Algorithms

CMPT 115/117 lecture slides

Notes written by Mark Eramian, Ian McQuillan, Michael Horsch, Lingling Jin, and Dmytro Dyachuk

Objectives

By the end of this lecture topic, you are expected to be able to

1. explain the relationships between problems, algorithms, and functions
2. define what pseudocode is as well as its components
3. justify the benefits of writing pseudocode of an algorithm before implementing it
4. represent an algorithm logic using pseudocode
5. count the number of primitive operations of a given piece of pseudocode
6. compare the efficiencies of different algorithms using their asymptotic complexities (Big-O notations)
7. analyze the best and worst case complexities of an algorithm
8. define intractable and undecidable problems in terms of the computational complexity

Problems vs. Algorithms

The difference between a *problem* and an *algorithm* is subtle but important.

Definition

A *problem* is a description which associates inputs to output states.

Problem *sort*(*list*, *n*)
Sort the array list of integers into ascending order.

Pre: list is the array of integers to be sorted, of length n
Post: list contains the same elements, rearranged into ascending order.

Problems

- The **pre**conditions describe the inputs to the problem.
- The **post**conditions describe the resulting states of the variables.
- A problem statement itself does not indicate *how* to solve the problem.

Algorithms

Definition

An *algorithm* describes a sequence of steps which, when carried out, solves a problem (by meeting the postconditions upon completion).

- There can be more than one algorithm which solves the same problem.
- For example, *bubble sort*, *selection sort*, and *mergesort* are three algorithms which solve the sorting problem.
- There may be various advantages and disadvantages of each type of algorithm, but all solve the problem.

Example Pseudocode Algorithm

Algorithm *quickSort*(*list*, *n*)
Sort the array list of integers into ascending order.

Pre: *list* :: **shared** *List***OfInteger**
n :: **Integer** -- the length of *list*
Post: *list* contains the same elements, rearranged into ascending order.

if **not**(*isEmpty*(*list*))
then
  Integer pivot ← *first*(list)
  *List***OfInteger** smaller ← *selectSmaller*(*list*, *pivot*)
  *List***OfInteger** larger ← *selectLarger*(*list*, *pivot*)
  quickSort(*smaller*)
  quickSort(*larger*)
  list ← *glueTogether*(*smaller*,*pivot*,*larger*)

Functions

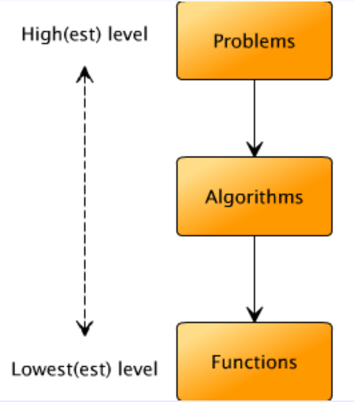
- The Problem versus Algorithm comparison works equally well for subroutines, or functions, of a larger program.
- If we are writing a spreadsheet application, and we would like to take a column of marks and sort them into ascending order, we could call the *sort* function.
- If we know the *sort* function solves the sorting problem, we don't need to know *how* the algorithm, or function, works.
- The details of the function can be **hidden**.
- If the function *sort* implements bubble sort, we could replace its contents with selection sort and the spreadsheet app would still work!

Solving Problems vs Implementing Solutions

- A pseudocode algorithm is easier to write than a complete program (in any language)
- You can solve most software design problems using pseudocode (C.E.R.A.R.)
- You can compare different solutions to a problem before you implement any of them.
- After you're happy with your pseudocode algorithm, you can implement it (i.e., convert it to source code)
- Implementing a well-understood pseudocode algorithm allows you to focus on technical aspects of programming, after you've done all the software design problem-solving.

