Föreläsning 16

Rekursiv sökning

TDDD86: DALP

Utskriftsversion av föreläsning i *Datastrukturer, algoritmer och programmeringsparadigm*
3 november 2015

Tommy Färnqvist, IDA, Linköpings universitet

Innehåll

Innehåll

1 Rekursiv sökning

Rekursiv problemlösning

`if` (problemet är tillräckligt enkelt) {

- Lös problemet direkt
- Returnera lösningen

} `else` {

- Dela upp problemet i ett eller fler mindre problem med samma struktur som ursprungsproblemet
- Lös de mindre problemen
- Kombinera resultatet till en lösning till ursprungsproblemet
- Returnera lösningen

}

1.1 Uttömmande sökning

Generera alla möjligheter

- Vanligt förekommande att behöva generera alla objekt som uppfyller något visst kriterium
  - Ordkedjor: Generera alla ord som skiljer sig åt i exakt en bokstav
- Ofta kan objekten genereras iterativt
- I många fall är det dock bättre att fundera på en rekursiv metod för att generera alla möjligheter

Delmängder

- Givet en mängd S kan vi bilda en delmängd till S genom att välja ett antal element från S
- Exempel:
  - $\{0, 1, 2\}$ är en delmängd av $\{0, 1, 2, 3, 4, 5\}$
  - $\{\text{dikdik, ibex}\}$ är en delmängd av $\{\text{dikdik, ibex}\}$
  - $\{A, G, C, T\}$ är en delmängd av $\{A, B, C, D, E, \ldots, Z\}$
  - $\{\} \subseteq \{a, b, c\}$
  - $\{\} \subseteq \{\}$

Generera delmängder

- Många viktiga problem i datalogi kan lösas genom att lista alla delmängder av en mängd S och hitta den "bästa" av dem.
- Exempel:
  - Du har en uppsättning sensorer på en autonom farkost som alla tar emot data
  - Vilken delmängd av sensorerna väljer du att lyssna till givet att var och en tar olika lång tid att läsa av?

16.6

Generera delmängder

$$\{0, 1, 2\}$$

$$\{\} \quad \{0\}$$
$$\{2\} \quad \{0, 2\}$$
$$\{1\} \quad \{0, 1\}$$
$$\{1, 2\} \quad \{0, 1, 2\}$$

16.7

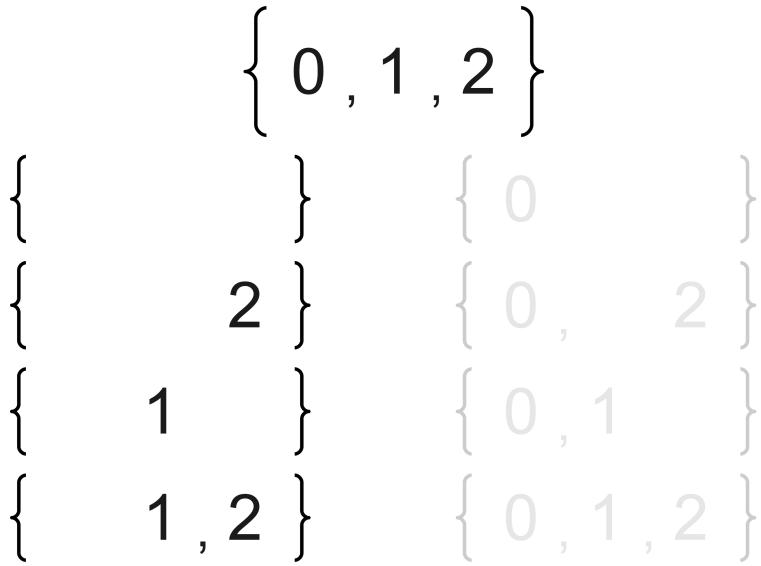
Generera delmängder

$$\{0, 1, 2\}$$

$$\{\} \quad \{0\}$$
$$\{2\} \quad \{0, 2\}$$
$$\{1\} \quad \{0, 1\}$$
$$\{1, 2\} \quad \{0, 1, 2\}$$

