Algorithm Design

Debswapna Bhattacharya, PhD
Assistant Professor
Computer Science and Software Engineering
Auburn University

Steps of Computational Problem Solving

Part I: Algorithm Design

1. Specify the problem well
2. Come up with at least one solution strategy
3. Develop corresponding algorithms
    1. Find and understand existing algorithms (by simulating their operation) that use these strategies to solve the problem
    2. If no such algorithms can be found, find and understand existing algorithms that solve a similar problem, then modify them appropriately to solve your problem
    3. Design new algorithms

Part II: Algorithm Analysis

4. Ensure/prove correctness of all algorithms
5. Analyze and compare their performance/efficiency
    4. Theoretically: Using a variety of mathematical tools
    5. Empirically: Code, run and collect performance data
6. Choose the most efficient algorithm to implement

Recursive Algorithm Design

- We have discussed several examples in class
- Generally lead to compact, elegant algorithms
- Often (but not always) produce efficient algorithms
  - With the exception of
    - tail recursion
    - duplicated work
- Starting point-come up with a **recursively stated computational strategy**. E.g.,
  - The maximum number in an array is the larger of the first/last number and the maximum number in the rest of the array
  - An array can be sorted by sorting its left half and the right half, and then merging the two sorted halves
  - $F_n=F_{n-1}+F_{n-2}$

Recursive Divide & Conquer Algorithm Design

- A **special case** of recursive algorithm design
- **Divide** the input into two or more **non-overlapping** pieces
- Solve the problem **recursively** for each of these partial inputs
- Then **combine** these partial solutions into an overall solution (conquer)
- Starting point-come up with a recursively stated computational strategy that makes the **divide and conquer aspects explicit. E.g.,**
  - The largest number in an array is the larger of the largest number in the left and right halves of the array
  - An array can be sorted by sorting its left half and the right half, and then merging the two sorted halves
  - The number of times a character appears in a string can be counted by splitting the string into two substrings, counting that character in each substring, then adding the count

Iterative Algorithm Design

- We have discussed several examples in class
- Solve a problem by **repeatedly** solving a **partial problem** by means of a loop
- Generally produce efficient algorithms with complexities of $O(n)$, $O(n^2)$, $O(n^3)$ etc. (depending on how many nested loops there are in the algorithm)
- Starting point-come up with an **iterative** computational strategy. E.g.,
  - The largest number in an array can be found by iterating through the array and comparing each number with a "current largest" and updating it
  - An array can be sorted by iterating through the array picking out each number, and inserting it into the correct sorted place among all numbers to its left
  - All instances of a specific number x in an array can be moved to its left end by iterating through the array from the left and the right simultaneously, and swapping each occurrence of a number not equal to x found from the left with an occurrence of x found on the right.

Iterative Divide & Conquer Algorithm Design

- A **special case** of iterative algorithm design
- **Divide** the input into two or more pieces
- Solve the problem **iteratively** for each of these partial inputs
- Then **combine** these partial solutions into an overall solution (conquer)
- Starting point-come up with an iteratively stated computational strategy that makes the **divide and conquer aspects** and **iteration explicit.** E.g.,
  - The largest number in an array can be found by iterating through the left half of the array and comparing each number with a "current largest" and updating it to find the largest in the left half, doing the same for the right half, and then selecting the larger of these two numbers
  - An array can be sorted by iterating through its left half, picking out each number, and inserting it into the correct sorted place among all numbers to its left, similarly iteratively sorting its right half, and then merging the two sorted halves

All these design techniques illustrate the notion of **problem decomposition**

Breaking a problem into subproblems and solving those subproblems in order to solve the overall problem

Iterative Divide & Conquer Algorithm Design

Thinking Assignment: Come up with a well-defined problem specification

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 the **inherent complexity** of this "simplified" Boggle, and computational strategies to play the game.

Inherent Complexity

How many strings must be tested from a nXn board?

approximately $\mathbf{n^3}$ strings

Inherent problem complexity is $\mathbf{\Omega(n^3)}$

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? I.e., what are the correctness criteria for this problem?
- How do you construct all possible substrings?

Strategy # 1: Elaboration

1. Construct all possible substrings
    1. Take one row and construct all of its substrings going left to right
    2. Then repeat in the other direction
    3. Repeat for each of the other rows
2. Repeat for each column

Iterative Divide & Conquer

- Divide the problem into subproblems
- Solve each subproblem iteratively 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!

First Step

how to solve the problem using iteration for the first 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 the first 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 algorithm fragment correct?
- Is this algorithm fragment 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 fragment 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 fragment 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? How does algorithm efficiency compare with inherent problem complexity?

Are there other algorithms that implement the same strategy?

