LINKÖPINGS UNIVERSITET

TDDI16: Datastrukturer och algoritmer

Lab 2: Knäcka lösenord

INSTITUTIONEN FÖR DATAVETENSKAP

Höstterminen 2018

2018-06-27

1 Upplägg

Första delen av instruktionen, avsnitt 2 till 7, innehåller en fullständig beskrivning av problemet utan några konkreta lösningsidéer. Känner ni att ni har en bra idé redan efter att ha läst första delen kan ni börja programmera redan då. Resten av instruktionen är tips och lösningsidéer som ni kan läsa vid behov.

Fyll i filen `readme.txt` och läs och fundera på frågorna i avsnitt 8 och 10 innan ni redovisar laborationen för assistent.

Resten av instruktionen innehåller en mer ingående beskrivning av de givna filerna (avsnitt 9), ytterligare exempel av *subset sum* (avsnitt 11), lite tips för testning (avsnitt 12 och 13), en introduktion till *meet in the middle* (avsnitt 14) och slutligen en lösningsidé (avsnitt 15 och 16) följt av tips vid användning av `std::unordered_map` (avsnitt 17).

2 Hantering av lösenord

När du loggar in på ett system (exempelvis IDA:s Linuxdatorer) måste systemet kontrollera att det inmatade lösenordet stämmer överens med vad systemet förväntade sig, för att på så sätt kontrollera om du är den du utger dig att vara. En enkel implementation av detta skulle vara att lagra alla användarnamn och lösenord i klartext någon stans i systemet (exempelvis i filen `/etc/passwd`, eller `/etc/shadow`). Detta är dock en mycket dålig idé eftersom lösenorden för alla användare blir tillgängliga ifall någon obehörig skulle få tillgång till systemet (som vi alla vet är det lätt att använda samma lösenord på flera ställen, och därmed är det viktigt att vara extra försiktig med hanteringen av lösenord).

Det vi kan göra för att lösa problemet är att använda en *hashfunktion* (gärna en så kallad *kryptografisk hashfunktion*). En hashfunktion är en funktion, $h(x)$, som ”krypterar” x på ett sådant sätt att vi utifrån $h(x)$ inte enkelt kan beräkna x igen. Vi kan alltså inte hitta en invers, $h^{-1}(y)$, som går att beräkna inom rimlig tid. I och med att vi inte enkelt kan ”avhasha” lösenorden som vi en gång har hashat, så blir användarnas lösenord säkrare: även om filen med alla hashade lösenord skulle hamna i fel händer så skulle inte denna person enkelt kunna läsa lösenorden i filen.

Trots att vi har hashat alla användarnas lösenord kan vi ändå kontrollera om lösenordet var korrekt. När vi vill kontrollera ett lösenord kan vi helt enkelt hasha det lösenord som vi vill kontrollera och jämföra den hashen mot hashen som är lagrad i filen. Är de lika så vet vi (i alla fall med hög sannolikhet, det kan finnas kollisioner) att lösenordet stämmer överens med det riktiga lösenordet, och då kan vi släppa in användaren i systemet.

3 Subset sum för hashning av lösenord

I den här labben kommer vi använda hashfunktionen *subset sum* för att hasha lösenorden. För enkelhets skull antar vi att lösenorden består av tecknen 0 och 1, och alltid är en viss längd, säg N. Systemet har också en tabell, T, som används som en nyckel för hashfunktionen. Tabellen består av N heltal numrerade från 0 till $N - 1$, vart och ett N bitar långa. För att hasha ett lösenord väljer systemet en delmängd av raderna i T motsvarande de positioner i lösenordet som innehåller tecknet 1 och summerar dem. Summan är hashvärdet för lösenordet.

Följande exempel illustrerar processen. Antag att vi ska hasha lösenordet `00101` med hjälp av följande tabell:

ID	Innehåll
0	10110
1	01101
2	00101
3	10001
4	01011

Lösenordet innehåller två 1:or, på plats nummer 2 och 4 (kom ihåg, vi börjar räkna från 0), alltså väljer rad 2 och 4 från tabellen (vilket kan ses i kolumnen *lösenord* här nedanför). Sedan summerar vi de valda raderna för att få hashvärdet. I detta fall blir det $00101 + 01011 = 10000$. I praktiken skulle såklart en mycket större tabell och längre lösenord användas.