Summary

Levels of Abstraction:



Pseudocode

- English-like representation of algorithm logic
- Independent of implementation language
- Describes task solution without unnecessary detail (e.g. error handling)

Variables

- We represent data structures and algorithms with pseudocode.
- We may not explicitly declare primitive data items. E.g.:

  $count \leftarrow 1$

  just implicitly defines the integer `count`. We may omit the type if it is obvious from the context.
- Record types ("structs"), however, are declared:

  Student
      firstName
      lastName
      studentNumber
  end *Student*

Elements of Pseudocode
Variables

- Conventions for variable names:
  1. Use **meaningful** variable names.
  2. Do not use single-character names, except for stylized purposes like loop counters
  3. Do not use generic names: *count*, *sum*, *row*
  4. Use descriptive names: *pixelCount*, *rotationMatrixRow*
  5. Abbreviations are OK: *pixCnt*, *rotMatRow*
  6. Use camelCaseWords to ease readability!
- Choice of variable names is critical to the readability of pseudocode (and actual code as well!).

Elements of Pseudocode
Algorithm Header

- Algorithms will begin with a header that contains its name, parameters, description, pre/post-conditions, and the return condition.

Algorithm *search* (*list*, *key*, *location*)
Search array for a specific item. Return index of its location.

pre: *list* :: **ArrayOf**<u>something</u>
key :: <u>something</u> -- data to search for
location :: **shared** -- <u>location</u>
post: location contains matching index
or undefined if not found
return: true if found, **false** if not found.

Elements of Pseudocode

Algorithm Header

- Description is a short statement about what the algorithm does.
- **Pre**conditions describe requirements for the parameters - the expected state of the input variables.
- **Post**conditions describe actions taken and state of output parameters after the algorithm's completion.
- **Return** condition describes what the program returns, if anything, and the meaning of the possible return values.

Pre-and-Post
Exercise

1. Problem: calculate the square root of a given number N.
   - Pre:
   - Post:
2. Problem: return the smallest number in an array.
   - Pre:
   - Post:

Elements of Pseudocode

Exercise: Algorithm Header

Identify the SIX elements of the algorithm header: name, parameters, description, pre/post-conditions, and the return condition.

Algorithm *binarySearch*(*arr*, *key*)
Search array for a specific item. Return index of it's location.

pre: *arr* :: **ArrayOfInteger**[*n*] -- sorted in ascending order
    key :: **Integer** to search for in *arr*
post: *arr* is unchanged
return: Integer index of *key* in *arr*, -1 if not found

lo ← 0
hi ← n-1
while (*lo* ≤ *hi*)
    mid ← (*lo* + *hi*) / 2
    if (key < *arr*[*mid*])   *hi* ← *mid* - 1
    else if (key > *arr*[*mid*]) *lo* ← *mid* + 1
    else return *mid*;
return -1;

Elements of Pseudocode

Statements

- A statement is a line of pseudocode that describes an action.
- Our pseudocode has three kinds of statement constructs
  - Sequences of statements, blocks
  - Loop statement
  - Conditional statement
- In pseudocode, we emphasize concepts, not syntax; but some consistency is helpful for communications. Follow these conventions!

Elements of Pseudocode

Sequences of Statements

- A sequence is one or more statements that don't alter the execution path within the algorithm.
- A sequence may include an invocation of another algorithm.
- A sequence of statements may contain types of statements including, not limited to:
  - Assignment
  - Input (console or file)
  - Output (console or file)
  - Arithmetic expressions
  - Higher-level English descriptions of actions

Elements of Pseudocode

Loop Statements

- loop statements denote repetition of a block of code.
- A guarded loop statement consists of the word `while` followed by a condition.

```
i ← 0
while (i < 10) do
    print i
    i ← i + 1
done
```

We may also write counted loops in a more C-style:

```
for i from 0 to 10 by 1 do
    print i
done
```

If we omit the **by** part, the increment/decrement is assumed to be 1.

Elements of Pseudocode

Conditional Statement

- The conditional statement (“if” statement) tests a condition, and executes different code depending on the outcome.

  > **if** (condition)
  > **then** sequence1
  > **else** sequence2

- Exercise:
  What would the pseudocode be for printing the string "positive" if the integer *i* is positive and "negative" if it is negative?