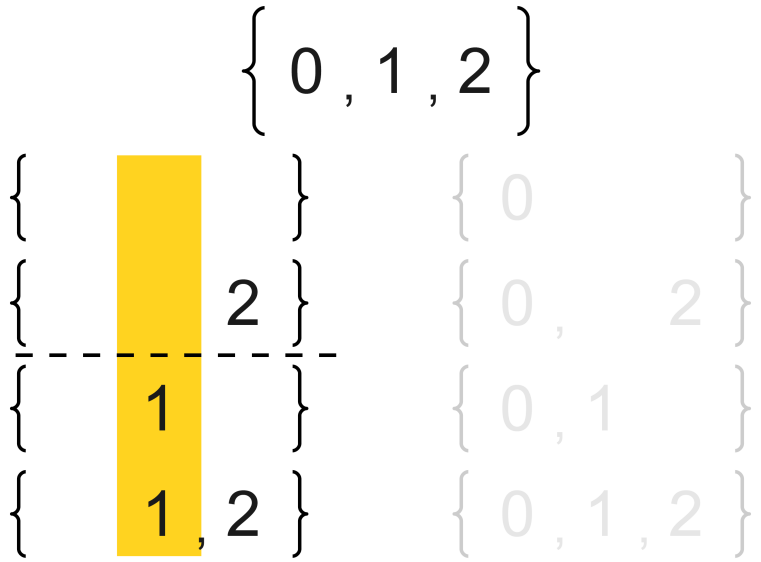
16.8

Generera delmängder



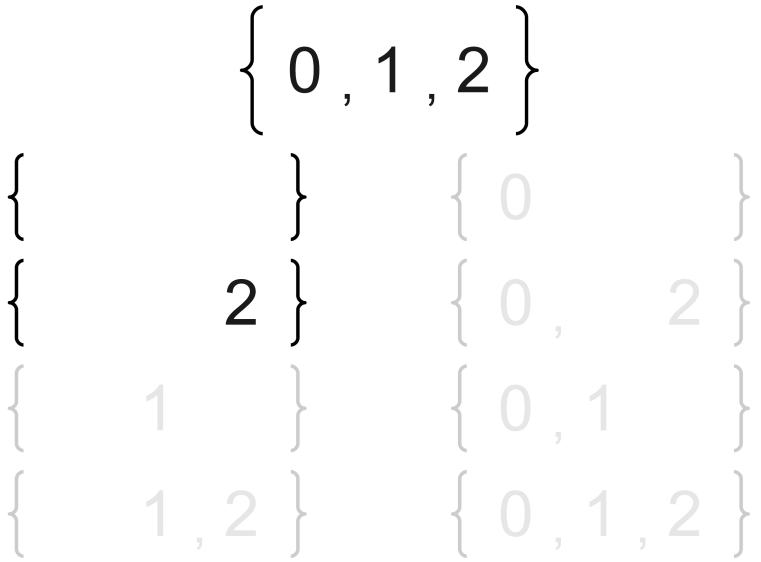
16.9

Generera delmängder



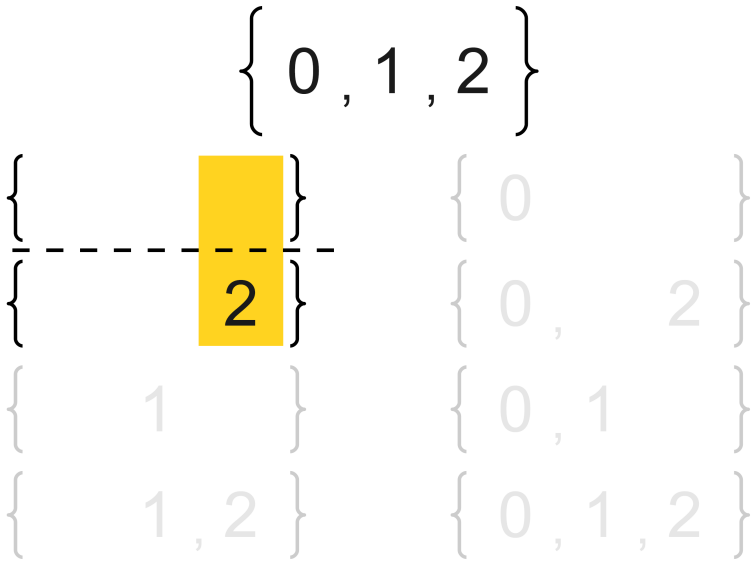
16.10

Generera delmängder



16.11

Generera delmängder



16.12

Generera delmängder

- Basfall:
  - Den enda delmängden av den tomma mängden är den tomma mängden
- Rekursivt fall:
  - Fixera något element x i mängden
  - Generera alla delmängder av mängden som bildas när x tas bort ur huvudmängden
  - Dessa delmängder är delmängder till ursprungsmängden
  - Alla mängder som bildas genom att lägga till x till dessa delmängder är delmängder av ursprungsmängden

