Reading Assignments

By now you should have already read Chapter 1

Now read Chapter 2 Section 2.1

Computational Problem Solving

- Problems
- Designing solution strategies
- Developing algorithms
  - Writing algorithms that implement the strategies
  - Understand existing algorithms and modify/reuse
- 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

Correctness Proofs

If we think that an algorithm is correct/incorrect, how can we convincingly show that?

- Correctness proofs demonstrate that an algorithm is correct (or incorrect)
- Proof by Counterexample
- Proof by Contradiction
- Proof by Loop Invariants
- Proof by Induction

K-th Largest Number Problem

```
Algorithm A1(A: Array [1..n] of
distinct numbers, k:integer,
1<=k<=n)
        print A[k]
```

Is this algorithm correct?

Why or why not?

How will you justify your answer?

Proof by Counterexample

- Suppose n=4 and and k=1
- Suppose A is [1000, 2000, 4000, 3000]
- What does the algorithm output?
  - 1000
- What is the correct answer?
  - 4000
- The algorithm is incorrect!

Proof by Counterexample

- Show one valid input for which the algorithm produces incorrect output
- Can be used to show that an algorithm is incorrect
- Cannot be used to show an algorithm is correct! (why?)

Another Algorithm

Algorithm A2(A: Array [1..n] of distinct numbers, k:integer, 1<=k<=n)

    Sort A in ascending order
    print A[k]

Is this algorithm correct?

Why or why not?

How will you justify your answer?

Proof by Contradiction (A2)

1. Suppose the algorithm is correct.
2. That means that for all valid input arrays of n distinct numbers and all integers k, 1<=k<=n, it must print the k-th largest number.
3. Algorithm A2 sorts the array A in the increasing order.
4. So after sorting, A[n-k+1] will contain the k-th largest number.
5. Therefore by our assumption A2 must print out A[n-k+1].

Proof by Contradiction (A2)

4. A2 actually prints A[k].
5. A[n-k+1] = A[k] only when n-k+1 = k or 2k = n+1. I.e. only when n is odd and k = (n+1)/2. For all other values of k, A[n-k+1] ≠ A[k].
6. This means that for all valid n-sized arrays and all integers k, 1<=k<=n, it will not print the k-th largest number.
7. Step 6 of the proof contradicts step 2!
8. So our assumption in step 1 that the algorithm is correct must be wrong, i.e. the algorithm is incorrect.

Proof by Contradiction

- Start by assuming the opposite of what you want to prove;
- develop the logical and factual implications of this assumption;
- compare with what the algorithm specification says about its operation;
- show that this leads to a contradiction.
- Can be used to show that an algorithm is correct or incorrect

A Third Algorithm

Algorithm A3(A: Array [1..n] of distinct numbers, k:integer, 1<=k<=n)

Sort A in descending order
print A[k]

Is this algorithm correct?

Why or why not?

How will you justify your answer?

Proof by Contradiction (A3)

(Read and understand this yourself)

1. Suppose the algorithm is incorrect.
2. That means that for at least one valid input array of n distinct numbers and integer k, 1<=k<=n, it will produce an incorrect answer.
3. In other words, for at least one valid input array and integer k where 1<=k<=n, A3 will not print the k-th largest number.

Proof by Contradiction (A3)

4. Algorithm A3 sorts the array A in the decreasing order of incomes.
5. So after sorting, A[1..k] must contain the k largest numbers in A for any k where 1<=k<=n.
6. So for any n and 1<=k<=n, A[k] must contain the k-th largest number after A3 finishes sorting. Then it prints A[k].
7. So for all valid inputs A and k, A3 will print the k-th largest number.

Proof by Contradiction (A3)

8. Step 7 of the proof contradicts step 3!
9. So our assumption in step 1 must be wrong, i.e. the algorithm has to be correct.

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

Correctness Proof by Contradiction

What does it mean for this algorithm to be correct?

It must halt.

