Föreläsning 14
Recursive search
TDDD86: DALP

Utskriftsversion av Föreläsing i *Datastrukturer, algoritmer och programmeringsparadigm*
10 October 2023

IDA, Linköpings universitet

14.1

Content

Contents

14.2

1 Recursive search

Recursive problem solving

if (the problem is simple enough) {

- Solve the problem directly
- Return the solution

} else {

- Divide the problem into one or several similar smaller problems
- Solve the smaller problems
- Combine the results to get a solution to the original problem
- Return the solution

}

14.3

1.1 Exhaustive search

Generate all possibilities

- It is not rare that one needs to generate all objects satisfying a given constraint
  - Word chains: Generate all words that only differ in a single letter
- The objects can often be generated iteratively
- In several cases it is better to think about a recursive method to generate all the possibilities.

14.4

Subsets

- Given a set S, we can generate a subset of S by chosing a number of elements from S
- Exampel:
  - $\{0, 1, 2\}$ is a subset of $\{0, 1, 2, 3, 4, 5\}$
  - {dikdik, ibex} is a subset of {dikdik, ibex}
  - {A, G, C, T} is a subset of {A, B, C, D, E, . . . , Z}
  - $\{\} \subseteq \{a, b, c\}$
  - $\{\} \subseteq \{\}$
- Many important problems in Computer Science can be solved by listing all possible subsets of a set S and by finding the "best" one.

