bloomfilter finns det risk för falska positiva. Går inte att ta bort nummer utan att göra om tabellen. Bestämma storleken på förhand. Besvärligt att stoppa in nya nummer i.

Hashtabell blir inte obalanserad som ett träd, men ifall man får många kollisioner (dålig hashfunktion eller inte tillräckligt stor tabell) så blir den mycket oeffektiv.

Man kan räkna med att många har registrerat sig så snabb sökning är nödvändig. Trots att tre alternativen är snabba, faller trädet bort eftersom det går snabbare att söka i ett bloomfilter och hashtabell än ett träd (O(1) vs O(log (n)), sen faller hashtabellen bort eftersom den inte uppfyller kravet för en framtida internet-kontroll om att det inte ska gå att få fram telefonnmmer som finns lagrade i tabellen genom att kolla i den. (I verkligheten måste man dock även kunna ta bort nummer efter ett antal år. Det gör att Bloomfilter inte är lämpligt ifall man inte kan schemalägga detta, dvs. generera om Bloomfiltret varje dag/vecka/månad. Förmodligen bäst att utgå från en lämpligt stor hashtabell, och generera "tillfälliga" Bloomfilter för snabb sökning.)

- **Om bloomfilter**, använda sig av flera hashfunktionern som ger index i samma tabell. Ett ord finns bara i tabellen om alla 14 hasfunktioner returnerar sant. Om hashtabellen som är n lång är fylld med m ettor (m<n) blir sannolikheten för att ett felstavat ord (1-1/m)^14. För 1/m=0.5 är sannoliketen 0.5^14=0.006%

4. Webbtoppen (Tildatenta 980321)

Vissa webbsidor räknar hur många besökare dom har eftersom välbesökta webbsidor ger prestige. Du får i uppdrag att skapa webbtoppen, ett program som för varje dag läser av räknarna för tiotusen webbsidor och sedan publicerar dagens tio i topp.

Din första tanke är att spara talen i en lista med längd tiotusen, leta fram och skriva ut segraren, nollställa segraren och göra detta tio gånger. ***Hur många jämförelser skulle krävas för denna algoritm?***

Din andra tanke är att spara talen i en trappa (heap) och sedan ta ut och skriva ut tio tal ur trappan. *Hur många jämförelser kan det då bli frågan om?*

Din tredje tanke är att det borde räcka med en trappa med plats för tio tal.

Hur skulle man då göra och hur många jämförelser skulle krävas?

Tio genomletningar av listan tar

$10 * 10000 = 100000$ jämförelser

Inmatning i trappan tar cirka $n * log_2(n)$, alltså 130 000 jämförelser och utplockning cirka $10 * log_2(n)$, alltså ytterligare 130 jämförelser.

Det smarta är att ha en tioplatsers min-heap! (Dvs, alla barn-noder är större än rootnoden.)

Överst ligger då det tionde i ordningen av alla tal man sett och det är ju det man ska jämföra varje tal med för att se om det ska in bland dom tio bästa. Vi fyller på trappan med de först tio talen. Sen sätter vi bara in ett tal i trappan om root talet är mindre än de nästa värde listan. Med normal tur blir det inte så ofta man ska byta ut tionde talet, så antalet jämförelser blir bara drygt tiotusen. Om talen är slumpmässigt utspridd kommer vi behöva sätta in mer i början och färre i slutet. Vi kommer behöva göra maximal 10 log 10 (skapa heapen från tio första talen) + n*log 10 (om vi behöver byta ut alla talen) vilket ger komplexiteten O(n).

5. Lönar sig sortering? (Tildatenta 970404)

En miljon dumbolotter säljs var månad. För varje lott sparas lottnumret och köparen i ett objekt. En lista med en miljon objekt finns alltså i datorn vid dragningen, då tusen vinstnummer slumpas fram, ett efter ett. För varje nummer måste hela listan letas igenom, eftersom den är osorterad. ***Hur många jämförelser får man räkna med totalt? Lönar det sej att först sortera listan, en gång för alla?***

Ja. I en osorterad lista krävs cirka en 500 000 jämförelser för varje sökning, dvs totalt en 1000*500 000=500 000 000= 500 miljoner

$(0 : 5 * 1000000$ sökningar$*1000$ vinstnummer).

Sortering med quicksort kräver cirka n log n jämförelser, dvs cirka 20 miljoner $(1000000 * log_2(1000000)) = 1000000 * 20$ Sedan tar varje binärsökning (O(log n)) bara tjugo jämförelser, dvs tjugotusen totalt (20 jämförelse/vinstnummer * 1 000 vinstnummer). $(120000000 + 20 * 1000 = 20020000$

Ja det lönar sig!