It must print out all English words that appear in a row or column in either direction.

It must not print out any non-English word.

- **Thinking Assignment**: Prove the correctness of this algorithm using the proof by contradiction strategy before looking up the answer in the following slides

Boggle Correctness Proof

A. Assume the algorithm is incorrect.

B. All its loops have fixed bounds so it will halt.

C. We assume dict_lookup works correctly and the algorithm will print only those words that the lookup confirms, so no non-English words will be printed.

D. So the implication of our assumption is that there must be at least one English word that the algorithm fails to print.

E. This implies that there must be at least one string that the algorithm does not check against the dictionary among the $(2n^3 – n^2)$ possible strings row-wise and columnwise in an nXn character array

F. There are two cases: (1) the string is in a row (2) it is in a column

G. Consider (1). Let this string be in row a: A[a,b]…A[a,c] where $1 \leq a,b,c \leq n$.

H. The part of the algorithm checking row-wise is:

```
1.For i=1…n
2.  For j=1…n
3.          temp ← make-string(A[i,j])
4.          if dict_lookup(temp)=T then print temp
5.          For k=(j+1)…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
```

I. Lines 1, 2 and 3 ensure that all 1-character strings A[i,j]…A[i,k] for $1 \le i,j \le n$, k=j are checked against the dictionary in line 4

J. Lines 1, 2, 5 and 6 ensure that all strings A[i,j]…A[i,k] are checked against the dictionary in line 7 for $1 \le i,j \le n$, $j<k \le n$

K. Therefore lines 1-7 of the algorithm check all possible strings A[i,j]…A[i,k] for $1 \le i,j \le n$ and $j \le k \le n$

L. So the string A[a,b]…A[a,c] that is not considered must have c<b

M. But if c<b then string A[a,b]…A[a,c] is the reverse of string A[a,c]…A[a,b] where $1 \le a,c \le n$ and $c<b \le n$, so by statement K line 6 will consider string A[a,c]…A[a,b]

N. But then lines 8 & 9 will consider string A[a,b]…A[a,c]

O. So there can be **no string** string A[a,b]…A[a,c] in any row that is not checked

Proof

P. An argument similar to steps G-O can be used to show that there can be no unconsidered string A[a,b]…A[c,b] in a column either.

Q. O & P together contradict E.

R. Therefore the assumption we started with must be incorrect. I.e., the algorithm is correct!

Proof by Loop Invariants

- A Loop Invariant (LI) is a property that holds true before any execution of the loop.
- Usually this is a property of the data that the loop manipulates.
- Usually the property is stated in terms of the i^{th} execution of the loop.

