**This course is really about Computational Problem Solving.**

It is the process of coming up with a correct AND efficient computational solution (i.e., one or more algorithms) to a given problem.

The various steps of computational problem solving, and the mathematical tools and design techniques you can use for this, are what we will discuss next.

# Computational Problem Solving

- Problem specification
- Designing solution **strategies**
- Developing corresponding **algorithms**
  - Understand existing algorithms and modify/reuse, or
  - Design new algorithms
- Understanding an algorithm by simulating its operation on an input
- Ensuring/proving correctness
- Analyzing and comparing performance/efficiency
  - Theoretically: Using a variety of mathematical tools
  - Empirically: Code, run and collect performance data
- Choosing the best algorithm to implement

# Solving problems computationally

- What do you do first?
- Write code?  Bad idea!
- Specify the problem
- Come up with multiple strategies
- Design corresponding algorithms (or understand, modify and reuse existing ones)
- Then choose the best
- How do you choose?
  - Correctness
  - Efficiency

# Specifying the problem

Restate the problem in such a way that you begin to get an idea of the **complexity** of the problem, and a **computational solution** to it (i.e. a strategy to solve it, leading to an algorithm) can be developed.

E.g., online dating → matching problem

# Another Example


How will you specify the problem?

# Simplified Boggle

Given a nxn array of characters; find all English words horizontally and vertically in any direction.

Assume that you have a function Dict_lookup(s: string) available that returns True if the string s is an English word, else it returns False.

Now, we can start to think about how **complex** is this “simplified” Boggle is and computational strategies to play the game.

# Complexity

How many strings must be tested?

Approximately $\mathbf{n^3}$ strings!

Precisely, $\mathbf{n^3}$-$\mathbf{n^2}$ strings…

**Why?**

# Designing strategies

**Strategy # 1**

1. Construct all possible substrings
2. Check each

- Is this correct? What does it mean for this strategy to be correct?

# Strategy # 1: Algorithm # 1

How do you construct all possible substrings?

- You can take one row at a time and construct all of its substrings
- But it has to be done twice (why?)
- Repeat for each column

- Divide the problem into subproblems
- Solve each subproblem the same way
- Then construct the overall solution from the subproblem solutions

- Solving the problem for a 4X4 matrix
  - Solving the problem 8 times, once for each row and column
    - Solving the problem twice for each row/column (once in the forward and once in the backward direction)
- So, once you figure out how to solve the problem for a row/column in one direction, you are done!

# Notes about Notation

- A[p…p] is an array with one cell with index p
- A[q…p] where q>p is an empty array
- A[p…q] where q≥p is an array with at least one element
- Two kinds of for loops: “for i←p to q” is an incrementing loop and “for i←q downto p” is a decrementing loop
- for i←q to p where q>p and for i←p downto q where q>p will both fail immediately without executing.

# First Step

how to solve the problem for a row in one direction?

```
i ← 1
   For j ← 1 to 4
        For k ← j to 4
             temp←make-string(A[i,j]…A[i,k])
             if dict_lookup(temp)=T then print temp
```

# Next Step

Solving the problem twice for a row (once in the forward and once in the backward direction)

```
i ← 1
    For j ← 1 to 4
      For k ← j to 4
            temp ← make-string(A[i,j]…A[i,k])
            if dict_lookup(temp)=T then print temp
            temp ← string-reverse(temp)
             if dict_lookup(temp)=T then print temp
```

# Next Step

- Is this piece of algorithm correct?
- Is this piece of algorithm efficient?

Issues:

1. Is it more efficient to avoid duplicate dictionary lookups for single character strings?
2. Is it more efficient to call make-string again instead of string-reverse?

# Making the algorithm piece more efficient

For a 4-character row a single character needs to be looked up once, but will be looked up twice: 4 extra calls to dict_lookup

For a 4X4 matrix a single character needs to be looked up once, but will be looked up 4-times: 48 extra calls to dict_lookup

For a nXn matrix a single character needs to be looked up once, but will be looked up 4-times: $3n^2$ extra calls to dict_lookup

looking up a dictionary is expensive and should be minimized!

