Lecture 16
Recursive search

TDDD86: DALP

Print version of the lecture *Data structures, algorithms and programming paradigms*
4 november 2016

Jalil Boudjadar, Tommy Färnqvist. IDA, Linköping University

16.1

Content

Innehåll

16.2

1 Recursive search

Solution for recursive problems

`if` (the problem is simple enough) {

- Solve the problem directly
- Return the solution

} `else` {

- Divide the problem in one or more minor problems with the same structure as the original problem
- Solve the minor problem
- Combine the result with the solution of the next recursion, until reaching the original problem
- Return the solution

}

16.3

1.1 Exhaustive search

Generate all opportunities

- Usually, you need to generate all objects that meet a given criterion
  - Word chains: Generate all words that differ in exactly one letter
- Often, the objects are generated iteratively
- In many cases it is better to consider a method for recursive generation of the opportunities

16.4

Subsets

- Given a set S, we can form a subset of S by selecting a number of elements from S
- Example:
  - {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}
  - $\{\} \subseteq \{a, b, c\}$
  - $\{\} \subseteq \{\}$

16.5

Generate subsets

- Many important problems in computer science can be solved by listing all subsets of a set S and find the “best” of them.
- Example:
  - You have a set of sensors on an autonomous craft that all collect data
  - Which subset of the sensors you choose to listen to given that each one takes a different time to read?