16.13

Spåra rekursionen

{ A, H, I }

16.14

Spåra rekursionen

{ A, H, I }

{ H, I }

16.15

Spåra rekursionen

{ A, H, I }

{ H, I }

{ I }

16.16

Spåra rekursionen

{ A, H, I }

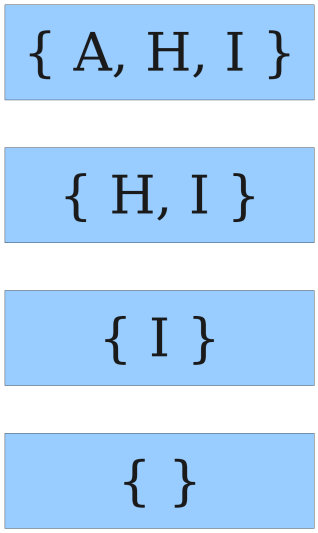
{ H, I }

{ I }

{ }

16.17

Spåra rekursionen



{ }

16.18

Spåra rekursionen

{ A, H, I }

{ H, I }

{ I }

{I}, { }

{ }

{ }

16.19

Spåra rekursionen

{ A, H, I }

{ H, I }

{H, I}, {H}, {I}, { }

{ I }

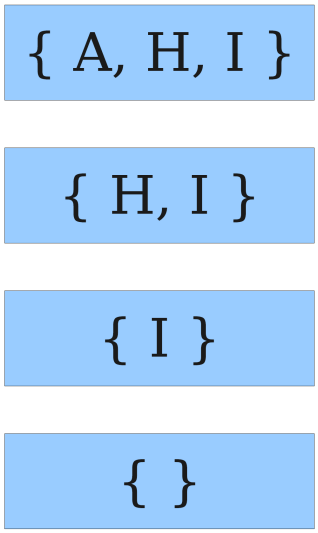
{I}, { }

{ }

{ }

16.20

Spåra rekursionen



16.21

Analys av metoden

- Hur många delmängder finns det av en mängd med n element?
- För varje element väljer vi om det ska vara med i delmängden eller inte
- Vi gör n val med 2 möjligheter för varje val, så det finns 2^n delmängder
- Den returnerade samlingen delmängder använder $\mathcal{O}(2^n)$ minne

16.22

Reducera minnesanvändningen

- I många fall behöver vi utföra någon operation på varje delmängd, men behöver inte spara delmängderna
  - Idé: Generera varje delmängd, behandla den och kasta sedan bort den
    - Fråga: Hur gör vi detta?

16.23

Permutationer

- Skriv en funktion `permute` som tar en strängparameter och matar ut alla möjliga omflyttningar av bokstäverna i strängen. Det spelar ingen roll i vilken ordning utmatningen av de olika omflyttningarna sker.
  - Exempel: `permute("MARTY")` matar ut följande sekvens av rader:

MARTY	MYRAT	ATYMR	RTMAY	TARMY	YMTAR
MARYT	MYRTA	ATYRM	RTMYA	TARYM	YMTRA
MATRY	MYTAR	AYMRT	RTAMY	TAYMR	YAMRT
MATYR	MYTRA	AYMTR	RTAYM	TAYRM	YAMTR
MAYRT	AMRTY	AYRMT	RTYMA	TRMAY	YARMT
MAYTR	AMRYT	AYRTM	RTYAM	TRMYA	YARTM
MRATY	AMTRY	AYTMR	RYMAT	TRAMY	YATMR
MRAYT	AMTYR	AYTRM	RYMTA	TRAYM	YATRM
MRTAY	AMYRT	RMATY	RYAMT	TRYMA	YRMAT
MRTYA	AMYTR	RMAYT	RYATM	TRYAM	YRMTA
MRYAT	ARMTY	RMTAY	RYTMA	TYMAR	YRAMT
MRYTA	ARMYT	RMTYA	RYTAM	TYMRA	YRATM
MTARY	ARTMY	RMYAT	TMARY	TYAMR	YRTMA
MTAYR	ARTYM	RMYTA	TMAYR	TYARM	YRTAM
MTRAY	ARYMT	RAMTY	TMRAY	TYRMA	YTMAR
MTRYA	ARYTM	RAMYT	TMRYA	TYRAM	YTMRA
MTYAR	ATMRY	RATMY	TMYAR	YMART	YTAMR
MTYRA	ATMYR	RATYM	TMYRA	YMATR	YTARM
MYART	ATRMY	RAYMT	TAMRY	YMRAT	YTRMA
MYATR	ATRYM	RAYTM	TAMYR	YMRTA	YTRAM