6. Billig standard selection sort (Tildatenta 960303)

Tilda och Totte skrev var sin sorteringsprocedur. Tilda valde en utsökt merge sort medan Totte tog en standard selection sort. När dom provkörde med tusen objekt gick ändå Tottes program lika fort, eftersom han har superdator. Men med tiotusen objekt vann Tilda. ***Med hur mycket?***

Vi vet at för n=1000 är $t_{1000} = t_{selectionsort} = t_{mergesort}$. I och med att vi vet komplexiteten för både Tottes och Tildas algorithm kan vi räkna ut låmng tid det tar att sortera 10000 objekt för de båda.
Selection sort (urvalssortering) är $O(n^2)$, dvs för någon konstant k är tiden t:

$t = k * n^2$

$$\left.\begin{array}{l} n = 10000 \Rightarrow t_1 = k_{urv} * 10000^2 \\ n = 1000 \Rightarrow; t_2 = k_{urv} * 1000^2 \end{array}\right\} \frac{t_2}{t_1} = 100$$

Tottes 10 000-objektssortering tar alltså 100 gånger så lång tid som 1 000- objektssorteringen.

Merge sort är O(n log n), dvs $t = k * n * log_2(n)$.

$$\left.\begin{array}{l} n = 10000 \Rightarrow t_2 = k_{merge} * 10000 * log_2(10000) \\ n = 1000 \Rightarrow; t_2 = k_{merge} * 1000 * log_2(1000) \end{array}\right\} \frac{t_2}{t_1} = \frac{10 * 13}{1 * 10} = 13\ (13,3...)$$

$2^{13} = 8192 \Rightarrow log_2(10000) \approx 13$ (exakt $13,287...$) $2^{10} = 1024 \Rightarrow log_2(1000) \approx 10$ (exakt $9,965...$)

Tildas 10 000-objekts sortering tar alltså 13 gånger så lång tid som 1 000-objekts sorteringen.

Tildas 10 000-objekts sortering är därför $100/13 = 8$ gånger så snabb som Tottes trots den långsammare datorn

7. Hoppfull sortering

Höjdhoppsfederationens databas över världens alla höjdhoppstävlingsresultat består av objekt med bland annat fälten **datum**, **plats**, **höjd** (cm), **hoppare** och **rivit/klarat**. På skivminnet ligger objekten i datumordning, men man vill sortera om dom i resultatordning, nämligen först efter kategorin rivit/klarat med klarade före rivna och sen inom varje katergori i rivit/klarat ska höga hopp komma före låga. ***Vilken sorteringsmetod är bäst? Motivera utförligt.***

De hopp som finns är grovt sett 100 - 300 cm, dvs det finns bara några hundra olika höjdvärden (distributioner) i hoppfilen. Antalet registrerade hopp är väldigt många fler än antalet höjdvärden (distributioner) och då är distributionsräkning bästa sorteringsalgoritmen. Tar vi hänsyn till rivit/klarat får vi dubbelt så många distributioner.

Algoritm: Läs igenom filen två gånger, första gången för att räkna hur många hopp det finns av varje rivit/klarat plus höjd. Sedan avsätter man lagom stort segment av listan för varje rivit/klarat plus höjd och vid andra genomläsningen av filen kan varje hopp sättas in på rätt ställe i listan.

8. Tjugondag Knut kastas julen ut (Tildatenta 010116)

För att kontrollera sanningen i detta talesätt har man i en fil samlat tre miljoner datum för svenska julgranars utkastning. Man vill veta mediandatum, alltså det datum då hälften av granarna slängts ut, ut, ut och hälften ännu står gröna och granna i stugan. Rangordna följande sex föreslagna metoder efter deras effektivitet. Binärsökning, hashning, insättningssortering, distributionsräkning, djupet-först- sökning, trappsortering (heap sort). Vi vill sortera datumen och plocka ut det mittersta.

Distributionsräkning är bäst (O(n)) eftersom det bara finns 365 olika datum.
Trappsortering är näst bäst (O(n log n) och man kan avbryta när hälften sorterats.
Insättningssortering fungerar också $(O(n^2))$.
Hashning är nästan oanvändbart; man bör i så fall vara säker på att hashfunktionen inte kan ge krockar. (går att ha en 6 miljoner lång hashtabell, som placerar datumen i ordning, gå sen in i tabellen och ta ut det mittersta värdet)
Binärsökning och djupet-först-sökning (vi vet inte när vi kommer till medianen) går inte att använda för att hitta medianen.(inga sorteringsalgoritmer, finns ingen referenspunkt som säger när vi kommit till medianen för dessa två)

9. Skatteregistret

Riksskatteverkets databas med nio miljoner svenskar finns sorterad på efternamn. Man vill sortera om den på personnummer. ***Hur många jämförelser krävs med quicksort? Hur många med den bästa metoden?*** Eftersom $n \approx 2^{23}$ (= 8 388 608) tar quicksort 9 000 000 * 23 = 207 miljoner jämförelser.

Radixsortering (gå igenom alla, dela upp i tio buntar efter sista siffran, lägg samman, gör om med näst sista siffran etc) tar 10 * 9 000 000 = 90 miljoner. Man kan faktiskt strunta i sista siffran eftersom den är checksiffra. Det finns inte två pnr som bara skiljer sej i den siffran.