Proof by Loop Invariants

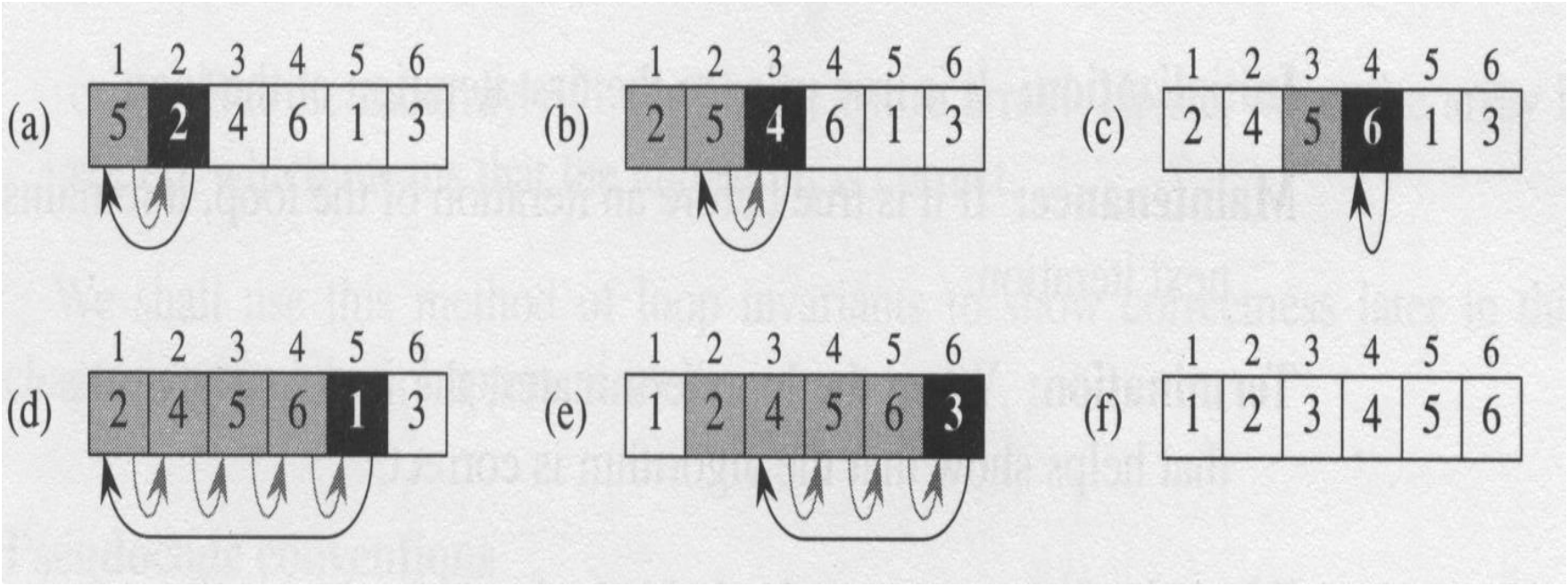
- For algorithms that do their work by using a main loop, stating and proving a LI that relates to what the algorithm is intended to do is a way of proving the algorithm to be correct.

Example: Proof by LI

How does Insertion Sort work?

```
Insertion-sort(A)
1 for j ←2 to n
2   do key←A[j]
3      ∗Insert A[j] into the sorted sequence A[1..j-1]
4      i ← j - 1
5      while i>0 and A[i]>key
6           do A[i+1] ←A[i]
7           i ← i - 1
8      A[i +1] ←key
```

Understanding an Algorithm by Simulating Its Operation on an Input



An Interesting Property of Insertion Sort

- *Sorted in place* :
  - The numbers are rearranged within the array A, without using an additional data structure (memory).

- *Insertion Sort Loop invariant* :
  - Before the start of j^{th} iteration, $2 \leq j \leq n$, of the for loop of lines 1-8, the subarray A[1..j-1] consists of the elements originally in A[1..j-1] but in sorted order.

Correctness proof by LI

1. Decide on the appropriate LI
2. Prove that it holds before the loop starts: **Initialization**
3. Prove that if it holds before the i^{th} execution of the loop, then it will hold before the next $(i+1)^{th}$ execution as well: **Maintenance**
4. These two proofs together prove that the LI must hold after the last execution too, and this should show that the algorithm solves the problem correctly: **Termination**

Initialization

- Before the start of first iteration of the loop with j=2, the subarray A[1…1] consists of the elements originally in A[1...1] but in sorted order.
- Trivially true because A[1…1] is a one-element array which is by definition sorted!