ID	Innehåll	Lösenord	Valda rader
0	10110	0	-
1	01101	0	-
2	00101	1	00101
3	10001	0	-
4	01011	1	01011
		Summa:	10000

4 Uppgiften

Du har fått tag på tabellen T från ett system tillsammans med en uppsättning användarnamn och hashade lösenord. Eftersom de hashade lösenorden inte är särskilt användbara vill du såklart hitta ett sätt att beräkna lösenorden från hasharna igen, helst inom rimlig tid.

Systemet använder subset sum för att hasha lösenorden. För att göra det enkelt för användarna tillåts tecknen `a-z` och `0-5` i stället för bara 0 och 1 som i exemplet ovan. I och med att det finns precis 32 tillåtna tecken kan varje tecken översättas till fem bitar. `a` kodas som 00000, `b` kodas som 00001, `c` som 00010 och så vidare. Ett lösenord som består av C tecken består alltså av $N = 5C$ bitar, och tabellen som används för ett sådant lösenord innehåller således också $N = 5C$ rader. Ett lösenord med 5 tecken behöver alltså en tabell som innehåller 25 rader.

Ett första försök att lösa problemet är att helt enkelt testa att hasha alla möjliga lösenord, ett efter ett, och se vilka som ger samma hashvärde som det vi hittade i systemet. Koden för detta finns i filen `brute.cpp`, men problemet är att programmet tar oerhört lång tid på sig att hantera lite längre lösenord. Din uppgift är att skapa ett nytt program, `decrypt`, som förbättrar `brute` avsevärt. Programmet ska klara av ett lösenord på **8 tecken** inom några sekunder, och lösenord på **10 tecken** inom ett par minuter.

Notera: Beroende på innehållet i T, så kan det finnas flera olika lösenord som producerar samma hashvärde. Ditt program ska i sådana fall hitta *alla* sådana lösenord.

Notera: Ditt program kommer behöva den abstrakta datatypen *symboltabell*. Du ska använda en hashtabell (`std::unordered_map`) för att realisera denna. Det innebär att du måste implementera en egen hashfunktion för `Key` för att kunna använda den som nyckel. Mer information om detta finns på `https://en.cppreference.com/w/cpp/utility/hash` och i avsnitt 17. Värt att tänka på är att detta är en hashfunktion som används för att hashtabellen ska kunna göra uppslagningar snabbt, den är helt separat från hashningen av lösenord som görs av subset sum.

Tips: Om du vill fokusera på att knäcka lösenorden först kan det vara en bra idé att använda `std::map` som symboltabell till en början. Den behöver bara `operator <`, vilken redan är implementerad för `Key`.

5 Indata

Längden på lösenord ändras genom att specifiera ett värde på konstanten C i filen `key.h`. Programmet ska fungera för alla C i intervallet $1 \leq C \leq 10$. Programmet ska läsa det hashade lösenordet från kommandoraden (`argv[1]`), och tabellen T ska läsas in från standard input. Tabellen består av ett ord på varje rad, där den första raden motsvarar bit 0 i lösenordet, nästa rad bit 1 och så vidare. Inläsning av data hanteras av det givna kodskelettet.

Programmet anropas lämpligtvis enligt följande för att inte behöva skriva in tabellen manuellt varje gång programmet körs (`njjbh` är lösenordet `pass` hashat med tabellen `easy5.txt`):

```
./decrypt njjbh < easy5.txt
```

6 Utdata

Programmet ska skriva ut alla lösenord som producerar det givna hashvärdet med den givna tabellen, ett möjligt lösenord på varje rad. Lösenorden ska skrivas ut på det formatet som utmatningsoperatorn för datatypen `Key` producerar. Utdatan ska avslutas med en text som indikerar den totala tidsåtgången för programmet (detta görs av den givna koden).