10. Båtflytt (Tildatenta 020406)

Under en seglingstävling vill varje båt hitta den snabbaste vägen till målet. Problemet är att en segelbåt inte kan segla hur som helst och att den seglar olika snabbt beroende på vindriktning och styrka. Antag att havet förenklat består av en massa jämnt fördelade punkter med information om vindstyrka, vindriktning och vilka punkter som finns runt om.

Beskriv en algoritm som på ett så effektivt sätt som möjligt tar reda på vilka punkter som ligger utefter den snabbaste seglingsvägen givet en startpunkt och en slutpunkt.

Båtägaren är orolig att hans miljövänliga bottenfärg ska nötas bort och vill därför istället ta den väg som är kortast (dvs minst antal steg). Förklara vad som behöver ändras i din föregående algoritm.

Använd bästaförstsökning med en prioritetskö som prioriterar på lägsta seglade totaltiden. Låt varje nod innehålla total seglingstid samt ha en faderspekare (för rekursiv utskrift av vägen då lösning hittats). Princip för genomgång av problemträdet:

- Lägg startpunktsnoden med totaltiden noll och tom faderspekare i prioritetskön.
- Upprepa så länge kön inte är tom:
    1. Plocka ut en fadersnod ur kön. Om detta är slutpunkten, skriv ut vägen rekursivt och avsluta.
    2. Generera en son i taget genom att för varje punkt runt omkring fadersnoden skapa en son-nod med seglingstiden ökad beroende på vindstyrka, vindriktning och placering i förhållande till fadersnoden. Lägg in son-noden i prioritetskön.

Om dumbarnskoll ska utnyttjas måste det ta hänsyn till både punkten och totala seglingstiden till den punkten. Alla söner med sämre tider till samma punkt är då dumsöner. På det viset slipper algoritmen besöka samma punkt flera gånger - den blir effektivare.

Eventuellt krävs någon snabb uppslagning av punkternas information, t ex med en hashtabell som hashar på punkternas nummer eller position. Noderna kan innehålla lägsta tid som uppnåtts för att tillåta dumbarnkoll utan att kräva extra minne.

För kortaste vägen, använd istället bredden-först med en vanlig kö och blanda inte in totala seglingstiden i varje nod. I detta fall måste vi göra dumbarns-koll för att sökningen ska bli effektiv.

Datastrukturer: Prioritetskö / kö Hashtabell Noder med seglingsdata

Reasoning on probability of collision

Here is an example to give an inutition on why is good to choose a prime as the lenght of the hash table in order to decrease the probability of collisions. Let a, b and n be integers, then

$a \equiv bmod(n)$

Means that a-b is a integer multiple of n. Let

$a < n$ and $b = a + n * q$ with q beeing an integer

then b mod n $\Rightarrow a$

The probability of a collision in a table of size n is 1/n given that the probability of hashed numbers are the same. If instead the probability of having numbers that are a multiple of a factor of n.
For example: Our hash only returns 0,3,3*2,3*3,..,3*k. Thus The factor three is in all hased numbers. Now lets choose a hashvektor of size 6. Lets now hash a few number.
Hashing 0 -> pos 0
Hashing 3*1 -> pos 3
Hashing 3*2 -> pos 0
Hashing 3*3 -> pos 3
...
Hashing 3*k -> to pos 0 if k is even else to 3.

The probability of hashing a number to positions 1,2,4,5 is zero whereas the probability it is 0.5 to hash to 0 or 3. This is not good since the this will give rise to many more collisions than if the probability where equal for all posisions. /textitCan we do somthing to prevent this from happening?

Yes we can!, Choose n to be and prime, say 7 which is not a multiple of the hashed values. Whe would then have in our example' that
Hashing 0 -> pos 0
Hashing 3*1 -> pos 3
Hashing 3*2 -> pos 6
Hashing 3*3 -> pos 2
Hashing 3*4 -> pos 5
Hashing 3*3 -> pos 1
...
Hashing 3*k -> to either position with equal probability

Now the probability is 1/7 for each of the posisions.

Sortering

- Urvalssortering (Selection sort), komplexitet O(n^2).
- Bubbelsortering (Bubble sort), komplexitet O(n^2) men bara i värsta fall. För en nästan sorterad lista går det mkt fortare.
- Insättningssortering (Insertion sort), kompelxitet O(n^2). Bäst för att sortera in ett värde i en redan sorterad lista.
- Quicksort, O(n*log n). Snabbast av alla sorteringsalgorithmer. Intution: är att sortera listan med ungefär hälften med alla mindre talen till vänster och de större till höger, jämfört med ett pivo-tal som man sätter. Sen gör man om samma sak fas nu på den vänsta och högra delen var för sig. Det blir en rekursive tanke.
- Merge sort, O(n*log n) men kräver mer minnesutrymme än quicksort. Intution: dela upp listan i två delar, fortsätt med del listorna tills de är av storleken 2, sortera. Slå sedan samman två sorterade del listorna. Fortsätt tills alla listor är sammanslagna.
- Räknesortering (Distribution sort), O(n). Kräver dock att värdena som ska sorteras in är av ett litet antal.