Practical compromises

On lecture slides, we'll write a conditional statement like this

```
if (condition)
then sequence1
else sequence2
```

to conserve vertical space. You should prefer this layout:

```
if (condition)
then
   sequence1
else
   sequence2
```

Pseudocode

Exercise

Write the pseudocode of searching an element (`ele`) in an unsorted array (`arr[size]`). Don't forget

- to include an algorithm header
- to use good variable names;
- the six elements of the algorithm header;
- the statement constructs.

Measuring Algorithm Efficiency I

- How do we measure running time?
- Possible solution: run algorithm for different inputs and record actual running time.

Example:

- Implement the algorithm of Selection Sort.
- Use a stopwatch to measure the program's execution time for n = 20, 40, 80, 120, 160, 200

N	Time (s)
20	0.31
40	1.15
80	4.28
120	9.28
160	16.06
200	24.67

Measuring Algorithm Efficiency II

- Can be useful, but disadvantages:
    - Can't test *all* inputs.
    - Running time is machine dependent.
    - Must implement the algorithm to study it.
    - Choice of programming language can affect algorithm speed.

Measuring Algorithm Efficiency

- We consider a method of analyzing algorithms from their pseudocode.
- Advantages of this method:
  - Takes into account all possible inputs.
  - Permits comparison of algorithms independently from hardware.
  - Can be applied to both pseudocode *or* actual code.

Measuring Algorithm Efficiency

Implementation-independent Method

- Express running time as a relationship between size of the input n, and the number of time units t needed to execute the algorithm on that input.
- If the number of time units required is five times the size of the input, then these two quantities are related by the equation

  $$t = 5n$$

  Typically, we will express t as a function of the size of the input:

  $$t = f(n) = 5n$$

Measuring Algorithm Efficiency

What is a "time unit"?

- We define a set of *primitive operations*.
- We assume each primitive operation takes the same amount of time.
- Now our problem is reduced to counting the number of primitive operations required by the algorithm as a function of input size.

Primitive Operations

- The following are all primitive operations:
  - Assigning a value to a variable
  - Calling another algorithm (function)
  - Performing an arithmetic operation (adding, etc)
  - Indexing into an array
  - Comparing two values (includes all logical and relational operators)
  - Following (dereferencing) an object reference (pointer)
  - Returning from a method (function, procedure)

Counting Primitive Operations

Example: arrayMax

Algorithm *arrayMax*(A, n)
pre: A :: **ArrayOfInteger**[n]
return: value of largest element of A

currentMax ← A[0]	2 ops
i ← 1	1 op
while (i < n) *do*	1 op
if (*currentMax* < A[i])	2 ops
then *currentMax* ← A[i]	2 ops
i ← i + 1	2 ops
done	
return currentMax	1 op

- Loop Body:
  - Single loop body iteration: *5 or 7 operations*
  - Loop body repeated $n-1$ times. Note: the loop condition will be tested 1 additional time, when $i = n$ and the loop condition fails.

  So: *between* $5(n-1)$ *and* $7(n-1)$ *operations*, when $n > 1$
- Best case: $t = 2 + 1 + 5(n-1) + 1 + 1 = 5n$
- Worst case: $t = 2 + 1 + 7(n-1) + 1 + 1 = 7n - 2$

Asymptotic Analysis

- Is there a difference in efficiency between an algorithm that needs $2n$ time units vs. one that needs $5n$ time units?
- Running times of algorithms are often more complex. For example:

$$f(n) = n^2 + 100n + \log_{10} n + 1000$$

- We really only need to consider the term that grows the fastest to characterize algorithms.
- This approximation is known as *asymptotic complexity* or *time complexity*.