Exempelvis ska körningen `decrypt njjbh < easy5.txt` producera följande utdata (ordningen på raderna spelar ingen roll):

```
passw  15  0 18 18 22   01111000001001010010110
p3csw  15 29  2 18 22   01111111010001010010110
p35cw  15 29 31  2 22   01111111011111100010110
p355g  15 29 31 31  6   01111111011111111100110
2dcsw  28  3  2 18 22   11100000110001010010110
2d5cw  28  3 31  2 22   11100000111111100010110
2d55g  28  3 31 31  6   11100000111111111100110
2qssw  28 16 18 18 22   11100100001001010010110
Decryption took 0 seconds.
```

7 Testfall

Följande lösenord har krypterats med `rand8.txt`, `rand10.txt` och `rand12.txt` respektive. När du har löst problemet ska ditt program åtminstone klara av lösenorden med 10 tecken:

```
xwtyjjin  h554tkdzti  x5ousvhg5eym
rpb4dnye  oykcetketn  2xaells41oor
kdidqv4i  bkzlquxfnt  vuhobda1rp53
m5wrkdge  wixxliygk1  oynkianhmv0h
```

8 Att tänka på innan ni redovisar

- Hur delar ni upp tabellen? Vilka olika uppdelningar har ni testat? Vilken eller vilka fungerar bäst?
- Hur har ni konstruerat hashfunktionen som ni använder till er hashmap? Kan den förbättras på något sätt?

- Vad har er lösning för tidskomplexitet? Är den bättre än `brute`?

9 Givna filer

Följande filer är givna för labben:

`key.{h,cpp}` Innehåller en implementation av datatypen `Key`, som används för att lagra lösenord och hashade lösenord i en representation som kan manipuleras som vanliga heltal. `Key` representerar lösenordet som ett tal av längd C i bas 32. Alla beräkningar sker modulo 2^N, så om talet `5555` ökas med ett slår talet runt och resultatet blir `aaaa`.

`encrypt.cpp` Innehåller ett program som kan användas för att hasha lösenord. Programmet används på samma sätt som de andra programmen i labben. Det vill säga, för att hasha ett lösenord med tabellen `easy5.txt` körs `encrypt` enligt följande:

```
./encrypt passw < easy5.txt
```

Programmet producerar då följande utdata:

```
   passw  15  0 18 18 22   01111000001001010010110
 1 aaaac   0  0  0  0  2   00000000000000000000010
 2 aaaae   0  0  0  0  4   00000000000000000000100
 3 aaaai   0  0  0  0  8   00000000000000000001000
 4 aaaaz   0  0  0  0 25   00000000000000000011001
10 aabaa   0  0  1  0  0   00000000000010000000000
13 aaiaa   0  0  8  0  0   00000000001000000000000
15 abaaa   0  1  0  0  0   00000000100000000000000
18 aiaaa   0  8  0  0  0   00000100000000000000000
20 baaaa   1  0  0  0  0   00001000000000000000000
22 eaaaa   4  0  0  0  0   00100000000000000000000
23 iaaaa   8  0  0  0  0   01000000000000000000000
   njjbh  13  9  9  1  7   01101010010010000100111
```

Första raden är det lösenord som angavs på kommandoraden, utskrivet i tre olika representationer: först med text som vanligt, sedan skrivs ordningsnumret för varje bokstav ut, och till sist skrivs den binära representationen ut.

Nästkommande rader har en siffra framför sig. Var och en av dessa motsvarar en rad i tabellen som har valts ut av lösenordet. I det här fallet har raderna 1, 2, 3, 4, och så vidare valts ut eftersom bit 1, 2, 3, 4 och så vidare var 1 i lösenordet. Den sista raden är summan av alla rader som valdes ut, och därmed det slutgiltiga hashvärdet.

`brute.cpp` Innehåller ett program som implementerar en möjlig lösning på problemet genom att testa alla möjliga lösenord och därigenom se vilka som är korrekta. Problemet med detta program är att det tar otroligt lång tid på sig att knäcka ett lösenord. I övrigt fungerar programmet som programmet `decrypt` ska fungera.

`decrypt.cpp` Innehåller ett kodskelett som sköter in- och utmatning till den lösning som ni ska producera. I nuläget gör programmet ingenting utöver att läsa in all indata och skriva ut den totala tidsåtgången.

`Makefile` Innehåller regler för att kompilera den givna koden. Kör `make` för att kompilera alla givna filer, eller `make encrypt`, `make decrypt` eller `make brute` för att kompilera enskilda program.