14.5

Generate subsets

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

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

14.6

Generate subsets

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

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

14.7

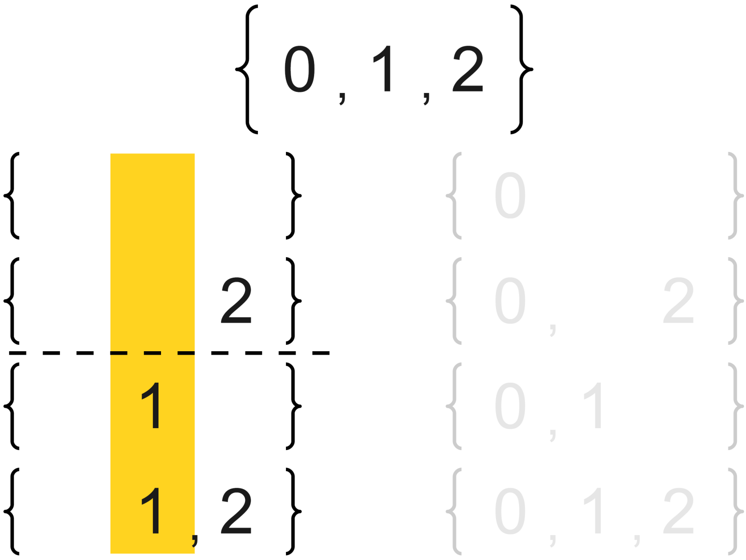
Generate subsets

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

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

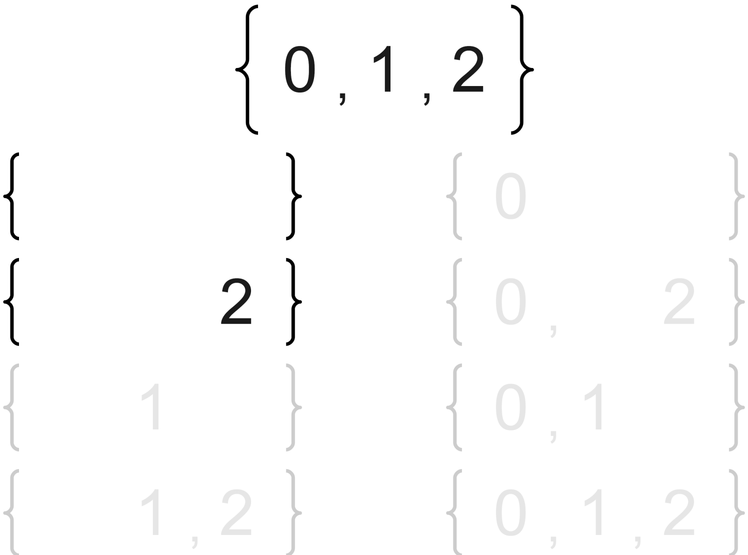
14.8

Generate subsets



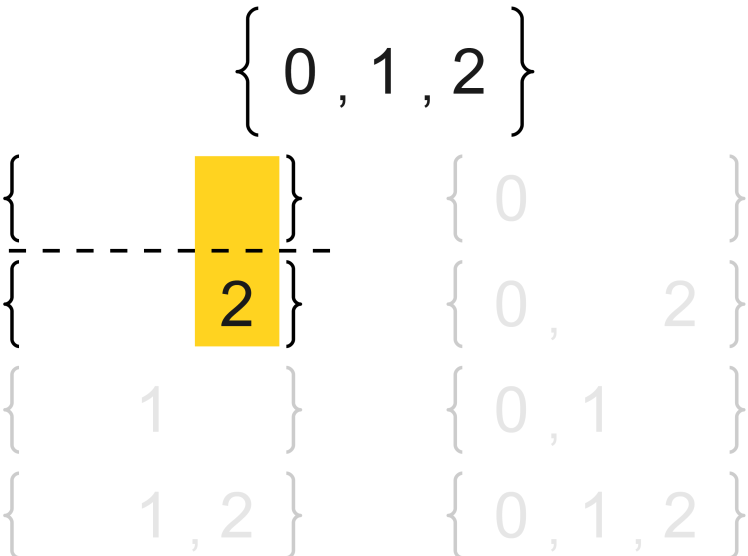
14.9

Generate subsets



14.10

Generate subsets



14.11

Generate subsets

- Base case:
  - The only subset of the empty set is the empty set
- Recursive case:
  - Choose an element x in the set original set
  - Generate all subsets of the set obtained by excluding x from the set
  - These subsets are also subsets to the original set
  - All subsets obtained by adding x are also subsets to the original set

14.12

Follow the recursion

{ A, H, I }

14.13

Follow the recursion

{ A, H, I }

{ H, I }

14.14

Follow the recursion

{ A, H, I }

{ H, I }

{ I }

14.15

Follow the recursion

{ A, H, I }

{ H, I }

{ I }

{ }

14.16

Follow the recursion

{ A, H, I }

{ H, I }

{ I }

{ }   { }

14.17

Follow the recursion

{ A, H, I }

{ H, I }

{ I }   {I}, { }

{ }   { }

14.18

Follow the recursion

{ A, H, I }	
{ H, I }	`{H, I}, {H}, {I}, { }`
{ I }	`{I}, { }`
{ }	`{ }`

14.19

Follow the recursion

{ A, H, I }	`{A, H, I}, {A, H}, {A, I}, {A}` `{H, I}, {H}, {I}, { }`
{ H, I }	`{H, I}, {H}, {I}, { }`
{ I }	`{I}, { }`
{ }	`{ }`

14.20

Analyzing the method

- How many subsets are there in a set with n elements?
- For each element, we choose if it will be part of the subset or not
- We make n choices with 2 possible outcomes for each choice. This results in 2^n subsets.
- The returned set of subsets will use $\mathcal{O}(2^n)$ in memory

14.21

Reducing memory usage

- We need often to perform some operations on each subset, without needing to save them.
  - Idea: Generate each subset, handle it, then throw it away
    - Question: How do we do that?

14.22

Permutations

- Write a function `permute` that takes a string parameter and that outputs all possible permutations of the letters in the string. The order in which the permutations are output does not matter.

– Exampel: `permute("MARTY")` outputs the following sequence:

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

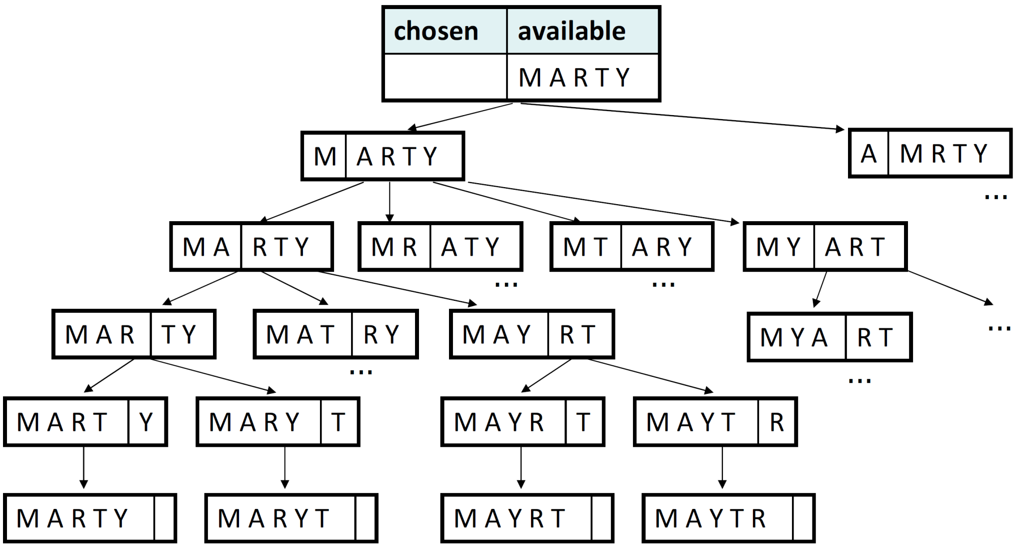
14.23

Let's look at the problem

- Think about each permuation as a sequence of choices or *decisions*
  - Which letter should be chosen first?
  - Which letter should be chosen second?
  - ...
  - Solutions' space: set of all possible sets of decisions to be explored.
- We want to generate all possible sequences of decisions
  - for (each possible first letter):
  - for (each possible second letter):
  - for (each possible third letter):
  - ...
  - output the permuation!
  - This amounts to a depth-first search

14.24

Decision tree



14.25

1.2 Backtracking

Backtracking

- A general algorithm to find solutions to a problem by testing solutions to subproblems and giving up on them ("backtracking") if they turn out to be not suitable

- a "brute force"-technique (tests all possibilites)
- Often (but not always) implemented recursively

- Applications:
  - produce all permutations of a set of values
  - parsing a language
  - games: anagrams, crosswords, 8 queens, Boggle
  - Combinatorial and logic programming

14.26

Backtracking algorithms

General pseudo-code for a backtracking algorithm:

- Explore(choice):
  - if there are no further choices: stop
  - otherwise, for each possible choice C:
    - choose C
    - Explore the remaining choices
    - "Unchoose" C if needed (backtrack)

14.27

Backtracking strategies

- Ask the following questions when you use backtracking to solve a problem:
  - What represents a "choice" in this problem?
    - What is the "base case(s)"? How do I know there are no more choices left?
  - How do I "choose"?
    - do I need extra variables to remember my choices?
    - do I need to modify the values of the existing variables?
  - How do I explore the remaining choices?
    - Do I need to remove the made choices from the list of choices?
  - what should I do when I am done exploring the remaining choices?
  - How do I "unchoose" a choice?

14.28

Permutations revisited

- Write a function `permute` that takes a string parameter and that outputs all possible permutations of the letters in the string. The order in which the permutations are output does not matter.
  - Exampel: `permute("MARTY")` outputs the following sequence:
  - (In what way is this problem uniform? recursive?)

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

14.29

Solution

```
// 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
      }
   }
}
```

14.30

Combinations

- Write a function `combinations` that takes a string and a natural number k and that outpus all possible k-long-strings that can be obtained from unique letters from the string. The order in which the resulting combinations are output does not matter.
  - Exampel: `combinations("GOOGLE", 3)` outputs the sequence of lines to the right.
  - To simplify the problem, we assume the string contains at least k unique letters.

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

14.31

First attempt

```
// 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: writes the same string several times.

14.32

Solution

```
// 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);
         }
      }
   }
}
```

14.33

Rolling dices

- Write a function `diceRoll` that takes a natural number representing a number of 6-sided dices to be rolled. Output all possible combinations of values that can be obtained.