Asymptotic Analysis
Growth of Terms in $f(n)$

$$f(n) = n^2 + 100n + \log_{10} n + 1000$$

n	$f(n)$	n^2		$100n$		$\log_{10} n$		1,000	
	Value	Value	%	Value	%	Value	%	Value	%
1	1,101	1	0.1	100	9.1	0	0.0	1,000	90.83
10	2,101	100	4.76	1,000	47.6	1	0.05	1,000	47.60
100	21,002	10,000	47.6	10,000	47.6	2	0.001	1,000	4.76
1,000	1,101,003	1,000,000	90.8	100,000	9.1	3	0.0003	1,000	0.09
10,000	101,001,004	100,000,000	99.0	1,000,000	0.99	4	0.0	1,000	0.001
100,000	10,010,001,005	10,000,000,000	99.9	10,000,000	0.099	5	0.0	1,000	0.00

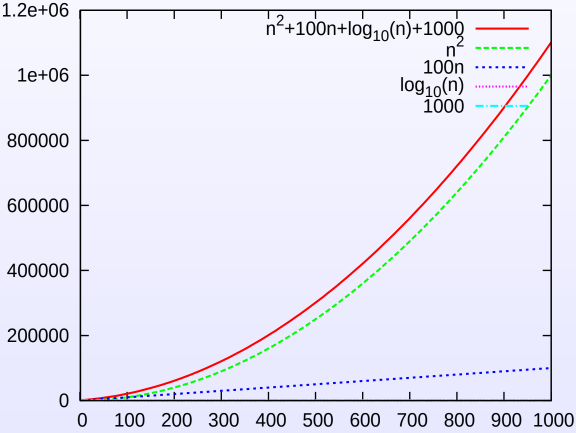
Source: A. Drozdek, Data Structures and Algorithms in Java, Thompson, 2005.

For sufficiently large n, only the n^2 term is significant.

Asymptotic Analysis

Growth of Terms in $f(n)$

$$f(n) = n^2 + 100n + \log_{10} n + 1000$$

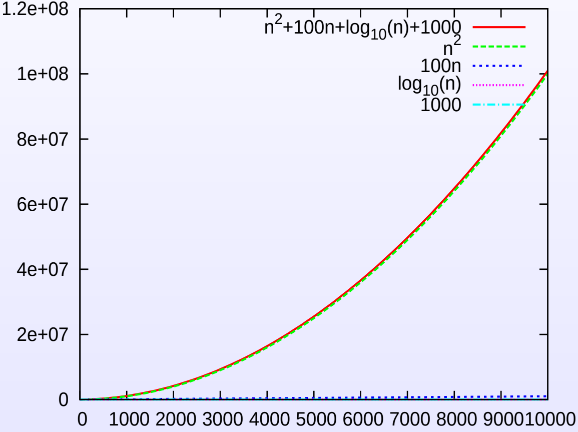


For sufficiently large n, only the n^2 term is significant.

Asymptotic Analysis

Growth of Terms in $f(n)$

$$f(n) = n^2 + 100n + \log_{10} n + 1000$$



For sufficiently large n, only the n^2 term is significant.

Asymptotic Approximation

Big-O Notation

- Do we need to know exactly how many primitive operations are performed?
- It is usually enough to know that the running time of some algorithm grows proportionally to n rather than, say, $5n + 9$.
- We formalize these notions by introducing *Big-O notation*.

Big-O Notation

- Let $f(n)$ and $g(n)$ be functions from non-negative integers into real numbers.
- We say "$f(n)$ is $O(g(n))$" if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \leq cg(n)$ for every integer $n \geq n_0$".
- What does **that** mean?
- It means that for inputs of size larger than n_0, the value of $cg(n)$ is always bigger than that of $f(n)$.
- In other words, $f(n)$ is $O(g(n))$ if and only if for large enough n, $f(n)$ grows no faster than $cg(n)$.

Big-O Notation

Example 1

- Claim: $7n - 2$ is $O(n)$.
  - Let $f(n) = 7n - 2$ and $g(n) = n$.
  - Choose $c = 7$ and $n_0 = 1$.
  - Now consider the relationship between $f(n)$ and $cg(n) = 7n$.
  - For any $n \geq n_0$ it is clear that $f(n) \leq cg(n)$ because $7n - 2 \leq 7n$.
  - Therefore, it must be true that $7n - 2$ is $O(n)$.