`easy*.txt` Innehåller tabeller med relativt tydliga mönster som kan vara användbara under testning, eftersom det är relativt enkelt att lista ut vilka rader som valts manuellt. Siffran i namnet anger vilken

lösenordslängd tabellen är avsedd för, räknat i tecken. Filen `easy5.txt` innehåller alltså en tabell för 5 tecken långa lösenord, det vill säga 25 rader.

`rand*.txt` Innehåller tabeller med slumpmässiga rader. Den här typen av data vill vi använda när vi hanterar ”riktiga” lösenord. I och med att raderna innehåller slumpmässiga tal kommer det inte finnas särskilt många mönster som kan utnyttjas för att förenkla dekrypteringen.

10 Att tänka på i praktiken

I den här labben har vi sett att den hashfunktion vi valde för att lagra lösenorden inte var särskilt säker.

- Vilka egenskaper har hashfunktionen som ni kan utnyttja för att knäcka den?
- Baserat på detta, om vi ska välja en hashfunktion som är säker för att lagra lösenord. Vilka egenskaper ska vi då leta efter?
- Vad har `brute` för tidskomplexitet? Vad har `decrypt` för tidskomplexitet? Vad gör `decrypt` för att bli så mycket snabbare?
- Vad kan vi göra för att även `decrypt` ska ta tillräckligt lång tid på sig för att det ska bli ohållbart att dekryptera lösenorden på detta sätt?

11 Ytterligare exempel av hasningen

I det här exemplet hashar vi lösenordet `password` med tabellen `rand8.txt`. Detta kan göras med hjälp av det givna programmet `encrypt` genom att köra:

```
./encrypt password < rand8.txt
```

Programmet kommer då att skriva ut följande text:

```
   password  15  0 18 18 22 14 17  3   01111000001001010010101100111010001000 11
 1 gobxmqkt   6 14  1 23 12 16 10 19   00110011100000110111011001000001010100 11
 2 qdrvjxwz  16  3 17 21  9 23 22 25   10000000111000110101010011011110110110 01
 3 joobqxtz   9 14 14  1 16 23 19 25   01001011100111000001100001011110011110 01
 4 xnoixmnk  23 13 14  8 23 12 13 10   10111011010111001000101110110001101010 10
10 tcixtvem  19  2  8 23 19 21  4 12   10011000100100010111100111010100100011 00
13 lqtsdtca  11 16 19 18  3 19  2  0   01011100001001110010000111001100010000 00
15 zlptzlfp  25 11 15 19 25 11  5 15   11001010110111110011110010101100101011 11
18 gmjuvyqw   6 12  9 20 21 24 16 22   00110011000100110100101011100010000101 10
20 uoqrdhwp  20 14 16 17  3  7 22 15   10100011101000010001000110011111011001 11
22 ltdkzndz  11 19  3 10 25 13  3 25   01011100110001101010110010110100011110 01
23 btezrznq   1 19  4 25 17 25 13 16   00001100110010011001100011100101101100 00
26 bujilqno   1 20  9  8 11 16 13 14   00001101000100101000010111000001101011 10
27 qgaicljl  16  6  0  8  2 11  9 11   10000001100000001000000100101101001010 11
28 yyefwcld  24 24  4  5 22  2 11  3   11000110000010000101101100001001011000 11
30 gnvowyjk   6 13 21 14 22 24  9 10   00110011011010101110101101100001001010 10
34 aynzobxh   0 24 13 25 14  1 23  7   00000110000110111001011100000110111001 11
38 lxwewfhh  11 23 22  4 22  5  7  7   01011101111011000100101100010100111001 11
39 aenipbjd   0  4 13  8 15  1  9  3   00000001000110101000011110000010100100 11
   vbskbezp  21  1 18 10  1  4 25 15   10101000011001001010000010010011001011 11
```

På första raden skriver programmet ut originallösenordet i tre olika representationer. Först i ren text, sedan som 8 heltal mellan 0 och 31 som motsvarar tecknens ordningsnummer. Till sist skrivs den binära representationen ut, vilket är väldigt användbart för hashningen, eftersom den binära representationen anger vilka rader i tabellen som ska summeras.