# Making the algorithm piece more efficient

Solving the problem twice for the first row (once in the forward and once in the backward direction) but checking one-character strings only once:

```
i ← 1
    For j ← 1 to 4
        temp ← make-string(A[i,j]…A[i,j])
        if dict_lookup(temp)=T then print temp
        For k ← (j+1) to 4
                temp ← make-string(A[i,j]…A[i,k])
                if dict_lookup(temp)=T then print temp
                temp ← string-reverse(temp)
                if dict_lookup(temp)=T then print temp
```

# Next Step

Generalizing: solving the problem twice for all 4 rows (once in the forward and once in the backward direction)

```
For i ← 1 to 4
        For j ← 1 to 4
            temp ← make-string(A[i,j]…A[i,j])
            if dict_lookup(temp)=T then print temp
            For k ← (j+1) to 4
                        temp ← make-string(A[i,j]…A[i,k])
                        if dict_lookup(temp)=T then print temp
                        temp ← string-reverse(temp)
                        if dict_lookup(temp)=T then print temp
```

# Next Step

Generalizing: reusing the same loops a second time to solve the problem twice for each column (upward and downward) as well as for each row, i.e. for the full matrix:

```
For i ← 1 to 4
        For j ← 1 to 4
            temp ← make-string(A[i,j]…A[i,j])
            if dict_lookup(temp)=T then print temp
            For k ← (j+1) to 4
                        temp ← make-string(A[i,j]…A[i,k])
                        if dict_lookup(temp)=T then print temp
                        temp ← string-reverse(temp)
                        if dict_lookup(temp)=T then print temp
For j ← 1 to 4
      For i ← 1 to 4
            temp ← make-string(A[i,j])
            if dict_lookup(temp)=T then print temp
            For k ←(i+1) to 4
                        temp ← make-string(A[i,j]…A[k,j])
                        if dict_lookup(temp)=T then print temp
                        temp ← string-reverse(temp)
                        if dict_lookup(temp)=T then print temp
```

# Making it more efficient

We can make this more efficient by noticing that since all single character strings would have been considered in the first set of nested loops, they don't have to be considered again in the second set of loops.

```
For i ← 1 to 4
        For j ← 1 to 4
            temp ← make-string(A[i,j]…A[i,j])
            if dict_lookup(temp)=T then print temp
            For k ← (j+1) to 4
                        temp ← make-string(A[i,j]…A[i,k])
                        if dict_lookup(temp)=T then print temp
                        temp ← string-reverse(temp)
                        if dict_lookup(temp)=T then print temp
For j ← 1 to 4
     For i ← 1 to 4
            For k ←(i+1) to 4
                        temp ← make-string(A[i,j]…A[k,j])
                        if dict_lookup(temp)=T then print temp
                        temp ← string-reverse(temp)
                        if dict_lookup(temp)=T then print temp
```

# Last Step

Generalizing: solving the problem for any nXn matrix:

```
For i ← 1 to n
        For j ← 1 to n
            temp ← make-string(A[i,j]…A[i,j])
            if dict_lookup(temp)=T then print temp
            For k ← (j+1 to n
                        temp ← make-string(A[i,j]…A[i,k])
                        if dict_lookup(temp)=T then print temp
                        temp ← string-reverse(temp)
                        if dict_lookup(temp)=T then print temp
For j ← 1 to n
     For i ← 1 to n
            For k ←(i+1) to n
                        temp ← make-string(A[i,j]…A[k,j])
                        if dict_lookup(temp)=T then print temp
                        temp ← string-reverse(temp)
                        if dict_lookup(temp)=T then print temp
```

# Algorithm

```
Boggle(A: n X n array of characters)
1 For i ← 1 to n
2       For j ← 1 to n
3           temp ← make-string(A[i,j]…A[i,j])
4           if dict_lookup(temp)=T then print temp
5           For k ← (j+1) to n
6                       temp ← make-string(A[i,j]…A[i,k])
7                       if dict_lookup(temp)=T then print temp
8                       temp ← string-reverse(temp)
9                       if dict_lookup(temp)=T then print temp
10 For j ← 1 to n
11      For i ← 1 to n
12          For k ←(i+1) to n
13                      temp ← make-string(A[i,j]…A[k,j]
14                      if dict_lookup(temp)=T then print temp
15                      temp ← string-reverse(temp)
16                      if dict_lookup(temp)=T then print temp
```

# Questions

We discussed why the strategy {construct all possible substrings and check each against the dictionary} is correct. Does it mean that the algorithm is correct as well?