Maintenance

1. Suppose that before the **start of j^{th} iteration** of the loop, the subarray A[1...j-1] consists of the elements originally in A[1...j-1] but in sorted order.
2. During the j^{th} iteration of the loop, the while loop (lines 5-7) compares A[j] with A[j-1], A[j-2] etc. until (i) either a number $A[k] \leq A[j]$ is found or (ii) it turns out that all numbers A[j-1], A[j-2]…A[2],A[1] are greater than A[j].
3. Each number found to be greater than A[j] is moved one cell to the right, i.e., if A[j-1]>A[j] then A[j-1] is moved to A[j] and so on.
4. If case (i) holds, then the numbers A[k+1],A[k+2],…A[j-1] are moved to cells A[k+2], A[k+3],…A[j] respectively and A[j] is placed (line 8) in A[k+1].
5. Since in this case A[1]…A[k] are not moved, by assumption (1) these are in the sorted order.
6. By the same assumption, A[k+1],A[k+2],…A[j-1] were in sorted order, so now A[k+2], A[k+3],…A[j] are in sorted order.
7. Since $A[k] \leq A[j]$ and A[j] is now in A[k+1], $A[k] \leq A[k+1]$.
8. Since A[k] is the first number found by the while loop such that $A[k] \leq A[j]$, we know that A[k+1] must have been greater than A[j]. So now $A[k+1] < A[k+2]$.
9. 7 & 8 imply that A[k], A[k+1] and A[k+2] are in sorted order.
10. 5, 6 & 9 imply that A[1]…A[j] are in sorted order at the **end of the j^{th} iteration**.

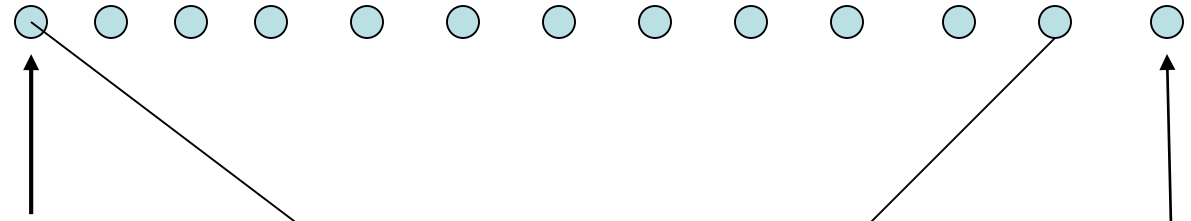
Maintenance

11. Suppose case (ii) holds instead, i.e., A[j-1], A[j-2]…A[2],A[1] are all greater than A[j].
12. Then the while loop will move A[j-1], A[j-2]…A[2],A[1] to A[j],A[j-1]…A[2] respectively, and place A[j] in A[1].
13. By assumption 1, A[j-1]…A[1] were in sorted order, so now A[j]…A[2] are in sorted order.
14. Since A[1] was greater than A[j], now A[2]>A[1].
15. 13 & 14 imply that A[1]…A[j] are in sorted order at the **end of the j^{th} iteration**.
16. So in both cases, at the end of the of j^{th} iteration of the loop, A[1]…A[j] will be in sorted order.
17. I.e., before the start of the of $(j+1)^{th}$ iteration of the loop, the subarray A[1…j] will be in sorted order and so the LI will be true.

Termination

1. Initialization showed that the LI will be true before the loop begins (i.e. before the first iteration of the loop with j=2).
2. Maintenance showed that if the LI was true before an iteration, it would still be true before the next iteration.
3. So LI must be true before the second iteration of the loop with j=3, before the third with j=4 and so on.
4. I.e., LI will be true before every iteration of the loop.
5. The loop ends after the n^{th} iteration, i.e., before the $(n+1)^{th}$ iteration. The LI must be true at that time.
6. LI: Before the start of j^{th} iteration, $2 \leq j \leq n$, of the for loop of lines 1-8, the subarray A[1..j-1] consists of the elements originally in A[1..j-1] but in sorted order.
7. I.e., before the start of $(n+1)^{th}$ iteration of the loop, the subarray A[1…n] will consist of the elements originally in A[1...n] but in sorted order.
8. Thus, this algorithm sorts the array correctly!