Programmet anropar sedan funktionen `subset_sum`, som itererar igenom alla bitar i talet och väljer de rader som innehåller en 1:a. I det här fallet kan vi se att `password` innehåller 18 1:or, och därför väljs 18 rader från tabellen i `rand8.txt`. Programmet skriver ut de valda raderna tillsammans med radens nummer, vilket gör att vi kan se att raderna 1, 2, 3 och 4 är valda eftersom att bit 1, 2, 3 och 4 är satta till 1 i talet. Den sista raden i utdatan är summan av alla raderna, vilket är det hashade lösenordet.

12 Tips för testning

Börja testa med små nyckelstorlekar, exempelvis 6 tecken för att se att din lösning fungerar. Då är problemen tillräckligt små så att du kan jämföra utdatan för `decrypt` med utdatan för `brute` för att se att du har fått med alla möjliga lösenord. Testa också att generera egna lösenord! Lösenord med många `a`:n kan vara problematiska ibland.

13 Datatyper

Trots att vi lagrar all vår data som datatypen `Key` har vi faktiskt två olika typer av data i våra program. Först och främst har vi datatypen *lösenord*, vilket motsvarar ett lösenord i klartext, exempelvis `password`. Utöver det har vi också datatypen *hashat lösenord*, vilket är alla resterande tal. Även om dessa datatyper båda representeras av typen `Key` i C++ så är det viktigt att hålla dem åtskiljda när vi tänker på problemet eftersom det är olika operationer som är meningsfulla att göra på dem.

Datatypen *lösenord* vill vi se som en serie bitar snarare än ett heltal. När vi hashar ett lösenord är vi bara intresserade av vilka bitar som är satta så att vi kan välja lämpliga rader i tabellen. Det är sällan intressant att betrakta lösenord som tal och exempelvis subtrahera ett lösenord från ett annat. Det är helt enkelt inte meningsfullt eftersom vi betraktar lösenorden som en sekvens av bitar. Med det sagt, så kan man ibland behöva använda aritmetik för att manipulera bitarna i talet på ett bra sätt, vilket beskrivs i avsnitt 16. Kort sagt skulle man kunna säga att ett *lösenord* beskriver vilken delmängd av raderna i tabellen som ska väljas (därav namnet *subset sum*).

Datatypen *hashat lösenord*, å andra sidan, vill vi betrakta som ett tal eftersom att hashade lösenord är en *summa* av tal från tabellen. På grund av det är det intressant att prata om summan eller differensen mellan två hashade lösenord. Däremot är det inte intressant att se om en viss bit är satt i ett hashat lösenord.

14 Meet in the middle

Meet in the middle är en teknik som är användbar för att snabba upp problem som detta, som går ut på att vi måste söka oss igenom en relativt stor mängd möjliga lösningar för att inse vilka som är rätt. För att illustrera idén kan vi beakta följande problem som förhoppningsvis är något enklare:

Givet ett tal c, hitta två tal a och b, större än 0, så att $a^2 + b^2 = c^2$.

En möjlig lösning till problemet är att helt enkelt testa alla kombinationer av a och b så att $0 < a, b < c$ för att hitta alla lösningar. Detta tar dock $\mathcal{O}(c^2)$ tid, vilket snabbt blir ohållbart för stora värden för c.

En alternativ lösning på detta problemet är att använda sig av *meet in the middle* (även om det finns andra smidigare lösningar för just detta problemet). Idén med meet in the middle är att dela upp alla möjliga indata i två delar så att vi kan beräkna ena halvan i förväg och därmed undvika mängder av onödiga beräkningar när vi kommer till andra halvan av indatan. I vårt exempel kan vi undersöka uttrycket a^2 och b^2 separat genom att skriva om problemet som $a^2 = c^2 - b^2$. Fördelen med omskrivningen är att vi med hjälp av en tabell för alla möjliga värden av a^2 enkelt kan testa om det finns en lösning för ett specifikt värde på b^2 (kom ihåg att c och därmed c^2 är givet).