How efficient is the algorithm?

Are there other algorithms that implement the same strategy?

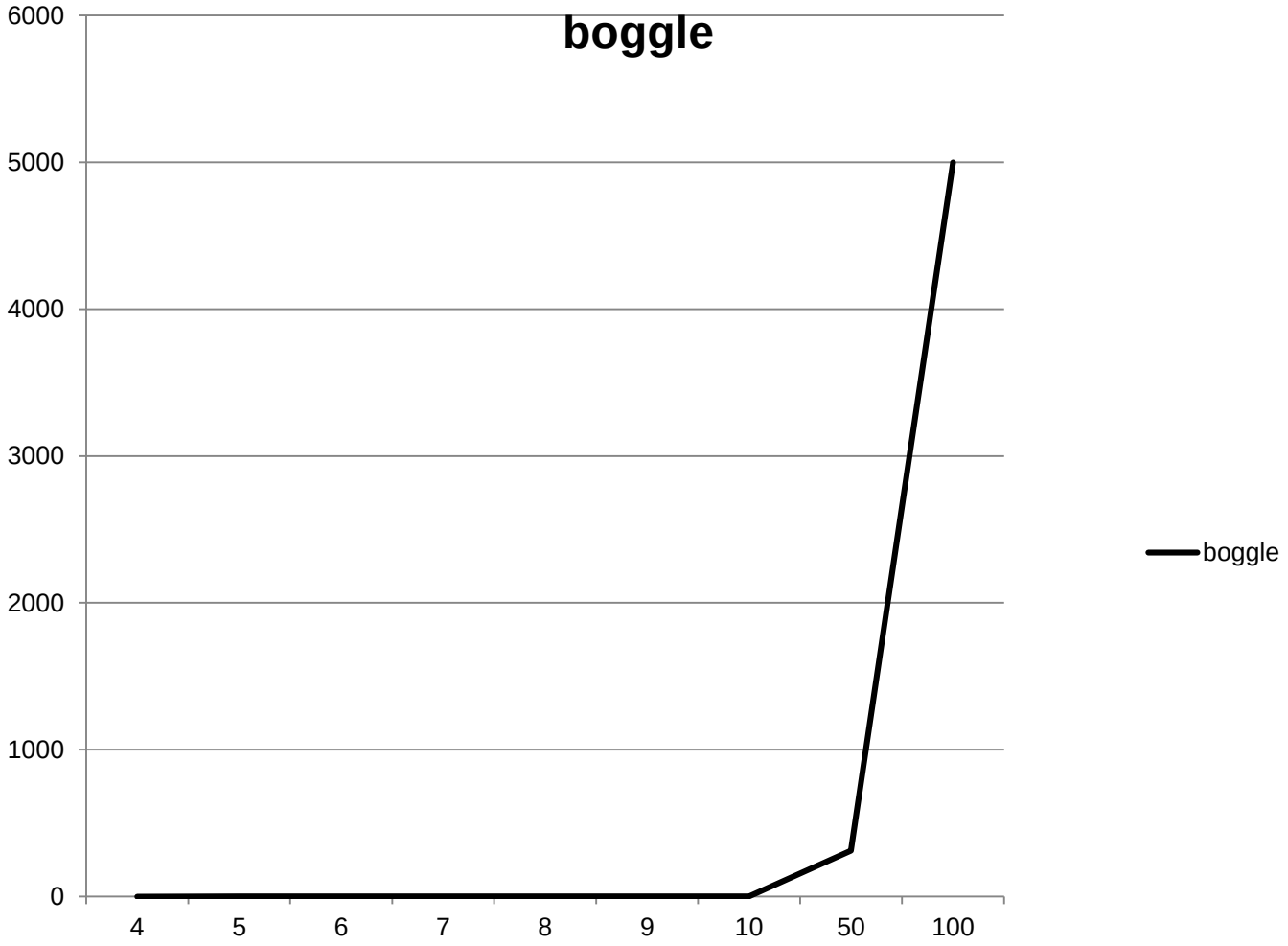
Could those be more efficient?