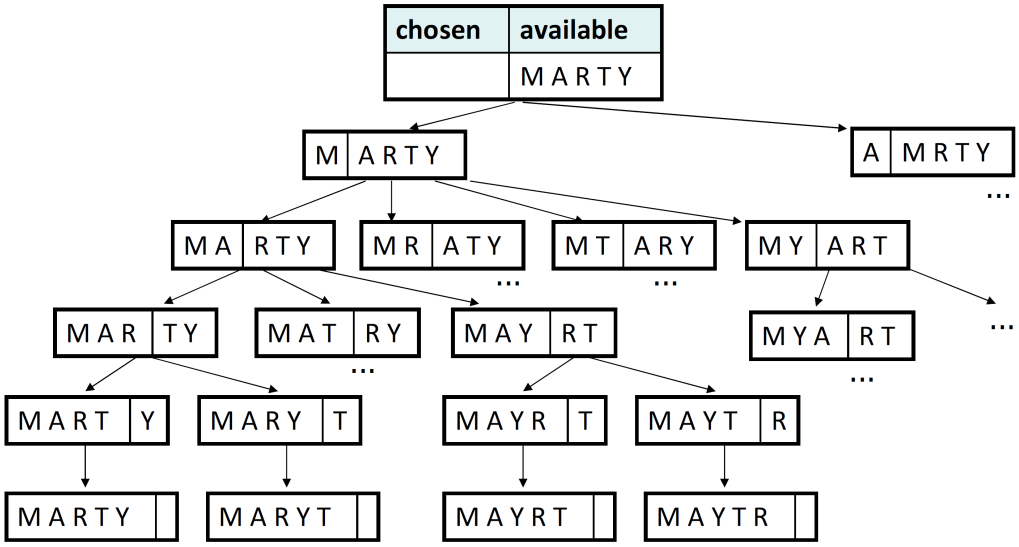
16.24

Granska problemet

- Tänk på varje permutation som en uppsättning val eller *beslut*
  - Vilken bokstav vill jag placera först?
  - Vilken bokstav vill jag placera på andra plats?
  - ...
  - Lösningsrymd: mängd av alla möjliga mängder av beslut att utforska
- Vi vill generera alla möjliga sekvenser av beslut
  - för (varje möjlig första bokstav):
  - för (varje möjlig andra bokstav):
  - för (varje möjlig tredje bokstav):
  - ...
  - skriv ut!
  - Detta är en djupet förstsökning

16.25

Beslutsträd



16.26

1.2 Backtracking

Backtracking

- En generell algoritm för att hitta lösningar till ett beräkningsproblem genom att testa dellösningar och sedan överge dem ("backtracking") om de inte passar.
  - en "brute force"-teknik (testar alla möjligheter)
  - ofta (men inte alltid) implementerad rekursivt
- Tillämpningar:
  - producera alla permutationer av en mängd värden
  - parsning av språk
  - spel: anagram, korsord, 8 queens, Boggle
  - kombinatorik och logikprogrammering

16.27

Backtracking-algoritmer

Generell psuedokod för backtrackingproblem:

- Utforska(val):
  - om det inte finns fler val: stanna
  - annars, för varje tillgängligt val *C*:
    - Välj *C*
    - Utforska resterande val
    - "O-välj" *C* om nödvändigt (backtrack)

16.28

Backtracking-strategier

- Ställ följande frågor vid användning av backtracking för att lösa ett problem:
  - Vad är det som utgör "valen" i det här problemet?
    - Vad är "basfallet"? (Hur vet jag när jag har slut på valmöjligheter?)
  - Hur "gör" jag ett val?
    - behöver jag skapa extra variabler för att komma ihåg mina val?
    - Behöver jag modifiera värdena hos existerande variabler?
  - Hur utforskar jag resterande val??
    - Behöver jag ta bort det gjorda valet från listan av val?
  - När jag är färdig med att utforska resterande val, vad gör jag då?
  - Hur gör jag ett val ogjort?