Vi börjar alltså med att bygga en tabell som innehåller värdet på a^2 för alla möjliga a. Vi lagrar lämpligtvis dessa i en symboltabell (exempelvis en hashtabell) med nyckel a^2 och värde a så att vi snabbt kan hitta dem senare. Sedan kan vi undersöka andra halvan. För alla möjligheter för b kan vi nu beräkna uttrycket $c^2 - b^2$ och se om det värdet finns i symboltabellen. Om det fanns i symboltabellen betyder det att det fanns ett a som tillsammans med det b vi undersöker för tillfället uppfyller det villkor vi var intresserade av, nämligen $a^2 + b^2 = c^2$.

Nyckel a^2	Värde a
1	1
4	2
9	3
16	4

Tabell 1: Symboltabell

Antag exempelvis att $c = 5$. Då skapar vi först en tabell för värdena av a^2 för alla $0 < a < 5$, vilken kommer innehålla datan i tabell 1. Sedan kan vi testa alla möjliga värden på b. $b = 1$ ger $c^2 - b^2 = 24$, och eftersom 24 inte finns i tabellen är inte $b = 1$ en lösning till problemet. $b = 2$ ger $c^2 - b^2 = 21$, vilket inte heller finns i tabellen. $b = 3$ ger $c^2 - b^2 = 16$, vilket finns i tabellen. Tabellen säger också att $a = 4$ ger $a^2 = 16$, alltså är $a = 4, b = 3$ en lösning till problemet. Om vi sedan fortsätter att försöka med $b = 4$ kommer vi (förvånansvärt nog) se att även $a = 3, b = 4$ är en lösning på problemet. I och med detta har vi lyckats hitta alla lösningar genom att bara testa $4 * 2 = 8$ värden i stället för att testa alla kombinationer av a och b, vilket skulle kräva $4^2 = 16$ test.

Om vi antar att symboltabellen implementeras med hjälp av en hashtabell (eller något ekvivalent) så kommer denna lösningen ta $\mathcal{O}(c + c) = \mathcal{O}(c)$ tid, vilket är snabbare än den intuitiva lösningen. I just det här fallet kan vi lösa problemet på andra sätt också, exempelvis i och med att vi vet att a^2 är monoton, och att det finns en invers till a^2 ($\sqrt{a}$) som vi kan utnyttja. *Meet in the middle* utnyttjar dock inte något av dessa fall, och därmed kan vi använda *meet in the middle* i fler situationer, exempelvis i *subset sum*, där vi varken har en monoton funktion eller en invers att tillgå.

15 Lösningsidé

Vi kan som sagt använda *meet in the middle* för att räkna fram lösenord som har hashats med hjälp av subset sum relativt snabbt (i alla fall mycket snabbare än `brute`). Den stora frågan är bara: Hur kan vi dela upp problemet?

För att belysa detta kan vi tänka oss att vi skriver hashfunktionen på följande sätt:

$$P_0T_0 + P_1T_1 + P_2T_2 + P_3T_3... + P_NT_N = H$$

P_i är bit nummer i i lösenordet som hashas, och T_i är rad nummer i i tabellen T. Detta motsvarar beskrivningen i avsnitt 3 eftersom raderna i tabellen multipliceras med antingen 1 om raden ska användas eller 0 om raden inte ska användas. När vi har skrivit hashfunktionen på detta sätt kan vi se att den har liknande struktur som det problemet vi löste i avsnitt 14, även om vi har fler termer här. På samma sätt som tidigare kan vi skriva om problemet på följande sätt för att kunna använda *meet in the middle*:

$$P_0T_0 + P_1T_1 + ... + P_mT_m = H - P_{m+1}T_{m+1} - ... - P_NT_N$$

Detta innebär i praktiken att vi tänker oss att vi delar upp tabellen i två delar, en del som motsvarar de m första bitarna i lösenordet, och den andra delen innehåller resterande bitar. Sedan kan vi testa alla möjliga kombinationer av bitarna i de två delarna separat genom att sätta in resultaten för ena halvan i en symboltabell. Om vi skriver om problemet ytterligare lite så kan vi se att vi inte ens behöver implementera vår egen version av `subset_sum`:

$$\texttt{subset\_sum}(P_a, T) = H - \texttt{subset\_sum}(P_b, T)$$

Där P_a är ett lösenord där alla bitar utom bitarna 0 till m alltid är noll, och P_b är ett lösenord där bitarna 0 till m alltid är noll. I och med att olika delar av P_a och P_b används, så kan vi enkelt slå ihop dem till lösenordet P genom att helt enkelt beräkna $P = P_a + P_b$.