```
diceRoll(2);

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

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

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

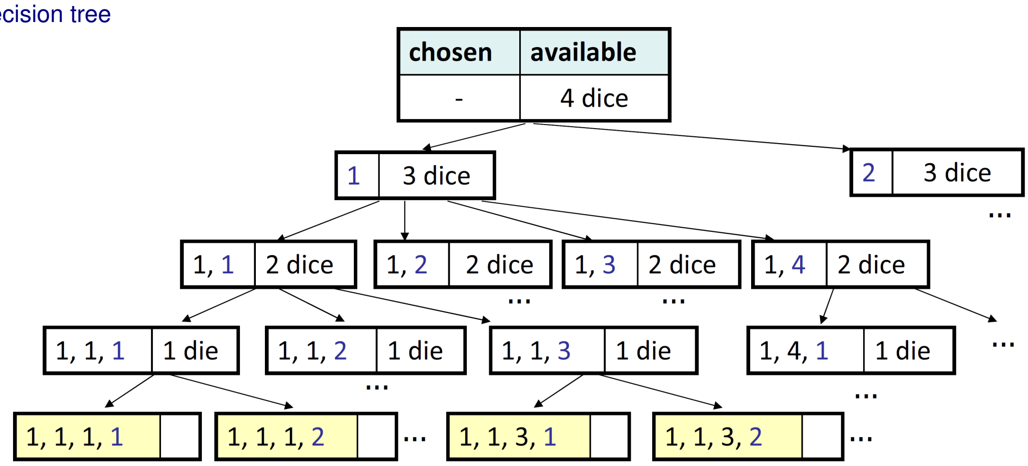
14.34

Study the problem

- We want to generate all possible sequences of decisions
  - for (each possible first letter):
  - for (each possible second letter):
  - for (each possible third letter):
  - ...
  - output!
  - This is a depth-first search
- How can we exhaustively explore this large search space?

14.35

Decision tree



14.36

Solution

```
// 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
      }
   }
}
```

14.37

DiceSum

- Write a function `diceSum` that resembles `diceRoll` but that also takes a sum and that only writes those combinations whose sum is the given sum.

```
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}
```

14.38

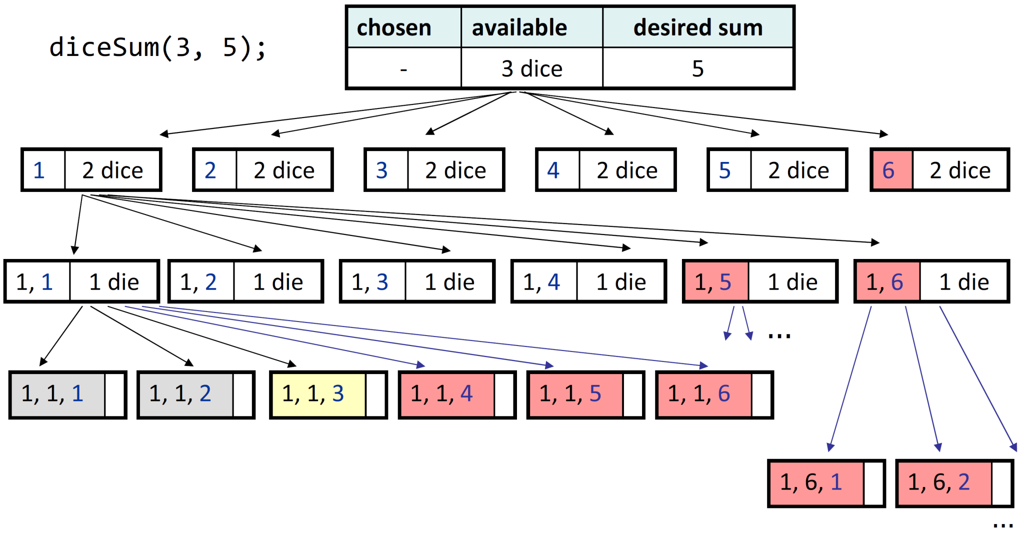
Minimal modifications

```
// 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;
}
```

14.39

Wasteful decision tree



14.40

Optimizations

- We do not need to explore each branch in the tree
  - Some branches will obviously not give any solution.
  - We can terminate or "prune" these branches
- Innefficiencies in the previous solution:
  - The current sum is sometimes already too high. (even a 1 in the next roll would exceed the targeted sum)
  - The current sum is sometimes too low. (even sixes in all remaining rolls would not be enough to obtain the targeted sum.)
  - Each time there are no more choices, the sums are computed.

14.41

Solution

```
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 + 1 * dice <= 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
        }
    }
}
```

14.42