Permutationer igen

- Skriv en funktion `permute` som tar en strängparameter och matar ut alla möjliga omflyttningar av bokstäverna i strängen. Det spelar ingen roll i vilken ordning utmatningen av de olika omflyttningarna sker.
  - Exempel: `permute("MARTY")` matar ut följande sekvens av rader:
  - (På vilket sätt är det här problemet självlikformigt? Rekursivt?)

MARTY	MYRAT	ATYMR	RTMAY	TARMY	YMTAR
MARYT	MYRTA	ATYRM	RTMYA	TARYM	YMTRA
MATRY	MYTAR	AYMRT	RTAMY	TAYMR	YAMRT
MATYR	MYTRA	AYMTR	RTAYM	TAYRM	YAMTR
MAYRT	AMRTY	AYRMT	RTYMA	TRMAY	YARMT
MAYTR	AMRYT	AYRTM	RTYAM	TRMYA	YARTM
MRATY	AMTRY	AYTMR	RYMAT	TRAMY	YATMR
MRAYT	AMTYR	AYTRM	RYMTA	TRAYM	YATRM
MRTAY	AMYRT	RMATY	RYAMT	TRYMA	YRMAT
MRTYA	AMYTR	RMAYT	RYATM	TRYAM	YRMTA
MRYAT	ARMTY	RMTAY	RYTMA	TYMAR	YRAMT
MRYTA	ARMYT	RMTYA	RYTAM	TYMRA	YRATM
MTARY	ARTMY	RMYAT	TMARY	TYAMR	YRTMA
MTAYR	ARTYM	RMYTA	TMAYR	TYARM	YRTAM
MTRAY	ARYMT	RAMTY	TMRAY	TYRMA	YTMAR
MTRYA	ARYTM	RAMYT	TMRYA	TYRAM	YTMRA
MTYAR	ATMRY	RATMY	TMYAR	YMART	YTAMR
MTYRA	ATMYR	RATYM	TMYRA	YMATR	YTARM
MYART	ATRMY	RAYMT	TAMRY	YMRAT	YTRMA
MYATR	ATRYM	RAYTM	TAMYR	YMRTA	YTRAM

Lösning

```
// Outputs all permutations of the given string.
void permute(string s, string chosen = "") {
   if (s == "") {
      cout << chosen << endl; // base case: no choices left
   } else {
      // recursive case: choose each possible next letter
      for (int i = 0; i < s.length(); i++) {
         char c = s[i]; // choose
         s.erase(i, 1);
         permute(s, chosen + c); // explore
         s.insert(i, 1, c); // un-choose
      }
   }
}
```


Kombinationer

- Skriv en funktion `combinations` som tar en sträng s och ett heltal k och matar ut alla möjliga k-bokstavssträngar som kan bildas med hjälp av unika bokstäver från den strängen. Det spelar ingen roll i vilken ordning utmatningen av de olika kombinationerna sker.
  - Exempel: `combinations("GOOGLE", 3)` matar ut sekvensen av rader till höger:
  - För att förenkla problemet kan vi anta att strängen s innehåller minst k unika bokstäver.

EGL	LEG
EGO	LEO
ELG	LGE
ELO	LGO
EOG	LOE
EOL	LOG
GEL	OEG
GEO	OEL
GLE	OGE
GLO	OGL
GOE	OLE
GOL	OLG

Första lösningsförsök

```
// Outputs all unique k-letter combinations of the given string.
void combinations(string s, int length, string chosen = "") {
   if (length == 0) {
      cout << chosen << endl; // base case: no choices left
   } else {
      for (int i = 0; i < s.length(); i++) {
         if (chosen.find(s[i]) == string::npos) {
            char c = s[i];
            s.erase(i, 1);
            combinations(s, length - 1, chosen + c);
            s.insert(i, 1, c);
         }
      }
   }
}
```

- Problem: Skriver ut samma sträng flera gånger.