16 Binär aritmetik

För att dela upp symboltabellen i två delar kan det vara smidigt att känna till hur man kan utnyttja aritmetiska operationer för att enkelt hitta alla kombinationer av ett känt antal bitar.

`brute` behöver iterera igenom alla möjliga kombinationer av rader i tabellen. Detta görs genom att i varje iteration lägga på 1 till en `Key` som har initierats till 0. När `brute` ser att `Key`-variabeln har slagit runt till 0 igen vet den att den har itererat igenom alla tal som `Key` kan representera, och därmed alla möjliga kombinationer av rader i tabellen.

Vi kan utnyttja samma idé för att iterera igenom delar av tabellen. För att iterera igenom de minst signifikanta bitarna (de längst till höger i utskriften) kan vi göra på samma sätt, så länge vi ser till att vi avslutar iterationen i tid. Detta kan vi exempelvis göra med hjälp av medlemsfunktionen `bit` för att se om en viss bit i talet är 1.

Vi kan också iterera genom alla kombinationer för de mest signifikanta bitarna (de längst till vänster i utskriften) genom att öka med ett tal som är större än 1. Om vi exempelvis ökar med 2 varje gång så kommer vi hoppa över alla tal där den lägsta biten är satt, och därmed bryr vi oss bara om kombinationerna av alla bitar förutom den lägsta. Detta fungerar för godtycklig tvåpotens. Talet 4 hoppar över de två minst signifikanta bitarna, talet 8 hoppar över de tre minst signifikanta bitarna, och så vidare.

17 Hashfunktion till `std::unordered_map`

`std::unordered_map` implementeras i allmänhet med en hashtabell, och behöver därför en hashfunktion för att fungera korrekt. Hur man implementerar en hashfunktion för en datatyp så att `std::unordered_map` använder den beskrivs på `https://en.cppreference.com/w/cpp/utility/hash`. Däremot är det inte alltid helt enkelt att inse vad hashfunktionen ska innehålla för att den ska fungera bra. Det kommer vi titta lite närmare på här!

För att kunna använda en datatyp som en nyckel i en hashtabell måste vi kunna omvandla den till ett heltalsvärde som hashtabellen kan använda för att snabbt hitta nyckeln. I literatur om datastrukturer och algoritmer brukar man använda en hashfunktion som omvandlar en nyckel av någon typ till ett heltal som motsvarar vilken plats nyckeln ska lagras på i den interna tabellen. Detta är dock ofta opraktiskt, eftersom det kräver att hashfunktionen känner till hur hashtabellen är implementerad (exempelvis: hur stor är tabellen nu? hur hanteras kollisioner?). I praktiken brukar man därför dela upp hashfunktionen i två delar: först en funktion som omvandlar nyckeln till ett heltal, sedan en funktion som beräknar vilken plats elementet ska vara på i tabellen givet det första heltalet. Den här uppdelningen gör att den första funktionen, som är

den man brukar kalla för hashfunktionen, inte behöver bry sig om implementationsdetaljer i hashtabellen och kan användas i många hashtabellsimplementationer. Den andra funktionen brukar ses som en del av hashtabellens implementation eftersom den är hårt kopplad till många andra designbeslut i tabellen. Den finns därmed inbyggd i `std::unordered_map`.

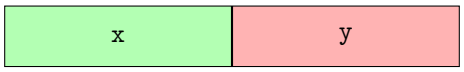
Den hashfunktionen vi behöver implementera behöver alltså bara bry sig om att på något sätt omvandla ett värde av nyckeldatatypen till ett heltal som får plats i en `size_t` (ofta ett 64-bitars heltal, kan lagra talen 0 till $2^{64} - 1$). För heltalsnycklar är detta enkelt, det går ofta bra att använda talet rakt av. När vi har lite mer komplicerade datastrukturer blir det dock inte lika uppenbart vad som ska göras. Vi kan börja med att titta närmare på vad som förväntas av en hashfunktion:

- Hashfunktionen ska skapa ett heltal från en nyckel i tabellen. I detta fall vill vi ha en hashfunktion som tar en `Key` som parameter och producerar en `size_t` (ett heltal) som resultat.
- Hashfunktionen ska ge samma resultat för samma värde varje gång. Det vill säga at om $a = b$ så $h(a) = h(b)$. Detta innebär också att om $h(a) \neq h(b)$ så är $a \neq b$.
- Hashfunktionen behöver inte ge unika resultat för olika värden. Det vill säga att om $a \neq b$ så är det inte garanterat att $h(a) \neq h(b)$.