16.6

Generate subsets

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

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

16.7

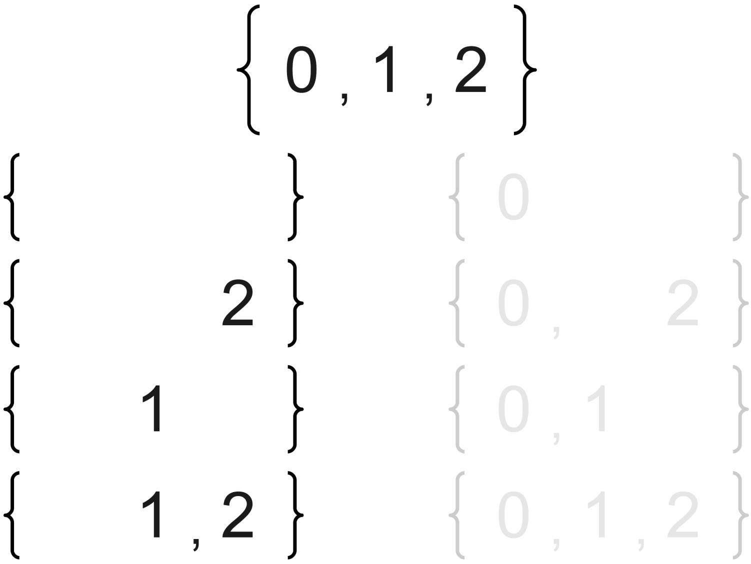
Generate subsets

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

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

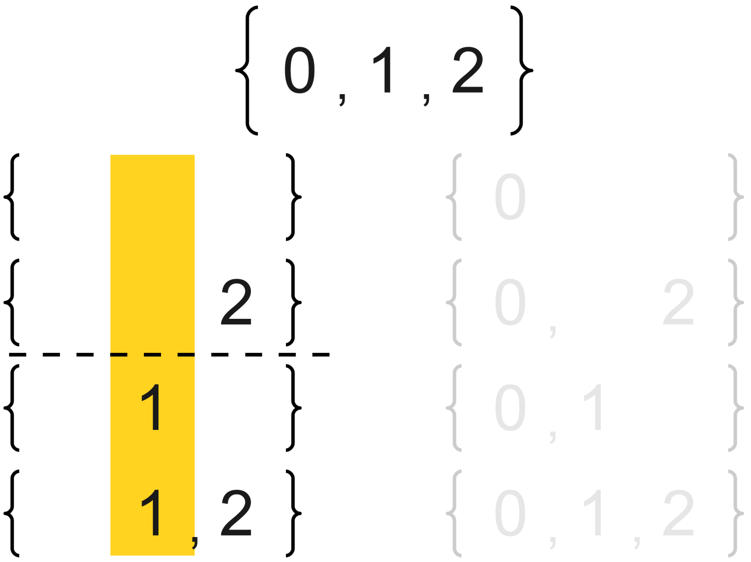
16.8

Generate subsets



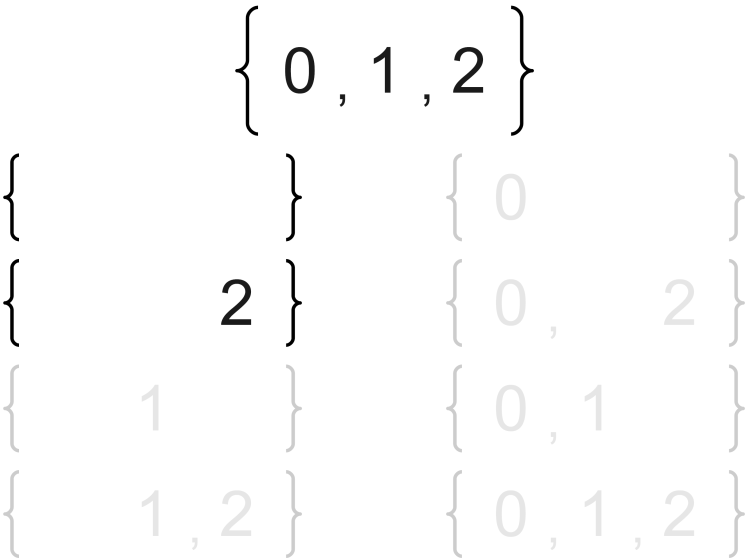
16.9

Generate subsets



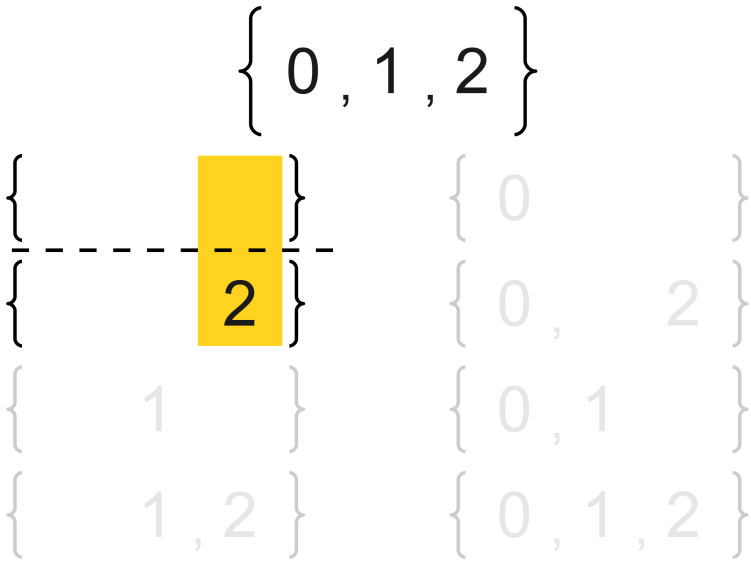
16.10

Generate subsets



16.11

Generate subsets



16.12

Generate subsets

- Basic case:
  - The only subset of the empty set is the empty set
- Recursive case:
  - Choose any element X in the set
  - Generate all subsets of the given set when x is removed from the set
  - These subsets are subsets of the origin set
  - All sets formed by adding the x to these subsets are subsets of the original set

16.13

Track the recursion

{ A, H, I }

16.14

Track the recursion

{ A, H, I }

{ H, I }

16.15

Track the recursion

{ A, H, I }

{ H, I }

{ I }

16.16

Track the recursion

{ A, H, I }

{ H, I }

{ I }

{ }

16.17

Track the recursion

{ A, H, I }

{ H, I }

{ I }

{ } { }

16.18

Track the recursion

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

16.19

Track the recursion

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

16.20

Track 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}, { }
{ }	{ }

16.21

Analysis of the method

- How many subsets exist for a set of n elements?
- For each element, we choose whether it will be included in the subset or not
- We do n choice with 2 possibilities for each choice, so there are 2^n subsets
- The returned collection of subsets use $\mathcal{O}(2^n)$ memory

16.22

Reducing the memory utilization

- In many cases, we need to perform an operation on each subset but we do not need to save the subsets
  - Idea: Generate each subset, treat it and throw it away
    - Question: How do we do this?