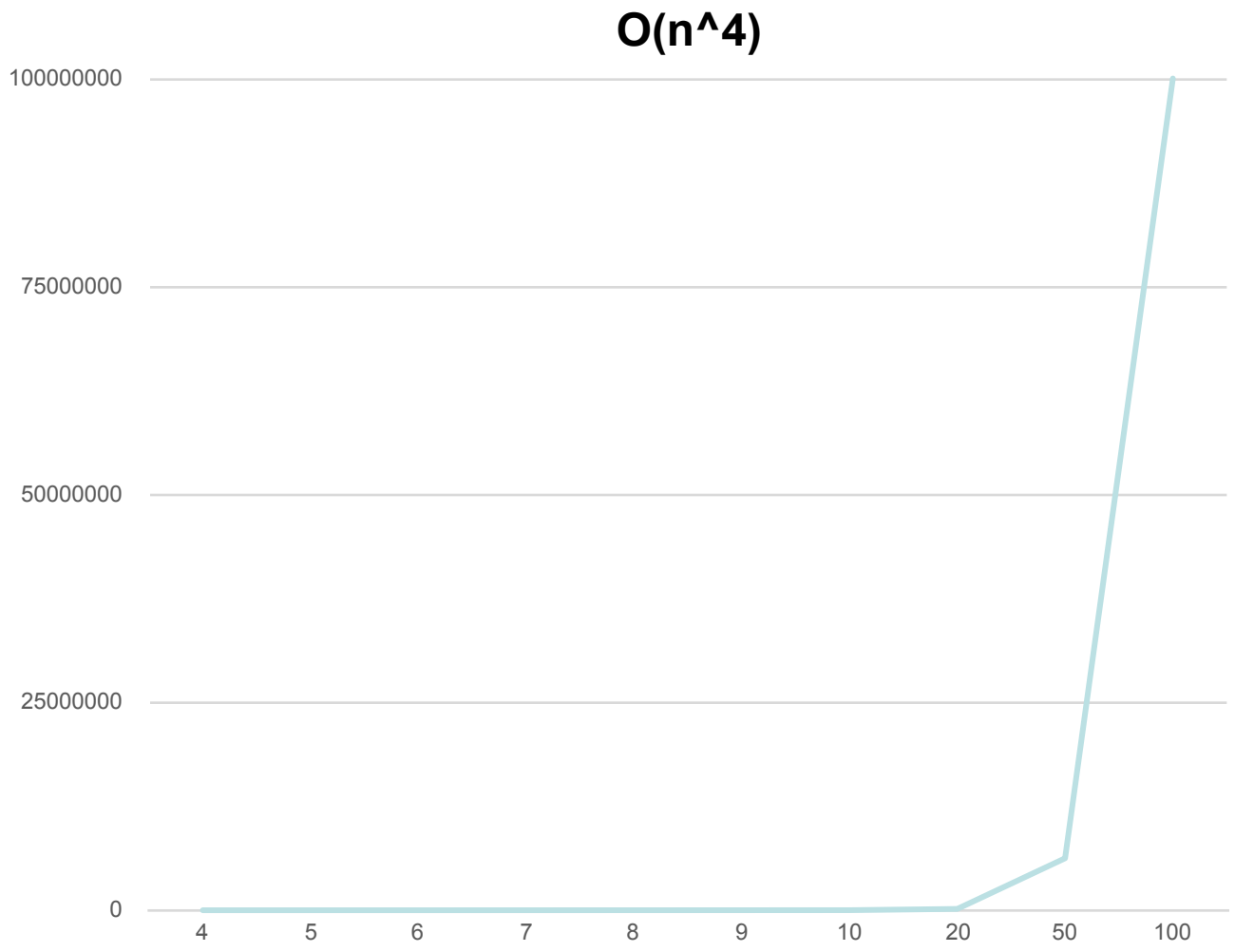
Could those be more efficient?

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?
  - **$O(n^4)$**
  - why? we will work this out in class
  - Using same timing assumptions as when we discussed the selection problem, how long will it take to solve a 4X4 Boggle board?

The performance of this algorithm

board size	time
10	4 millisecs
100	40 secs
1000	5 days



Same Strategy: Different Algorithm

- 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

Thinking Assignment

1. Write the corresponding algorithm
2. Compare it with the algorithm on the next slide

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

Previous 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: Estimate the efficiency of this algorithm as we did for the earlier one

Thinking Assignment

- Is this algorithm correct?
- Based on what you have 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 Assignment

- **Another Strategy**: 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 from the dictionary (or return nil when all words have been fetched), which has a complexity of O(n).

- Is it possible to determine which strategy/algorithm is most efficient? if so, which one? If not, what information do you need?

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

Recap

- Well-defined problem specification
- Inherent complexity of a computational problem
- Computational strategy development
  - Make sure your strategy is **correct** and **efficient**
- Translating strategies into to algorithms
  - Do not introduce logical errors during translation
  - Make sure your algorithm is **correct** and **efficient**

Thinking Assignment
Algorithm Design Exercise

http://codeforces.com/contest/611/problem/D

Steps of Computational Problem Solving

We are finished with: Algorithm Design

1. Specify the problem well
2. Come up with at least one solution strategy
3. Develop corresponding algorithms
    1. Find and understand existing algorithms (by simulating their operation) that use these strategies to solve the problem
    2. If no such algorithms can be found, find and understand existing algorithms that solve a similar problem, then modify them appropriately to solve your problem
    3. Design new algorithms

Next: Algorithm Analysis

4. Ensure/prove correctness of all algorithms
5. Analyze and compare their performance/efficiency
    4. Theoretically: Using a variety of mathematical tools
    5. Empirically: Code, run and collect performance data
6. Choose the most efficient algorithm to implement