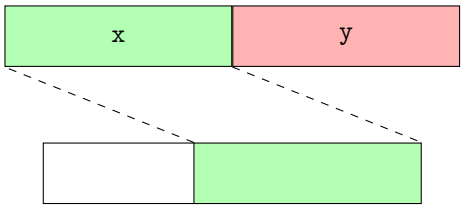
Utifrån dessa krav så kan vi konstruera en väldigt enkel hashfunktion som fungerar för alla datatyper: en funktion som alltid returnerar talet 0 (eller något annat tal). Den funktionen uppfyller alla krav ovan, och kan därför användas utan problem tillsammans med en `std::unordered_map`. Däremot är det ingen bra idé eftersom alla nycklar kommer att lagras på samma ställe i tabellen, och implementationen behöver därmed linjärsöka bland alla nycklarna varje gång. Inte alls bra!

För att hashtabellen ska kunna arbeta effektivt vill vi använda en hashfunktion som har få *kollisioner*. Det innebär att hashvärdet för två olika nycklar, $h(a)$ och $h(b)$, är olika så ofta som möjligt. Intuitivt kan man tänka att det gör det möjligt för hashtabellen att snabbt se att två nycklar är olika i många fall, vilket gör den snabbare. Har vi en hashfunktion som alltid returnerar 0 kommer hashtabellen tvingas anta att alla nycklar skulle kunna vara lika, och behöver därmed göra onödigt mycket arbete. För att ytterligare illustrera fördelen av en bra hashfunktion kan vi tänka oss en person som sorterar plastleksaker i olika högar. Om högarna är märkta med olika storlekar (vi hashar elementen baserat på deras storlek) så kommer många hamna i samma hög eftersom alla är ungefär lika stora, och då blir det väldigt jobbigt att hitta en specifik leksak senare. Märker vi i stället högarna med exempelvis färg (vi hashar elementen baserat på färg) så får vi sannolikt en jämnare fördelning av leksakerna i högarna, och det blir därmed enklare att hitta en specifik leksak senare (i alla fall om vi antar att de är färgglada!).

Intuitivt kan man tänka att hashfunktionen skapar en sammanfattning av de viktiga delarna av nyckeln. Vi vill att sammanfattningen ska innehålla så mycket information som möjligt från nyckeln så att det oftast är så att olika nycklar har olika sammanfattningar. Vi kan illustrera idén med följande exempel: Antag att vi har en datastruktur med två heltal som vi vill använda som en nyckel i en hashtabell:

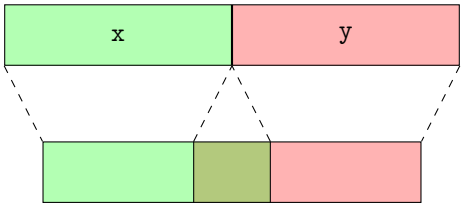


En enkel hashfunktion skulle vara att helt enkelt bara välja en hashfunktion som helt enkelt returnerar den ena datamedlemmen:



hashvärde

Detta skulle fungera bra så länge `x`-värdet varierar mycket bland de värden som lagras i hashtabellen. Skulle `x`-värdet vara detsamma för alla punkter blir det dock inte så bra. För att förbättra hashfunktionen för den situationen kan vi i stället kombinera båda datamedlemmarna på något sätt, exempelvis genom att multiplicera `x`-värdet med ett stort tal och addera `y`-värdet:



hashvärde

Detta kan göra att de båda datamedlemmarna ”överlappar” något i hashvärdet, men det gör ofta inte så mycket. Det viktiga här är att hashvärdet nu innehåller information från båda datamedlemmarna, vilket gör att skillnader i både `x`- och `y`-värdet syns i hashvärdet, vilket i sin tur hjälper hashtabellen att vara effektiv.