Lösning

```
// Outputs all unique k-letter combinations of the given string.
void combinations(string s, int length) {
   Set<string> found;
   combinHelper(s, length, "", found);
}

void combinHelper(string s, int length, string chosen, Set<string>& found) {
   if (length == 0 && !found.contains(chosen)) {
      cout << chosen << endl; // base case: no choices left
      found.add(chosen);
   } else {
      for (int i = 0; i < s.length(); i++) {
         if (chosen.find(s[i]) == string::npos) {
            char c = s[i];
            s.erase(i, 1);
            combinHelper(s, length - 1, chosen + c, found);
            s.insert(i, 1, c);
         }
      }
   }
}
```


Tärningskast

- Skriv en funktion `diceRoll` som tar in ett heltal representerande ett antal 6-sidiga tärningar att kasta och mata ut alla möjliga kombinationer av värden som kan uppträda på tärningarna.

```
diceRoll(2);
```

```
{1, 1}  {3, 1}  {5, 1}
{1, 2}  {3, 2}  {5, 2}
{1, 3}  {3, 3}  {5, 3}
{1, 4}  {3, 4}  {5, 4}
{1, 5}  {3, 5}  {5, 5}
{1, 6}  {3, 6}  {5, 6}
{2, 1}  {4, 1}  {6, 1}
{2, 2}  {4, 2}  {6, 2}
{2, 3}  {4, 3}  {6, 3}
{2, 4}  {4, 4}  {6, 4}
{2, 5}  {4, 5}  {6, 5}
{2, 6}  {4, 6}  {6, 6}
```

```
diceRoll(3);
```

```
{1, 1, 1}
{1, 1, 2}
{1, 1, 3}
{1, 1, 4}
{1, 1, 5}
{1, 1, 6}
{1, 2, 1}
{1, 2, 2}
   ...
{6, 6, 4}
{6, 6, 5}
{6, 6, 6}
```

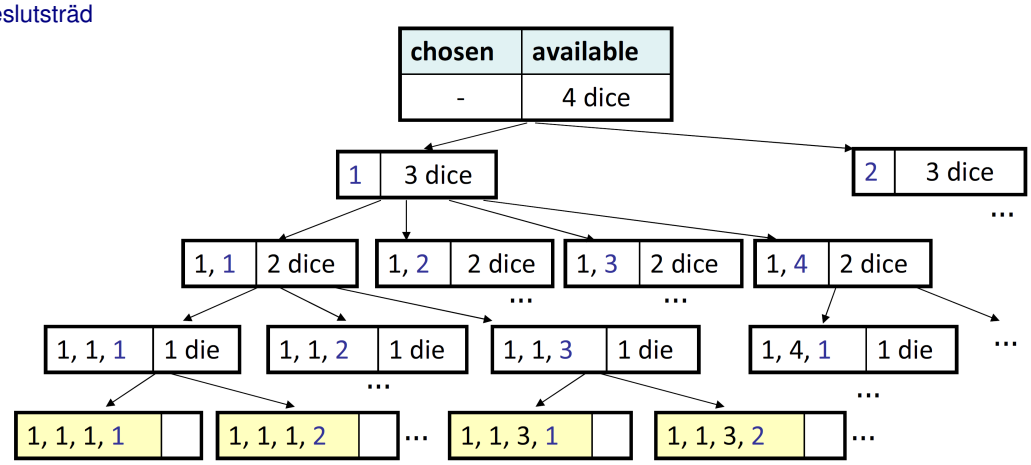
16.35

Granska problemet

- Vi vill generera alla möjliga sekvenser av beslut
  - för (varje möjlig första bokstav):
  - för (varje möjlig andra bokstav):
  - för (varje möjlig tredje bokstav):
  - ...
  - skriv ut!
  - Detta är en djupet förstsökning
- Hur kan vi fullständigt utforska en så stor sökrymd?

16.36

Beslutsträd



16.37

Lösning

```
// Prints all possible outcomes of rolling the given
// number of six-sided dice in {#, #, #} format.
void diceRolls(int dice) {
   vector<int> chosen;
   diceRollHelper(dice, chosen);
}

// private recursive helper to implement diceRolls logic
void diceRollHelper(int dice, vector<int>& chosen) {
   if (dice == 0) {
      cout << chosen << endl; // base case
   } else {
      for (int i = 1; i <= 6; i++) {
         chosen.add(i); // choose
         diceRollHelper(dice - 1, chosen); // explore
         chosen.remove(chosen.size() - 1); // un-choose
      }
   }
}
```

16.38

Tärningskastsumma

- Skriv en funktion `diceSum` som liknar `diceRoll` men som också tar en en summa och bara skriver ut de kombinationer som adderar ihop till exakt den summan.

```
diceSum(2, 7);
```

```
{1, 6}
{2, 5}
{3, 4}
{4, 3}
{5, 2}
{6, 1}
```

```
diceSum(3, 7);
```

```
{1, 1, 5}
{1, 2, 4}
{1, 3, 3}
{1, 4, 2}
{1, 5, 1}
{2, 1, 4}
{2, 2, 3}
{2, 3, 2}
{2, 4, 1}
{3, 1, 3}
{3, 2, 2}
{3, 3, 1}
{4, 1, 2}
{4, 2, 1}
{5, 1, 1}
```