Inductive Proofs



Base case

1.Prove for the base case

2. Inductive hypothesis: Assume true for the first k items

3. Inductive step: Show that it holds for the k+1-th item

Example

- Fibonacci numbers
  - $F_0 = 1$; $F_1 = 1$; $F_n = F_{n-1}+F_{n-2}$
  - 1 1 2 3 5…

Prove $F_i < (5/3)^i$ for $i >= 1$

- Two base cases: F_1 and F_2 - why?
- Proving the base cases:
  - $F_1 = 1 < (5/3)^1$ and $F_2 = 2 < (5/3)^2 = 25/9$
- Inductive Hypothesis:
  - $F_i < (5/3)^i$ for $1 <= i <= k$ for some k
- Inductive step
  - Now we need to show that $F_{k+1} < (5/3)^{k+1}$

Inductive Step

- We know $F_k < (5/3)^k$ and $F_{k-1} < (5/3)^{k-1}$ from the inductive hypothesis
- We also know that $F_{k+1} = F_k + F_{k-1}$ by definition of the Fibonacci series
- So $F_{k+1} < (5/3)^k + (5/3)^{k-1} = (3/5) * (5/3)^{k+1} + (9/25) * (5/3)^{k+1} = (5/3)^{k+1} * (3/5 + 9/25) = (5/3)^{k+1} * (24/25) < (5/3)^{k+1}$ - done!

Proof by Induction

- Start by showing that the algorithm produces the correct outputs for one or a few initial inputs (proving the base cases)
- Then make an assumption that the algorithm produces the correct outputs for a finite set of k valid inputs (making the inductive hypothesis)
- Finally (and most importantly) show that the algorithm will produces the correct output for the next (k+1-th) input (proving the inductive step)
- Generally used to prove correctness of recursive algorithms

Thinking about Recursion

- What is a recursive algorithm?
- How can I understand (think about) a recursive algorithm?
  - Draw a Recursion Tree
- Is a recursive algorithm always efficient/inefficient?
- Why should I use a recursive algorithm?
- When should I not use a recursive algorithm?
  - avoid tail recursion!
  - avoid duplicated work!

Recursive Fibonacci Algorithm

function fib (n: non-negative integer)

1 if n=0 or 1 then return 1
  else
2 return fib(n-1)+fib(n-2)

- How does this algorithm compute F_5 when executed as fib(5)?

Find-Max

```
function find-max(A:array [i...j] of integer)
k: integer
1 if i=j then return A[i]
   else
2    k← find-max(A:array [i+1...j])
3    if k>A[i] then return k else return A[i]
```

Thinking about Recursion

- How does this algorithm finds the max?
- What are the legal inputs?
- What is the base case?
- Draw the recursion tree showing inputs and outputs if the input is the array [1,2,3,4,5]

Proving Correctness of Recursive Algorithms

To prove that find-max correctly finds the max number in an array of n integers:

1. Proving the Base Cases:

   If A is an array of length 1, by definition its element is the max.

   Statement 1 of the algorithm returns the single element if array size is 1, so the algorithm is correct for the base case input.

Correctness Proof (contd.)

2. Making the Inductive Hypothesis

   Assume the algorithm returns the correct max element for all arrays of size from 1 to k, for some constant k>=1.

Correctness Proof (contd.)

3. Proving the Inductive Step:

We need to prove that if the input A is an array of size k+1, the algorithm will return the max element in it.

Correctness Proof (contd.)

If k>=1 then k+1>1. So the input array contains more than one element, so j>i, i.e., the condition checking in S#1 will fail, and S#2-S#3 will be executed.

After S#2 is executed, the variable k will contain the max element in the subarray A[i+1…j] of size k (by the Inductive Hypothesis).

S#3 will choose and return the larger of A[i] or k.

I.e., the algorithm will return the larger of the first element of the array and the max element in the rest of the array when the input is of size k+1.

Correctness Proof (contd.)

The maximum of k+1 numbers is the same as the maximumum of the first number and the maximum of the other k numbers.

So when the input size is k+1 the algorithm will return the correct answer.

Correctness Proof (contd.)