Big-O Notation

Example 2

- Claim: $f(n) = 20n^3 + 10n \log n + 5$ is $O(n^3)$.
  - let $g(n) = n^3$.
  - Choose $c = 35$ and $n_0 = 1$.
  - Claim: $f(n) \leq cg(n)$ for all $n \geq n_0$ (which would satisfy the definition of big-O notation). So let's check...
  - We must verify that

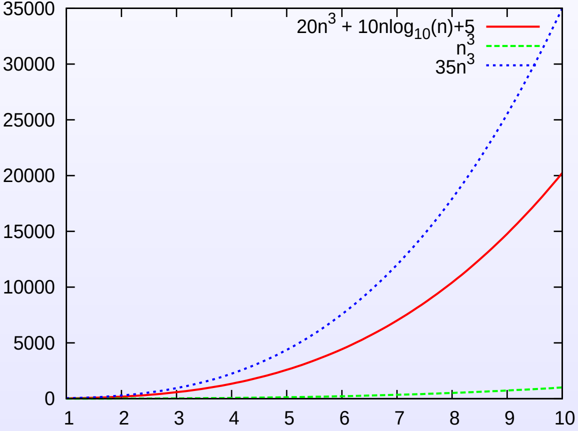
$$20n^3 + 10n \log n + 5 \leq 35n^3$$

is true for all $n \geq 1$.

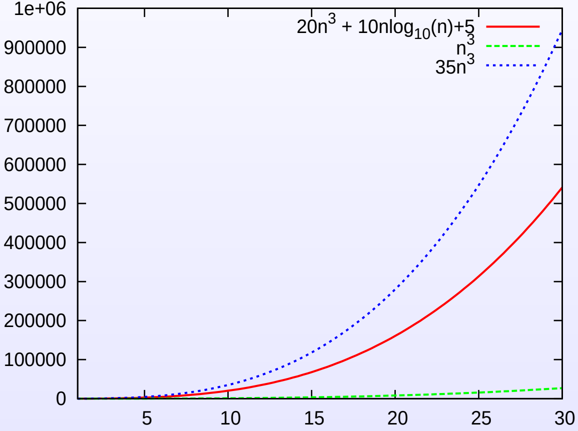
n	$f(n)$	$cg(n)$
1	$20 + 0 + 5 = 25$	$35 \cdot 1 = 35$
2	$20 \cdot 8 + 10 \cdot 2 \cdot \log 2 + 5 = 185$	$35 \cdot 8 = 280$
3	$20 \cdot 27 + 10 \cdot 3 \cdot \log 3 + 5 \approx 592.5$	$35 \cdot 27 = 945$
...	...	...

Big-O Notation

Example 2



Big-O Notation
Example 2



Big-O Notation

General Rule

- General rule: pick the term that grows the fastest and remove the constants from it:
  - $5n + 2\log n$ is $O(n)$.
  - $3n^3 + \frac{2^n}{6}$ is $O(2^n)$.
  - $8\log n + 7n$ is $O(n)$.

Common Growth Functions

- From most slowly growing, to most quickly growing, some common growth functions:
  - $\log \log n$
  - $\log n$ (logarithmic)
  - $\sqrt{n}$
  - n (linear)
  - $n \log n$
  - n^2 (quadratic)
  - n^3 (cubic)

  ...
  - n^k (polynomial hierarchy, k is a constant)
  - a^n (exponential hierarchy, a is a constant)
  - $n!$

Analysis of Algorithms

General Procedure

- General procedure for analyzing algorithms:
  - Count the maximum (worst case) number of primitive operations in terms of n, the input size.
  - Simplify the function that you get using Big-O notation.
  - Algorithms can then be compared according to their Big-O expressions.
- **We say that two algorithms are equally efficient if the Big-O expressions of their asymptotic complexity are the same.**