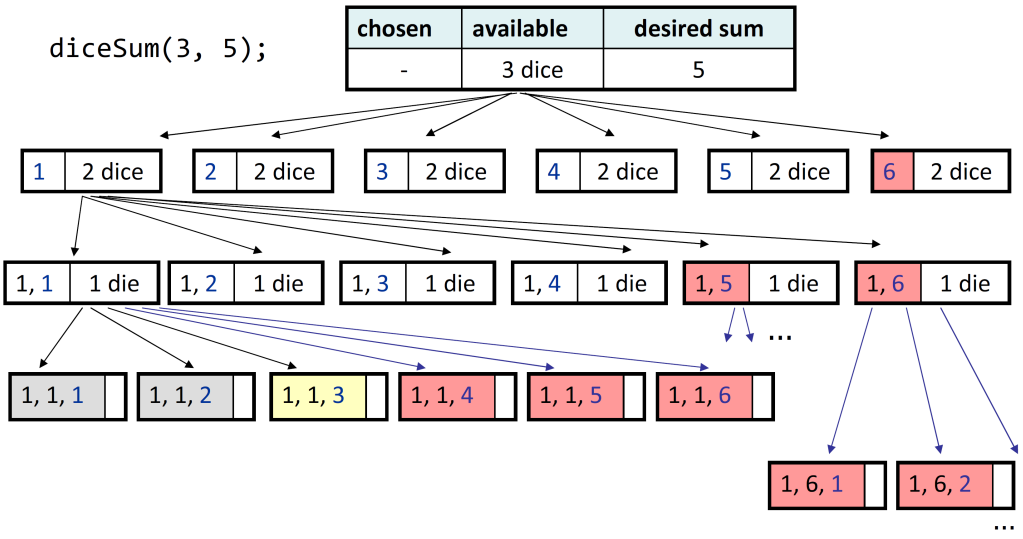
16.39

Minimal modifiering

```
// Prints all possible outcomes of rolling the given
// number of six-sided dice in {#, #, #} format.
void diceRolls(int dice, int desiredSum) {
   vector<int> chosen;
   diceSumHelper(dice, desuredSum, chosen);
}
void diceRollHelper(int dice, int desiredSum, vector<int>& chosen) {
   if (dice == 0) {
      if (sumAll(chosen) == desiredSum) {
         cout << chosen << endl; // base case
      }
   } else {
      for (int i = 1; i <= 6; i++) {
         chosen.add(i); // choose
         diceSumHelper(dice - 1, desiredSum, chosen); // explore
         chosen.remove(chosen.size() - 1); // un-choose
      }
   }
}
int sumAll(const vector<int>& v) {
   int sum = 0;
   for (int k : v) { sum += k; }
   return sum;
}
```

16.40

Slösaktigt beslutsträd



16.41

Optimeringar

- Vi behöver inte besöka varje gren av beslutsträdet.
  - Vissa grenar kommer uppenbarligen inte att leda till en lösning.
  - Vi kan avsluta, eller beskära (prune), dessa grenar
- Ineffektiviteter i vår lösning:
  - Ibland är den nuvarande summan redan för hög. (Att slå en etta skulle överskrida den önskade summan.)
  - Ibland är den nuvarande summan redan för låg. (Att slå sexor med alla kvarvarande tärningar skulle inte räcka för att nå den önskade summan.)
  - När vi är färdiga måste koden beräkna summan hela tiden.

16.42

Lösning

```
void diceSum(int dice, int desiredSum) {
   vector<int> chosen;
   diceSumHelper(dice, 0, desiredSum, chosen);
}

void diceSumHelper(int dice, int sum, int desiredSum, vector<int>& chosen) {
   if (dice == 0) {
      if (sum == desiredSum) {
         cout << chosen << endl; // base case
      }
   } else if (sum <= desiredSum && sum + 6*dice >= desiredSum) {
      for (int i = 1; i <= 6; i++) {
         chosen.add(i); // choose
         diceSumHelper(dice - 1, sum + i, desiredSum, chosen); // explore
         chosen.remove(chosen.size() - 1); // un-choose
      }
   }
}
```

16.43