Thus, we have shown that the algorithm:

- Produces the correct answer for the base case (array of size 1)

AND

- Assuming that it produces the correct answer for all arrays of size 1 to some k, it will produce the correct answer for an input array of size (k+1)

SO

- It will produce the correct answer for ALL input arrays of size 1 or more

Summary

How to prove that an algorithm is incorrect:
Come up with a counterexample
Or produce a proof by contradiction

How to prove that an algorithm is correct:
Produce a proof by contradiction

How to prove that a recursive algorithm is correct:
Produce an inductive proof

Thinking Assignments

1. Come up with a correct recursive algorithm for reversing a string
2. Write it in pseudocode
    1. How does your String-Reverse reverse strings?
    2. What if it gets the empty string as input?
    3. What if it gets a string of length 1?
    4. What if it gets a string of odd length? Even length? Does it matter?
    5. Draw the recursion tree if the input is the string "abcd"
3. Prove that it is correct using an inductive proof

Look at one set of answers to these questions in the following slides only after you have done them yourself!

Another Example: String-Reverse

```
function string-reverse(s:string)
{string-length, last-character, delete-last-character &
    concatenate are functions. Delete-last-character returns
    the input string without its last character; concatenate
    attaches its first argument to the beginning of the second
    argument and returns the resulting string}
lc: character; temp: string
1  if string-length(s)<=1 then return s
   else
2       lc←last-character(s)
3       temp←string-reverse(delete-last-character(s))
4       return concatenate(lc,temp)
```

Proving Correctness of Recursive Algorithms

To prove that string-reverse correctly reverses all strings:

1. Proving the Base Cases:

   If s is a string of length 0 or 1, by definition $s^{Reverse} = s$

   Statement 1 of the algorithm returns s if its length is 0 or 1, so the algorithm is correct for base case inputs.

Correctness Proof (contd.)

2. Making the Inductive Hypothesis

   Assume the algorithm returns the reverse of all strings of length from 0 to k, for some constant k>0.

Correctness Proof (contd.)

3. Proving the Inductive Step:

We need to prove that if the input s is a string of length k+1 (i.e. with 1^{st}, 2^{nd},..., k^{th} and $(k+1)^{th}$ characters), the algorithm returns its reverse.

Correctness Proof (contd.)

If k>0 then k+1>1, so the condition checking in S#1 will fail, and S#2-S#4 will be executed.

After S#2 is executed, the variable lc will contain the last – i.e. $(k+1)^{th}$ - character of s.

S#3: deletes last character of s, then calls string-reverse recursively on s, and assigns the returned value to temp.

After S#3 is executed, temp will contain a string of length k, containing the first k characters of s in reverse order (by the Inductive Hypothesis).

I.e. temp = “k^{th} $(k-1)^{th}$ ... 1^{st} characters of s”.

Correctness Proof (contd.)

S#4: concatenates (i.e. attaches) lc to the beginning of temp and then returns temp.

So after the concatenation temp will contain a string of length k+1.

I.e. temp = “$(k+1)^{th}$ k^{th} $(k-1)^{th}$ ... 1^{st} characters of s”

This is exactly the reverse of the input string s.

Correctness Proof (contd.)

Thus, we have shown that the algorithm:

- Produces the correct answer for the base cases (empty or 1-character strings)

AND

- Assuming that it produces the correct answer for all strings of length 0 to some k, it will produce the correct answer for a string of length (k+1)

SO

- It will produce the correct answer for ALL strings of length 0 or more

More Thinking Assignments

Algorithm efficiency depends on data structures too:

Develop recursive algorithms to reverse a string corresponding to these 4 strategies: pull off the last character and recursively reverse the rest, pull off the first character and recursively reverse the rest, swap the last and first characters and recursively reverse the rest, and a divide & conquer strategy - split the string into two halves and recursively reverse each half.

Make sure your algorithms work correctly for base cases, and even and odd string lengths.

A string can be represented either as an array of characters or as a linked list of characters with only a head pointer or with both a head and a tail pointer.

Which data structure best matches each algorithm? Notice how using the “wrong” data structure can decrease the efficiency of an algorithm.