Analysis of Algorithms
Exercise

Sort the algorithms below from the slowest to the fastest growing:

- $f(n) = 5\log(n) + 1000\log\log(n)$
- $g(n) = \log(n) + 3n\log(n)$
- $h(n) = 5n^{\frac{1}{2}} + \frac{n^2}{5}$
- $i(n) = 100n + n! + n^{25}$
- $j(n) = \frac{3^n}{n} + n^3$

Analysis Examples

Linear Loops

- Simple counted loops are Linear Loops:

```
for i from 0 to n do
    <statements>
done
```

  Loop executes n times (n is size of input) As long as number of primitive operations in loop body is independent of n, such a loop is $O(n)$.

- This loop is also linear. Why?

```
for i from 0 to n by 2 do
    <statements>
do
```

Analysis Examples
Logarithmic Loops

- Logarithmic Loops result when the counter is multiplied or divided each iteration:

```
i ← 1
while i < n
    <loop body>
    i ← i * 2
done

i ← n
while i ≥ 1
    <loop body>
    i ← i / 2
done
```

Each of these loops executes $f(n) = c \log_2(n)$ times where c is the number of primitive operations in the loop body. Thus each loop is $O(\log n)$.

Analysis Examples

Nested Loops

- When loops are nested, the number of times the inner loop body executes is equal to:

  $$\text{outer loop iterations} \times \text{inner loop iterations}$$

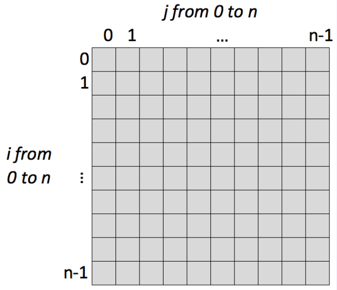
  We examine two kinds of nested loops: quadratic, and dependent quadratic.

Analysis Examples I

Quadratic Nested Loops

- Quadratic loops occur when the inner and outer loops each execute n times:

```
for i from 0 to n do
    for j from 0 to n do
        <loop body containing c primitive operations>
    done
done
```



Analysis Examples II

Quadratic Nested Loops

- Thus, total number of primitive operations is $f(n) = c \times n \times n$ which is $O(n^2)$.
- Question:
  What if the inner loop was **for** *j* **from** 0 **to** m **do**

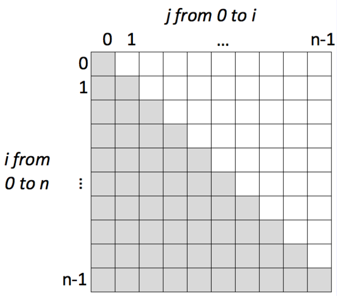
```
for i from 0 to n do
    for j from 0 to m do
        <loop body containing c primitive operations>
    done
done
```

Analysis Examples I

Dependent Quadratic Nested Loops

- Dependent quadratic loops result when the number of iterations in the inner loop depends on the value of the outer loop counter:

```
for i from 0 to n do
    for j from 0 to i+1 do
        <loop body containing c primitive operations>
    done
done
```



Analysis Examples II
Dependent Quadratic Nested Loops

Number of operations:

$$
\begin{aligned}
c + 2c + 3c + 4c + \cdots + nc &= c \cdot (1 + 2 + 3 + \ldots + n) \\
&= c \cdot \sum_{i=1}^{n} i \\
&= c \cdot \frac{(n+1)n}{2} \\
&= \frac{c}{2}(n^2 + n) = O(n^2)
\end{aligned}
$$

Analyzing Complete Algorithms

Exercise: Matrix Sum

Algorithm *addMatrix*(*matrix1*, *matrix2*, *matrix3*)
add two 2D arrays, putting it into the third
pre: a fixed size *nxn* is known
    matrix1, *matrix2* :: **ArrayOfInteger**[*n*][*n*]
post: *matrix3* :: **shared ArrayOfInteger**[*n*][n]
    will contain the sum of the two matricies

for *i* **from** 0 **to** *n* **do**
    for *j* **from** 0 **to** *n* **do**
        $matrix3[i][j] \leftarrow matrix1[i][j] + matrix2[i][j]$
    done
done

What is the time complexity in the worst case? Best case?