# How efficient this algorithm?

- How much does make-string cost?
  - O(n): why?
- How much does string-reverse cost?
  - O(n): why?
- How much does dict_lookup cost?
  - O(n); why?
- What is the total cost?
  - approximately (and at most) $8n^4+2n^2$
  - why? we will work this out in class
  - So this algorithm is $O(n^4)$
  - Using same assumptions as when we discussed the selection problem, how long will it take to solve a 4X4 Boggle board?

# What is the performance of this algorithm?

| board size | time |
|---|---|
| 4 | 0.01 ms |
| 5 | 0.03 ms |
| 6 | 0.06 ms |
| 7 | 0.12 ms |
| 8 | 0.20 ms |
| 9 | 0.33ms |
| 10 | 0.5 ms |
| 50 | 300 ms |
| 100 | 5000 ms |



# Strategy #1: Algorithm # 2

- Look at each cell
  - Check if that cell makes a one-letter word
  - Construct and check strings (and their reverses) made up of the character in that cell and the ones above it until array boundary is reached
  - Construct and check strings (and their reverses) made up of the character in that cell and the ones to the left of it until array boundary is reached
  - Construct and check strings (and their reverses) made up of the character in that cell and the ones below it until array boundary is reached
  - Construct and check strings (and their reverses) made up of the character in that cell and the ones to the right of it until array boundary is reached
- What is the corresponding algorithm?

# Corresponding Algorithm

```
Boggle(A: n X n array of characters)
For i=1 to n
    For j=1 to n
        temp ← make-string(A[i,j]…A[i,j])
        if dict_lookup(temp)=T then print temp
        For k=(i-1) to 1
            temp ← make-string(A[i,j]…A[k,j])
            if dict_lookup(temp)=T then print temp
            temp ← string-reverse(temp)
            if dict_lookup(temp)=T then print temp
        For k=(j-1) to 1
            temp ← make-string(A[i,j]…A[i,k])
            if dict_lookup(temp)=T then print temp
            temp ← string-reverse(temp)
            if dict_lookup(temp)=T then print temp
        For k=(i+1) to n
            temp ← make-string(A[i,j]…A[k,j])
            if dict_lookup(temp)=T then print temp
            temp ← string-reverse(temp)
            if dict_lookup(temp)=T then print temp
        For k=(j+1) to n
            temp ← make-string(A[i,j]…A[i,k])
            if dict_lookup(temp)=T then print temp
            temp ← string-reverse(temp)
            if dict_lookup(temp)=T then print temp
```

**Thinking Assignment**

Make sure you understand how/why this algorithm implements the same strategy but with a different approach

# Same Algorithm Made More Efficient

```
Boggle(A: n X n array of characters)
1 For i=1 to n
2    For j=1 to n
3          temp ← make-string(A[i,j]…A[i,j])
4          if dict_lookup(temp)=T then print temp
5          For k=(i+1) to n
6                temp ← make-string(A[i,j]…A[k,j])
7                if dict_lookup(temp)=T then print temp
8                temp ← string-reverse(temp)
9                if dict_lookup(temp)=T then print temp
10         For k=(j+1) to n
11               temp ← make-string(A[i,j]…A[i,k])
12               if dict_lookup(temp)=T then print temp
13               temp ← string-reverse(temp)
14               if dict_lookup(temp)=T then print temp
```

**Thinking Assignment**
Make sure you understand how we get this algorithm from the one on the previous slide and why this is more efficient

**Thinking Assignment** Compute the efficiency of this algorithm

# Thinking Assignment

- Is this algorithm correct?
- Based on what you haver learned so far, are you able to determine which of these two algorithms implementing the same strategy is more efficient?
- Are there other strategies/algorithms?
- Could those be more efficient?

# Thinking Assignments

- **Strategy # 2**: So far we've created strings from the board and looked these up in the dictionary. Instead, what if you go in the other direction: get each **full** word from the dictionary and see if it is on the board?

- Develop this strategy further and write down the corresponding algorithm on paper. Assume the dictionary contains m words of length at most n characters and **the only function you have** for accessing it is get_next_word that will fetch the next word (or return nil when all words have been fetched).

- Which strategy/algorithm is most efficient? Why?

- Can strategy 2 be made more efficient if the dictionary is organized in a particular way and/or provided other functions? Which functions? How much more efficient will this strategy become?