16.23

Permutations

- Write a function `permute` which takes a string parameter and outputs all possible rearrangements of the letters in the string. It does not matter in which order the output of the various displacements occur.
  - Example: `permute("MARTY")` outputs the following sequence of lines:

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

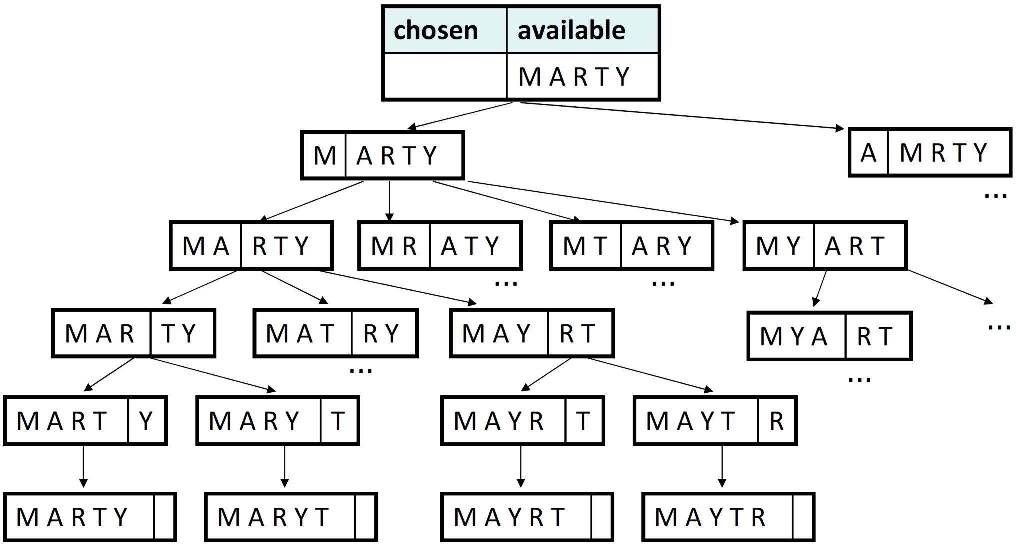
16.24

Reviewing the problem

- Think of each permutation as a set of choices or *decision*
  - Which letter I will place first?
  - Which letter I will put in the second place?
  - ...
  - Solution Space: set of all possible sets of the decision to explore
- We generate all possible sequences of decisions
  - for (each possible initial letter):
  - for (each possible second letter):
  - for (each possible third letter):
  - ...
  - print!
  - This is a depth-first search

16.25

Decision trees



16.26

1.2 Backtracking

Backtracking

- A general algorithm for finding solutions to a computational problem by testing partial solutions and then abandon them ("backtracking") if they do not fit
  - en "brute force"-technique (test all possibilities)
  - often (but not always) implemented recursively
- Applications:
  - produce all permutations of a set of values
  - parse the language
  - Game: anagrams, crosswords, 8 queens, Boggle
  - Combinatorics and logic programming

16.27

Backtracking-algorithms

General pseudo-code for back-tracking problems:

- Explore (choice):
  - if there is no more choice: stay
  - otherwise, for each available choice *C*
    - Select *C*
    - Explore the remaining choices
    - "deselect" *C* if necessary (backtrack)

16.28

Backtracking strategies

- Ask the following questions when using backtracking to solve a problem:
  - What determines the "choices" in this problem?
    - What is the "base case"? (How do I know when I run out of choice possibilities?)
  - How "do" I do a choice?
    - Do I need to create additional variables to remember my selection?
    - Do I need to modify the values of existing variables?
  - How do I explore the remaining choices??
    - Do I need to remove the selection made from the list of choices?
  - When I finish exploring the remaining choices, what should I do?
  - How do I make a choice undone?

16.29

Permutations again

- Write a function `permute` which takes a string parameter and outputs all possible rearrangements of the letters in the string. It does not matter in which order the output of the various displacements occur.
  - Example: `permute("MARTY")` outputs the following sequence of cases:
  - (which way leads the problem to be 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

16.30

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
      }
   }
}
```

16.31

Combinations

- Write a function `combinations` which takes a string s and an integer k, and outputs all possible strings having k letters. Strings can be formed by different letters from the original string. The order in which the output of the different combinations occurs is not important.
  - Example: `combinations("GOOGLE", 3)` outputs the sequence of cases in the right:
  - To simplify the problem we can assume that the string s 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

16.32

First solution 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: prints the same string many times.

16.33

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);
         }
      }
   }
}
```

16.34

Dice Roll

- Write a function `diceRoll` which takes in an integer representing a number of six-sided dice to throw and outputs all possible combinations of values that can appear on the dice.

```
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}
```