Analyzing Complete Algorithms

Exercise: Prefix Averages

Algorithm *prefixAverages*(X)
compute prefix averages
pre: X :: **ArrayOfInteger**[n]
return: A :: **ArrayOfReal**[n]
    where $A[i]$ is the average of $X[0] \ldots X[i]$

for i **from** 0 **to** n **do**
    sum ← 0.0
    for j **from** 0 **to** *i+1* **do**
        sum ← *sum* + $X[j]$
    done
    $A[i]$ ← *sum* / (i + 1)
done
return A

What is the time complexity in the worst case? Best case?

Analyzing Complete Algorithms

Exercise: Binary Search

Algorithm *binarySearch*(*arr*, *key*)
Search array for a specific item.
Return index of it's location.
pre: *arr* :: **ArrayOfInteger**[*n*] -- sorted in ascending order
key :: **Integer** to search for in *arr*
post: *arr* is unchanged
return: Integer index of *key* in *arr*, -1 if not found

```
lo ← 0
hi ← n-1
while (lo ≤ hi )
    mid ← (lo + hi) / 2
    if (key < arr[mid])
        hi ← mid - 1
    else if (key > arr[mid])
        lo ← mid + 1
    else
        return mid;
done
return -1;
```

What is the time complexity in the worst case? Best case?

Hard Problems

How hard is hard?

- Algorithms with polynomial or better time complexity are considered *tractable*.
- Algorithms with exponential (or worse) complexity are considered *intractable*.
- Problems with very simple descriptions can be intractable!

The Traveling Salesman Problem

The Traveling Salesman Problem

Given a set of cities, the cost of travel between each pair of cities, and a starting point, what is the cheapest way of visiting all of the cities exactly once and returning to the starting point?

- Obvious solution: Try every ordering of cities. Find the one with minimum cost.
- Worst case time complexity: ??
- Other solutions?

The Knapsack Problem

The Knapsack Problem

Given a set of n items, each of which has a value and a weight, what is the most valuable subset of items whose weight does not exceed some threshold W?

- In other words, we have a backpack of finite capacity, and we want to pack into it the most valuable set of items that can fit.
- Obvious solution: Try every subset of items whose weight does not exceed W. Pick the most valuable.
- Worst case time complexity: ??
- Other solutions?

Intractable Problems

Are there better solutions?

- We haven't been able to find better solutions, but we also can't prove they don't exist!
- There are two possibilities:
  - There really aren't polynomial time solutions.
  - We are just dumb (we haven't found the fast solutions yet).

Worse than Intractable

Undecidable Problems

- There are problems that are worse than intractable.
- There are problems for which we can prove there is no algorithm *at all* that solves them.
- We may be able to solve these problems for some inputs, but there is no one algorithm that will solve these problems for *all inputs*.
- There are problems of this kind that also have very simple descriptions.

Post's Correspondence Problem

An Undecidable Problem

Post's Correspondence Problem

Given a set of n cards, each with an upper and lower sequence of symbols, is there a sequence cards (of any length) such that the top symbols read the same as the bottom symbols?

Note: We can use as many copies of a card as we want.

Input:

aba	bbb	aab	bb
a	aaa	abab	babba

a solution:

aba	bb	aab	aba
a	babba	abab	a

Post's Correspondence Problem

An Undecidable Problem

- Some interesting observations:
  - If there is a solution, there is an algorithm to find it (what is it?).
  - If there is no solution, the same algorithm runs forever.
  - If we remove the rightmost card in the previous example, there is no solution for any k.

Hard Problems

- How many problems are tractable?
- How many problems are intractable?
- How many problems are undecidable?