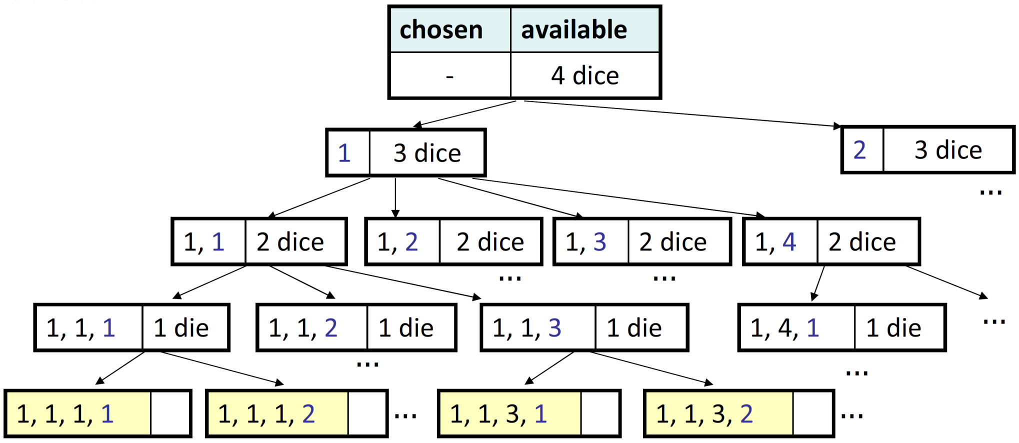
16.35

Reviewing the problem

- We generate all possible sequences of decisions
  - for (each possible initial letter):
  - for (each possible second letter):
  - for (each possible third letter):
  - ...
  - print!
  - This is a depth-first search
- How can we fully explore such a large search space?

16.36

Decision tree



16.37

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
      }
   }
}
```

16.38

Sum of the dice roll

- Write a function `diceSum` which is similar to `diceRoll` but takes also a number representing the *sum* and outputs the combinations having a summation equal to *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}
```

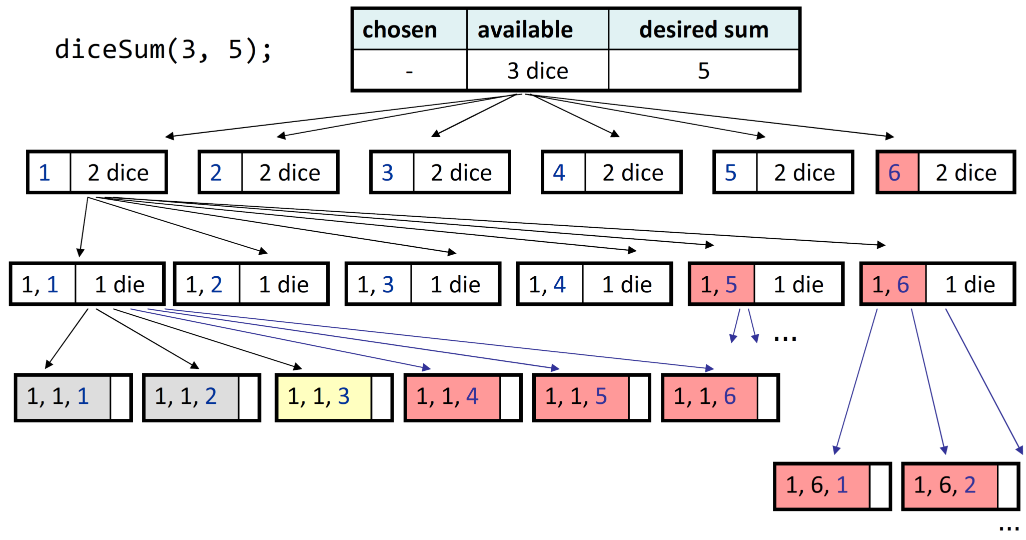
16.39

Minimal modification

```
// 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

Wasteful decision tree



16.41

Optimization

- We do not need to visit each branch of the decision tree.
  - Some branches will obviously not be added to a solution.
  - We can terminate, or crop (prune), these branches.
- Inefficiencies in the solution:
  - Sometimes the current sum is already too high. (Reaching one would exceed the desired sum.)
  - Sometimes the current sum is too low. (any remaining dice would not be enough to achieve the desired balance.)
  - When we finish, the code must always produce the sum.

16.42

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 <= 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